Power Series Composition and Change of Basis

Alin Bostan  
Algorithms Project  
INRIA Rocquencourt  
France  
Alin.Bostan@inria.fr

Bruno Salvy  
Algorithms Project  
INRIA Rocquencourt  
France  
Bruno.Salvy@inria.fr

Éric Schost  
ORCCA and CSD  
University of Western Ontario  
London, ON, Canada  
eschost@uwo.ca

ABSTRACT

Efficient algorithms are known for many operations on truncated power series (multiplication, powering, exponential, \ldots). Composition is a more complex task. We isolate a large class of power series for which composition can be performed efficiently. We deduce fast algorithms for converting polynomials between various bases, including Euler, Bernoulli, Fibonacci, and the orthogonal Laguerre, Hermite, Jacobi, Krawtchouk, Meixner and Meixner-Pollaczek.

Categories and Subject Descriptors:
1.1.2 [Computing Methodologies]: Symbolic and Algebraic Manipulation – Algebraic Algorithms

General Terms: Algorithms, Theory

Keywords: Fast algorithms, transposed algorithms, basis conversion, orthogonal polynomials.

1. INTRODUCTION

Through the Fast Fourier Transform, fast polynomial multiplication has been the key to devising efficient algorithms for polynomials and power series. Using techniques such as Newton iteration or divide-and-conquer, many problems have received satisfactory solutions: polynomial evaluation and interpolation, power series exponentiation, logarithm, \ldots can be performed in quasi-linear time.

In this article, we discuss two questions for which such fast algorithms are not known: power series composition and change of basis for polynomials. We isolate special cases, including most common families of orthogonal polynomials, for which our algorithms reach quasi-optimal complexity.

Composition. Given a power series \( g \) with coefficients in a field \( K \), we first consider the map of evaluation at \( g \)

\[ \text{Eval}_{m,n}(\cdot, g) : A \in K[x]_m \mapsto A(g) \mod x^n \in K[x]_n. \]

Here, \( K[x]_m \) is the \( m \)-dimensional \( K \)-vector space of polynomials of degree less than \( m \). We note \( \text{Eval}_a \) for \( \text{Eval}_{a,n} \).

To study this problem, as usual, we denote by \( M \) a multiplication time function, such that polynomials of degree less than \( n \) can be multiplied in \( M(n) \) operations in \( K \). We impose the usual super-linearity conditions of [17, Chap. 8].

Using Fast Fourier Transform algorithms, \( M(n) \) can be taken in \( O(n \log(n)) \) operations over fields with suitable roots of unity, and \( O(n \log(n) \log \log(n)) \) in general [31, 14].

If \( g(0) = 0 \), the best known algorithm, due to Brent and Kung, uses \( O(\sqrt{n \log(n) M(n)}) \) operations in \( K \) [11]; in small characteristic, a quasi-linear algorithm is known [5]. There are however special cases of power series \( g \) with faster algorithms: evaluation at \( g = x^k \) takes linear time; evaluation at \( g = x^k \) requires no arithmetic operation. A non-trivial example is \( g = x + a \), which takes time \( O(M(n)) \) when the base field has characteristic zero or large enough [1]. Brent and Kung [11] also showed how to obtain a cost in \( O(M(n) \log(n)) \) when \( g \) is a polynomial; this was extended by van der Hoeven [22] to the case where \( g \) is algebraic over \( K(x) \). In \( \S 2 \), we prove that evaluation at \( g = \exp(x) - 1 \) and at \( g = \log(1 + x) \) can also be performed in \( O(M(n) \log(n)) \) operations over fields of characteristic zero or larger than \( n \).

Using associativity of composition and the linearity of the map \( \text{Eval}_{m,n} \), we show in \( \S 2 \) how to use these special cases as building blocks, to obtain fast evaluation algorithms for a large class of power series. This idea was first used by Pan [28], who applied it to functions of the form \( (ax + b)/(cx + d) \). Our extensions cover further examples such as \( 2x/(1 + x)^2 \) or \( (1 - \sqrt{-1})^2/x \), for which we improve the previously known costs.

Bivariate problems. Our results on the cost of evaluation (and of the transposed operation) are applied in \( \S 3 \) to special cases of a more general composition, reminiscent of umbral operations [30]. Given a bivariate power series \( F = \sum_{j \geq 0} \xi_j(x) t^j \), we consider the linear map

\[ \text{Eval}_a(\cdot, F, t) : (a_0, \ldots, a_{n-1}) \mapsto \sum_{j<n} \xi_j(x) a_j \mod x^n. \]

For instance, with

\[ F = \frac{1}{1 - tg(x)} = \sum_{j \geq 0} g(x) t^j, \]

this is the map \( \text{Eval}_a(\cdot, g) \) seen before. For general \( F \), the conversion takes quadratic time (one needs \( n^2 \) coefficients for \( F \)). Hence, better algorithms can only been found for structured cases; in \( \S 3 \), we isolate a large family of bivariate series \( F \) for which we can provide such fast algorithms. This approach follows Frumkin’s [16], which was specific to Legendre polynomials.

Change of basis. Our framework captures in particular the generating series of many classical polynomial families,
for which it yields at once conversion algorithms between the monomial and polynomial bases, in both directions.

Thus, we obtain in §4 change of basis algorithms of cost only \(O(M(n))\) for all of Jacobi, Laguerre and Hermite orthogonal polynomials, as well as Euler, Bernoulli, and Mott polynomials (see Table 3). These algorithms are derived in a uniform manner from our composition algorithms; they improve upon the existing results, of cost \(O(M(n) \log(n))\) or \(O(M(n) \log^2(n))\) at best (see below for historical comments). We also obtain \(O(M(n) \log(n))\) conversion algorithms for a large class of Sheffer sequences [30, Chap. 2], including antu-...
is described in the third column. In the case of powering, it is assumed that \( k \in \mathbb{N}_{>0} \). Here and in what follows, we freely identify \( \mathbb{K}[x]/m \) and \( \mathbb{K}^m \), through the isomorphism

\[
\sum_{i<m} a_i x^i \in \mathbb{K}[x]_m \iff (a_0, \ldots, a_{m-1}) \in \mathbb{K}^m.
\]

All of the cost estimates are straightforward, except for the shift, which, in characteristic 0, can be deduced from the other ones by the factorization [1]:

\[
\text{Shift}_{a,m} = \Delta_m(\text{Rev}_m(\text{Mul}_m(\text{Rev}_m(\Delta_m(\ldots, P)), P)), 1/\text{i}l),
\]

where \( P \) is the polynomial \( \sum_{i=0}^{\text{degree}} a_i x^i \). We continue with some equally simple operators, whose description however requires some more detail. For \( k \in \mathbb{N}_{>0} \), any polynomial \( A \) in \( \mathbb{K}[x] \) can be uniquely written as

\[
A(x) = A_0(k) x^k + A_1(k) x^k x + \cdots + A_{k-1}(k) x^k x^{k-1}.
\]

Inspecting degrees, one sees that if \( A \) is in \( \mathbb{K}[x]_m \), then \( A_{i/k} \) is in \( \mathbb{K}[x]_m \), with

\[
m_i = \left\lceil \frac{m}{k} \right\rceil + \begin{cases} 1 & \text{if } i \leq m \mod k, \\ 0 & \text{otherwise}. \end{cases}
\]

This leads us to define the map \( \text{Split}_{a,k} : \mathbb{K}[x]_m \mapsto (A_0(k), \ldots, A_{k-1}(k)) \in \mathbb{K}[x]_{m0} \times \cdots \times \mathbb{K}[x]_{mk-1} \).

It uses no arithmetic operation. We also use linear combination with polynomial coefficients. Given polynomials \( G_0, \ldots, G_{k-1} \) in \( \mathbb{K}[x]_m \), we denote by

\[
\text{Comb}_{a_m}(G_0, \ldots, G_{k-1}) : \mathbb{K}[x]_m^k \to \mathbb{K}[x]_m
\]

the map sending \( (A_0, \ldots, A_{k-1}) \in \mathbb{K}[x]_m^k \) to

\[
A_0 G_0 + \cdots + A_{k-1} G_{k-1} \mod x^m \in \mathbb{K}[x]_m.
\]

It can be computed in \( O(km(m)) \) operations. Finally, we extend our set of subroutines on polynomials with the following new results on the evaluation at \( \exp(x) - 1 \) and \( \log(1 + x) \).

**Proposition 1.** The maps

\[
\text{Exp}_{a,m} : A \in \mathbb{K}[x]_m \mapsto A(\exp(x) - 1) \mod x^n \in \mathbb{K}[x]_n,
\]

\[
\text{Log}_{a,m} : A \in \mathbb{K}[x]_m \mapsto A(\log(1 + x)) \mod x^n \in \mathbb{K}[x]_n
\]

can be computed in \( O(M(n) \log(n)) \) arithmetic operations.

**Proof.** We start by truncating \( A \) modulo \( x^n \), since

\[
\text{Exp}_{a,m}(A) = \text{Exp}_{a,m}(A \mod x^n).
\]

After shifting by \(-1\), we are left with the question of evaluating a polynomial in \( \mathbb{K}[x]_m \) at \( \sum_{i<n} x^i \). Writing its matrix shows that this map factors as \( \Delta_m(\text{MultiEval}(1, 1/\text{i}l)) \), where \( \text{MultiEval} \) is the map

\[
A \in \mathbb{K}[x]_m \mapsto \left( (A(0), \ldots, A(n-1)) \right) \in \mathbb{K}^n.
\]

To summarize, we have obtained that

\[
\text{Exp}_{a,m}(A) = \Delta_m(\text{MultiEval}_{\text{i}l}(\text{Shift}_{-1,m}(\text{mod}_n(\text{Exp}_{a,m}(A)))), 1/\text{i}l).
\]

Using fast transposed evaluation [13, 7], \( \text{Exp}_{a,m}(A) \) can thus be computed in \( O(M(n) \log(n)) \) operations. Inverting these computations leads to the factorization

\[
\text{Log}_{a,m}(A) = \text{Shift}_{0, \text{i}l}(\text{Interp}_{\text{i}l}(\Delta_m(\text{mod}_n(A), 1/\text{i}l))),
\]

where \( \text{Interp}_{\text{i}l} \) is interpolation at \( 0, \ldots, n - 1 \). Using algorithms for transpose interpolation [24, 7], this operation can be done in time \( O(M(n) \log(n)) \).

### 2.2 Associativity Rules

For each basic power series operation in Table 1, we now express \( \text{Eval}_{a,m}(A, o(g)) \) in terms of simpler operations; we call these descriptions *associativity rules*. We write them in a formal manner: this formalism is the key to automatically design complex composition algorithms, and makes it straightforward to obtain transposed associativity rules, required in the next section. Most of these rules are straightforward; care has to be taken regarding truncation, though.

**Scaling, Shift and Powering.**

\[
\begin{align*}
\text{Eval}_{a,m}(A, M (g)) &= \text{Eval}_{a,m}(\text{Scale}_{a,m}(A), g), \quad (A1) \\
\text{Eval}_{a,m}(A, A_n(g)) &= \text{Eval}_{a,m}(\text{Shift}_{a,m}(A), g), \quad (A2) \\
\text{Eval}_{a,m}(A, P_k(g)) &= \text{Eval}_{k(m-1)+1, a}(\text{Power}_{k, a}(A), g), \quad (A3)
\end{align*}
\]

**Inversion.** From \( A(1/g) = (\text{Rev}_m(A)) / g^{m-1} \) and writing \( h = g^{1-m} \mod x^n \), we get

\[
\text{Eval}_{a,m}(A, \text{Inv}(g)) = \text{Mul}_{a,m}(\text{Eval}_{a,m}(\text{Rev}_m(A), g), h). \quad (A4)
\]

**Root taking.** For \( g \) and \( h \) in \( \mathbb{K}[x] \), if \( g = h^k \), one has \( A(h) = A_{0/k}(g) + A_{1/k}(g) h + \cdots + A_{k-1/k}(g) h^{k-1} \). We deduce the following rule, where the indices \( m_i \) are defined in Equation (1).

\[
\begin{align*}
\text{Eval}_{a,m}(A, \text{R}_{k,a}(g)) &= \text{Comb}_{a_m}(B_0, \ldots, B_{k-1}, 1, \ldots, h_{k-1})). \quad (A5)
\end{align*}
\]

**Exponential and Logarithm.**

\[
\begin{align*}
\text{Eval}_{a,m}(A, E(g)) &= \text{Eval}_{a,m}(\text{Exp}_{a,m}(A), g), \quad (A6) \\
\text{Eval}_{a,m}(A, L(g)) &= \text{Eval}_{a,m}(\text{Log}_{a,m}(A), g). \quad (A7)
\end{align*}
\]

### 2.3 Composition sequences

We now describe more complex evaluations schemes, obtained by composing the former basic ones.

**Definition 1.** Let \( O \) be the set of actions from Table 1. A sequence \( o = (o_1, \ldots, o_k) \) with entries in \( O \) is defined at a series \( g \in \mathbb{K}[x] \) if \( g \) is in \( \text{dom}(o) \), and for \( i \leq L, o_{i-1}(\cdots (o_1(g)) \) is in \( \text{dom}(o) \). It is a composition sequence if it is defined at \( x \); in this case, \( o \) computes the power series \( g_1, \ldots, g_L \), with \( g_0 = x \) and \( g_i = o_i(g_{i-1}) \); it outputs \( g_L \).

**Examples.** As mentioned in [28], the rational series \( g = (ax + b)/(cx + d) \in \mathbb{K}[x] \), with \( cd \neq 0 \), decomposes as

\[
\frac{ax + b}{cx + d} = e \frac{c}{cx + d} + f + \frac{e - ad}{c} \text{ and } f = \frac{a}{c}.
\]
This shows that $g$ is output by the composition sequence $(M_c, A_d, \text{Inv}(M_c), A_f)$. A more complex example is
$$g = \frac{2x}{(1+x)^2} = \frac{1}{2} \left(1 - \left(1 - \frac{2}{1+x}ight)^2\right),$$
which shows that $g$ is output by the composition sequence $(A_1, \text{Inv}(M_{-2}), A_1, P_2, M_{-1}, A_1, M_{1/2})$.

Finally, consider $g = \log((1+x)/(1-x))$. Using
$$g = \log \left(1 + \left(-2 - \frac{2}{x-1}\right)\right),$$
we get the composition sequence $(A_{-1}, \text{Inv}(M_{-2}), A_{-2}, L)$.

Computing the associated power series. Our main algorithm requires truncations of the series $g_1, \ldots, g_{12}$ associated to a composition sequence. The next lemma discusses the cost of their computation. In all complexity estimates, the composition sequence $\circ$ is fixed; hence, our estimates hide a dependency in $\circ$ in their constant factors.

**Lemma 1.** If $\circ = (\circ_1, \ldots, \circ_L)$ is a composition sequence that computes power series $g_1, \ldots, g_{12}$, one can compute all $g_i \mod x^n$ in time $O(M(n))$.

**Proof.** All operators in $O$ preserve the precision, except for root-taking, since the operator $R_{k, \alpha, r}$ loses $r(k-1)$ terms of precision. For $i \leq L$, define $\varepsilon_i = r(k-1)$ if $\circ_i$ has the form $R_{k, \alpha, r}$, otherwise, and define $n_i = n$ and inductively $n_{i-1} = n_i + \varepsilon_i$. Starting the computations with $g_0 = x$, we iteratively compute $g_i \mod x^n$ from $g_{i-1} \mod x^{n_{i-1}}$.

Inspecting the list of possible cases, one sees that computing $g_i$ always takes time $O(M(n_{i-1}))$. For powering, this estimate is valid because we disregard the dependency in $\circ$ otherwise, terms of the form $\log(k)$ would appear. For the same reason, $O(M(n_{i-1}))$ is in $O(M(n))$, as is the total cost, obtained by summing over all $i$.

Composition using composition sequences. We now study the cost of computing the map $\text{Eval}(\cdot, g)$, assuming that $g \in K[[x]]$ is output by a composition sequence $\circ$. The cost depends on the operations in $\circ$. To keep simple expressions, we distinguish two cases: if $\circ$ contains no operation $E$ or $L$, we let $T_\circ(n) = M(n)$; otherwise, $T_\circ(n) = M(n) \log(n)$.

**Theorem 1 (Composition).** Let $\circ = (\circ_1, \ldots, \circ_L)$ be a composition sequence that outputs a series $g \in K[[x]]$. Given $\circ$, one can compute the map $\text{Eval}_\circ(\cdot, g)$ in time $O(T_\circ(n))$.

**Proof.** We follow the algorithm of Figure 1. The main function first computes the sequence $G = g_1, \ldots, g_L$ modulo $x^n$, using a subroutine $\text{ComputeG}(o, n)$ that follows Lemma 1. The cost $O(M(n))$ of this operation is in $O(T_\circ(n))$. Then, we call the auxiliary $\text{Evalaux}$ function.

On input $A, m, n, \ell, o, G$, this latter function computes $\text{Eval}_{m,n}(A, g)$, which is done recursively, applying the appropriate associativity rule $(A_1) \to (A_2)$; the pseudo-code uses a C-like switch construct to find the matching case. Even if the initial polynomial $A$ is in $K[[x]]$, this may not be the case for the arguments passed to the next calls; hence the need for the extra parameter $m$. For root-taking, the subroutine $\text{FindDegrees}$ computes the quantities $m_i$ of Eq. (1).

Since we write the complexity as a function of $n$, the cost analysis is simple: even if several recursive calls are generated ($k$ for $k$th root-taking), their total number is still $O(1)$.

---

**Figure 1:** Algorithm $\text{Eval}$.

Similarly, the degree of the argument $A$ passed through the recursive calls may grow, but only like $O(n)$.

Two kinds of operations contribute to the cost: precomputations of $g_{1-m} \mod x^n$ (for $\text{Inv}$) or of $1, g_1, \ldots, g_{k-1} \mod x^n$ for $R_{k, \alpha, r}$, and linear operations on $A$: shifting, scaling, multiplication ... The former take $O(M(n))$, since the exponents involved are in $O(n)$. The latter operations take $O(M(n))$ if no $\text{Exp}$ or $\text{Log}$ operation is performed, and $O(M(n) \log(n))$ otherwise. This concludes the proof.

**2.4 Inverse map**

The map $\text{Eval}(\cdot, g)$ is invertible if and only if $g'(0) \neq 0$ (hereafter, $g'$ is the derivative of $g$). We discuss here the computation of the inverse map.

**Theorem 2 (Inverse).** Let $\circ = (\circ_1, \ldots, \circ_L)$ be a composition sequence that outputs $g \in K[[x]]$ with $g'(0) \neq 0$. One can compute the map $\text{Eval}_\circ^{-1}(\cdot, g)$ in time $O(T_\circ(n))$.

**Proof.** If $h$ is the power series $h = \sum_{i \geq h_0} h_i x^i$, with $h_0 \neq 0$, $\text{val}(h) = h_0$ is the valuation of $h$, $\text{lc}(h) = h_{i=0}$ its leading coefficient and $\text{lt}(h) = h_{i=0}$ its leading term. We also introduce an equivalence relation on power series: $g \sim h$ if $g(0) = h(0)$ and $\text{lt}(g - g(0)) = \text{lt}(h - h(0))$. The proof of the next lemma is immediate by case inspection.

**Lemma 2.** For $\circ$ in $O$, if $h \sim g$ and $g$ is in $\text{dom}(\circ)$, then $h$ is in $\text{dom}(\circ)$ and $\circ(h) \sim \circ(g)$.

**Series tangent to the identity.** We prove the proposition in two steps. For series of the form $g = x \mod x^2$, it suffices
to “reverse” step-by-step the computation sequence for $g$. The following lemma is crucial.

**Lemma 3.** Let $g$ be in $\mathbb{K}[[x]]$, with $g = x \bmod x^2$, and let $\sigma = (\sigma_1, \ldots, \sigma_L)$ be a sequence defined at $g$. Then $\sigma$ is a composition sequence.

**Proof.** We have to prove that $\sigma$ is defined at $x$, i.e., that all of $\sigma_1(x), \sigma_2(x), \ldots$ are well-defined. This follows by applying the previous lemma inductively. □

We can now work on the inverse property proper. Let thus $\sigma = (\sigma_1, \ldots, \sigma_L)$ be a computation sequence, that computes $g_1, \ldots, g_L$ and outputs $g = g_L$, with $g = x \bmod x^2$. We define operations $\tilde{\sigma}_i, \ldots, \tilde{\sigma}_L$ through the following table (note that we reverse the order of the operations):

| Operation | $\tilde{\sigma}_i$ |
|-----------|------------------|
| Add       | $A_{\lambda}$    |
| Mul       | $M_{\lambda}$    |
| Powering  | $P_k$            |
| Root      | $R_{k,a,r}$      |
| Inverse   | $\text{inv}$     |
| Exp.      | $E$              |
| Log.      | $L$              |

**Lemma 4.** The sequence $\tilde{\sigma} = (\tilde{\sigma}_1, \ldots, \tilde{\sigma}_L)$ is a composition sequence and outputs a series $\tilde{g}$ such that $\tilde{g}(g) = x$.

**Proof.** One sees by induction that for all $i$, $\tilde{\sigma}_{i-1}(\cdots(\tilde{\sigma}_1(g))$ is in $\text{dom}(\tilde{\sigma}_i)$ and $\tilde{\sigma}_i(\cdots(\tilde{\sigma}_1(g)) = g_{L-i}$. This shows that the sequence $\tilde{\sigma}$ is defined at $g$ and that $\tilde{\sigma}_L(\cdots(\tilde{\sigma}_1(g)) = x$. From Lemma 3, we deduce that $\tilde{\sigma}$ is defined at $x$. Letting $\tilde{g}$ be the output of $\tilde{\sigma}$, the previous equality gives $\tilde{g}(g) = x$, which concludes the proof. □

Since $T_0 = T_1$, and in view of Theorem 1, the next lemma concludes the proof of Theorem 2 in the current case.

**Lemma 5.** With $g$ and $\tilde{g}$ as above, the map $\text{Eval}_{\tilde{\sigma}}(\cdot, \tilde{g})$ is the inverse of $\text{Eval}_{\sigma}(\cdot, g)$.

**Proof.** Let $F$ be in $\mathbb{K}[x]_n$ and let $G = \text{Eval}_{\tilde{\sigma}}(F, g)$, so that $F(g) = G + H$, with $\text{val}(H) \geq n$. Evaluating at $\tilde{g}$, we get $F = G(\tilde{g}) + H(\tilde{g}) = G(\tilde{g}) \bmod x^n$, since $\text{val}(\tilde{g}) = 1$. □

**General case.** Lemma 5 fails when $\text{val}(g) = 0$. We can however reduce the general case to that where $\text{val}(g) = 1$. Let us write $g = g_0 + g_1 x + \ldots$, with $g_1 \neq 0$, and define $\tilde{g} = (g - g_0)/g_1$, so that $\tilde{g} = x \bmod x^2$. If $\sigma$ is a composition sequence for $g$, then $\tilde{\sigma} = (\sigma, A_{-g_0}, M_{1/g_1})$ is a composition sequence for $\tilde{g}$, and we have $T_0 = T_1$. Thus, by the previous point, we can use this composition sequence to compute the map $\text{Eval}_{\tilde{\sigma}}(\cdot, \tilde{g})$ in time $O(T_0(n))$. From the equality

$$\text{Eval}_{\tilde{\sigma}}(A, g) = \text{Eval}_{\tilde{\sigma}}(\text{Scale}_{g_0,n}(\text{Shift}_{g_0,n}(A)), \tilde{g}),$$

we deduce

$$\text{Eval}_{\tilde{\sigma}}^{-1}(A, g) = \text{Shift}_{-g_0,n}(\text{Scale}_{1/g_1,n}(\text{Eval}_{\tilde{\sigma}}^{-1}(A, g))).$$

Since scaling and shifting induce only an extra $O(M(n))$ arithmetic operations, this finishes the proof of Theorem 2.

### 3. CHANGE OF BASIS

This section applies our results on composition to change of basis algorithms, between the monomial basis ($x^i$) and various families of polynomials ($P_i$), with $\text{deg}(P_i) = i$, for which we reach quasi-linear complexity. As an intermediate step, we present a bivariate evaluation algorithm.

#### 3.1 Main Theorem

Let $F \in \mathbb{K}[x, t]$ be the bivariate power series

$$F = \sum_{i,j \geq 0} F_{i,j} x^i t^j = \sum_{i \geq 0} \xi_i(x) t^i.$$

Associated with $F$, we consider the map

$$\text{Eval}_{\tilde{\sigma}}(\cdot, F, t) : (a_0, \ldots, a_{n-1}) \mapsto \sum_{i < n} \xi_i(x) a_j \bmod x^n.$$

The matrix of this map is $[F_{i,j}]_{i,j < n}$. The following theorem shows that for a large class of series $F$, the operation $\text{Eval}_{\tilde{\sigma}}(F, t)$ and its inverse can be performed efficiently. The proof relies on a transposition argument, given in §3.3.

**Theorem 3** (Main theorem). Let $f, g, h, u, v \in \mathbb{K}[[z]]$ be such that

- $g$ and $h$ are given by composition sequences $o_g$ and $o_h$;
- $f, u$ and $v$ can be computed modulo $z^n$ in time $T(n)$;
- $g(0) h(0) = 0$ and $g'(0), h'(0), u(0), v(0)$ are non-zero;
- all coefficients of $f$ are non-zero.

Then the series $F(x, t) = u(x) v(t) f(g(x) h(t))$ is well-defined. Besides, one can compute the map $\text{Eval}_{\tilde{\sigma}}(F, F', t)$ and its inverse in time $O(T(n) + T_{a_0}(n) + T_{a_0}(n))$.

**Proof.** Write $f = \sum_{k \geq 0} f_k z^k$,

$$g(x)^k = \sum_{i \geq 0} g_{k,i} x^i$$

and $h(t)^k = \sum_{j \geq 0} h_{k,j} t^j$.

Since $g(0) h(0) = 0$, we have that either $h_{k,j} = 0$ for $k > j$, or $g_{k,i} = 0$ for $k > i$. Thus, the coefficient $F'_{i,j}$ of $F'$ is well-defined and

$$F'_{i,j} = \sum_{k \leq n} f_k g_{k,i} h_{k,j}.$$

These coefficients are those of a product of three matrices, the middle one being diagonal; we deduce the factorization

$$\text{Eval}_{\tilde{\sigma}}(., F', t) = \text{Eval}_{\tilde{\sigma}}(., g) \circ \Delta_n(., f) \circ \text{Eval}_{\tilde{\sigma}}(., h).$$

The assumptions on $f, g$ and $h$ further imply that the map $\text{Eval}_{\tilde{\sigma}}(., F', t)$ is invertible, of inverse

$$\text{Eval}_{\tilde{\sigma}}^{-1}(., F', t) = \text{Eval}_{\tilde{\sigma}}^{-1}(., h) \circ \Delta_n(., f^{-1}) \circ \text{Eval}_{\tilde{\sigma}}^{-1}(., g).$$

By Theorems 1 and 2, as well as Theorem 4 stated below, $\text{Eval}_{\tilde{\sigma}}(F', F'', t)$ and its inverse can thus be evaluated in time $O(T(n) + T_{a_0}(n) + T_{a_0}(n))$. Now, from the identity $F = u(x)v(t)F'$, we deduce that

$$\text{Eval}_{\tilde{\sigma}}(F, F, t) = \text{Eval}_{\tilde{\sigma}}(., u) \circ \text{Eval}_{\tilde{\sigma}}(., F', t) \circ \text{Eval}_{\tilde{\sigma}}(., v).$$

Our assumptions on $u$ and $v$ make this map invertible, and

$$\text{Eval}_{\tilde{\sigma}}^{-1}(F, F, t) = \text{Eval}_{\tilde{\sigma}}^{-1}(., F', t) \circ \text{Eval}_{\tilde{\sigma}}^{-1}(., u) \circ \text{Eval}_{\tilde{\sigma}}^{-1}(., v),$$

with $a(x) = 1/u \bmod x^n$ and $b(t) = 1/v \bmod t^n$. The extra costs induced by the computation of $u, v, t$, their inverses, and the truncated products fit in $O(T(n) + M(n))$. □

#### 3.2 Change of Basis

To conclude, we consider polynomials $(P_i)_{i \geq 0}$ in $\mathbb{K}[x]$, with $\text{deg}(P_i) = i$, with generating series defined in terms of series $u, v, f, g, h$ as in Theorem 3 by

$$P = \sum_{i \geq 0} P_i(x) t^i = u(x) v(t) f(g(x) h(t)).$$
Proof. Let \( \mathcal{O} \) be a composition sequence that outputs \( g \in \mathcal{K}[x] \). Given \( \mathcal{O} \), one can compute the map \( \text{Eval}_{\lambda,\mathcal{O}}(A, g) \) in time \( O(T_\mathcal{O}(n)) \).

**Theorem 4** (Transposition). Let \( \mathcal{O} = (\mathcal{O}_1, \ldots, \mathcal{O}_L) \) be a composition sequence that outputs \( g \in \mathcal{K}[x] \). Given \( \mathcal{O} \), one can compute the map \( \text{Eval}_{\lambda,\mathcal{O}}(A, g) \) in time \( O(T_\mathcal{O}(n)) \).

**Proof.** This result follows directly from the transposition principle. However, we give an explicit construction of the transposed map \( \text{Eval}_{\lambda,\mathcal{O}}(A, g) \) in Figure 2. Non-linear precomputations are left unchanged. The terminal case \( \ell = 0 \) is dealt with by noting that the transpose of \( \text{mod}_{a,n} \) is \( \text{mod}_{a,n} \). To conclude, it suffices to give transposed associativity rules for our basic operators. The formal approach we use to write our algorithms pays off now, as it makes this transposition process automatic.

Recall that our algorithms deal with polynomials. The dual of \( \mathcal{K}[x]_m \) can be identified with \( \mathcal{K}[x]_m^\ast \) itself: to a \( \mathcal{K} \)-linear form \( \ell \) over \( \mathcal{K}[x]_m \), one associates \( \sum_{i < m} \ell(x^i) t^i \). Hence, transposed versions of algorithms acting on polynomials are seen to act on polynomials as well. Remark also that diagonal operators are their own transposes.

**Multiplication.** In [7], following [19], details of the transposed versions of plain, Karatsuba and FFT multiplications are given, with a cost matching that of the direct product. Without relying on such techniques, by writing down the multiplication matrix, one sees that \( \text{Mul}_{\lambda,\mathcal{O}}(A, .) \) is

\[
A \in \mathcal{K}[x]_m \mapsto (A \text{Rev}_{d+1}(P) \mod x^{n+d}) \div x^d \in \mathcal{K}[x]_n,
\]

if \( P \) has degree \( d \). Using standard multiplication algorithms, this formulation leads to slower algorithms than those of [7]. However, in our usage cases, \( n, m \) and \( d \) are of the same order of magnitude, and only a constant factor is lost.

**Scale.** The operator \( \text{Scale}_{\lambda,\mathcal{O}} \) is diagonal; through transposition, the associativity rule becomes:

\[
\text{Eval}_{\lambda,\mathcal{O}}(A, M_A(g)) = \text{Scale}_{\lambda,\mathcal{O}}(\text{Eval}_{\lambda,\mathcal{O}}(A, g)).
\]

**Shift.** The transposed map \( \text{Rev}_{\mathcal{O}} \) of the reversal operator coincides with \( \text{Rev}_{\mathcal{O}} \) itself, since this operator is symmetric. By transposing the identity for \( \text{Shift} \), we deduce

\[
\text{Shift}_{\lambda,\mathcal{O}}(A) = \Delta_{\mathcal{O}}(\text{Rev}_{\mathcal{O}}(\text{Mul}_{\lambda,\mathcal{O}}(\text{Rev}_{\mathcal{O}}(A, 1/1)), P), 1)).
\]

This algorithm for the transpose operation, though not described as such, was already given in [18]. This yields:

\[
\text{Eval}_{\lambda,\mathcal{O}}(A, A_\mathcal{O}(g)) = \text{Shift}_{\lambda,\mathcal{O}}(\text{Eval}_{\lambda,\mathcal{O}}(A, g)).
\]

**Powering.** The dual map \( \text{Power}_{\mathcal{O}} \) maps \( A \in \mathcal{K}[x]_{\ell_0} \) to \( A_{\ell_0,k} \in \mathcal{K}[x]_k \) (with the notation of §2.1). We deduce:

\[
\text{Eval}_{\lambda,\mathcal{O}}(A, P_k(g)) = \text{Power}_{\mathcal{O}}(\text{Eval}_{\lambda,\mathcal{O}}(A, g), k).
\]

**Inversion.** The transposed version of the rule for \( \text{Inv} \) is

\[
\text{Eval}_{\lambda,\mathcal{O}}(A, \text{Inv}(g)) = \text{Rev}_{\mathcal{O}}(\text{Eval}_{\lambda,\mathcal{O}}(\text{Mul}_{\lambda,\mathcal{O}}(\text{Inv}(g), k, m)), k, m)).
\]

**Root taking.** Considering its matrix, one sees that \( \text{Split}_{\mathcal{O}} \) maps \( (A_0, \ldots, A_{k-1}) \in \mathcal{K}[x]_0 \times \cdots \times \mathcal{K}[x]_{m-1} \) to

\[
A_0(x^k) + A_1(x^k) x + \cdots + A_{k-1}(x^k) x^{k-1} \in \mathcal{K}[x]_m.
\]

Besides, since the map \( \text{Comb} \) is the direct sum of the maps \( \text{Mul}_{\lambda,\mathcal{O}}(\mathcal{O}, G_i) : \mathcal{K}[x]_n \to \mathcal{K}[x]_n \), its transpose \( \text{Comb}_{\lambda,\mathcal{O}}(\mathcal{O}, G_0, \ldots, G_{k-1}) \) sends \( A \in \mathcal{K}[x]_n \) to

\[
(\text{Mul}_{\lambda,\mathcal{O}}(\mathcal{O}, G_i))_{0 \leq i < k} = \mathcal{K}[x]_n^k.
\]

Putting this together gives the transposed associativity rule

\[
\text{Eval}_{\lambda,\mathcal{O}}(A, g) = \text{Eval}_{\lambda,\mathcal{O}}(A, g).
\]
Exponential and Logarithm. From the proof of Proposition 1, we deduce the transposed map of $\exp_{m,n} \circ \log_{m,n}$ and their associativity rules

\[
\exp_{m,n}(A) = \mod_{m,n}(\shift_{1,n}(\mul\eval(\Delta_n(A, 1/t)))), \\
\eval_{m,n}(A, E(g)) = \exp_{m,n}(\eval_{m,n}(A, g)); \quad (A'_1) \\
\log_{m,n}(A) = \mod_{m,n}(\interp_{n}(\shift_{1,n}(A), t)), \\
\eval_{m,n}(A, L(g)) = \log_{m,n}(\eval_{m,n}(A, g)). \quad (A'_2)
\]

4. APPLICATIONS

Many generating functions of classical families of polynomials fit into our framework. To obtain conversion algorithms, it is sufficient to find suitable composition sequences. Table 3 lists families of polynomials for which conversions can be done in time $O(M(n))$ with our method (see e.g. [30, 3] for more on these classical families). In Table 4, a similar list is given, leading to conversions of cost $O(M(n) \log n)$; most of these entries are actually Sheffer sequences. Many other families can be obtained as special cases (e.g., Gegenbauer, Legendre, Chebyshev, Mittag-Leffler, etc).

The entry marked by $\ast$ is from [18]; the entries marked by $\ast\ast$ are orthogonal polynomials, for which one conversion (from the orthogonal to the monomial basis) is already given for the conversion from orthogonal to monomial bases; most of these entries are actually Sheffer sequences. Many other families can be obtained as special cases (e.g., Gegenbauer, Legendre, Chebyshev, Mittag-Leffler, etc).

In all cases, the pre-multiplier $u(x)v(t)$ depends on $t$ only and can be computed at precision $n$ in time $O(M(n))$; all our functions $f$ can be expanded at precision $n$ in time $O(n)$. Regarding the functions $g(x)$ and $h(t)$, most entries are easy to check; the only explanations needed concern some series $h(t)$. Rational functions are covered by the first example of §2.3; the second example of §2.3 deals with Jacobi polynomials and Spedal polynomials; the last example of §2.3 shows how to handle functions with logarithms. For Fibonacci polynomials, the function $h(t) = t/(1-t^2)$ satisfies

\[
(2h t)^2 = \left(\frac{1 + t^2}{1 - t^2}\right)^2 - 1.
\]

From this, we deduce the sequence for $h$:

\[
(P_2, M_{-1}, A_1, \linv, M_2, A_{-1}, P_2, A_{-1}, R_{2,2,1}, M_{1/2}).
\]

For Mott polynomials the series $h(t) = (1 - \sqrt{1 - t^2})/t$ can be rewritten as

\[
h = \sqrt{\frac{2}{1 + \sqrt{1 - t^2}}} - 1.
\]

This yields the composition sequence

\[
(P_2, M_{-1}, A_1, R_{2,1,0}, A_1, \linv, M_2, A_{-1}, R_{2,1,0}).
\]

5. EXPERIMENTS

We implemented the algorithms for change of basis using NTL [32]; the experiments are done for coefficients defined modulo a 40 bit prime, using the $\ZZ_p$ NTL class (our algorithms still work for degrees small with respect to the characteristic). All timings reported here are obtained on a Pentium M, 1.73 Ghz, with 1 GB memory.

Our implementation follows directly the presentation of the former sections. We use the transposed multiplication implementation of [7]. The Newton iteration for inverse is built-in in NTL; we use the standard Newton iteration for square root [10]. Exponentials are computed using the algorithm of [20]. Powers are computed through exponential and logarithm [10], except when the arguments are binomials, when faster formulas for binomial series are used. For evaluation and interpolation at $0, \ldots, n-1$, and their transposes, we use the implementation of [7].

We use the Jacobi and Mittag-Leffler orthogonal polynomials (a special case of Meixner polynomials, with $\beta = 0$ and $c = -1$), with the composition sequences of §2.3. Our algorithm has cost $O(M(n))$ for the former and $O(M(n) \log(n))$ for the latter. We compare this to the naive approach of quadratic cost in Figure 3 and 4, respectively. Timings are given for the conversion from orthogonal to monomial bases; those for the inverse conversion are similar.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig3.png}
\caption{Jacobi polynomials.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig4.png}
\caption{Mittag-Leffler polynomials.}
\end{figure}

Our algorithm performs better than the quadratic one. The crossover points lie between 100 and 200; this large value is due to the constant hidden in our big-Oh estimates: in both cases, there is a contribution of about $20M(n)$, plus an additional $M(n) \log(n)$ for Mittag-Leffler.

6. DISCUSSION

This article provides a flexible framework for generating new families of conversion algorithms: it suffices to add new composition operators to Table 1 and provide the corresponding associativity rules. Still, several questions need further investigation. Several of the composition sequences we use are non-trivial; this raises in particular the questions of characterizing what functions can be computed by a composition sequence, and of determining such sequences algorithmically. Besides, the costs of our algorithms are measured only in terms of arithmetic operations; the questions of numerical stability (for floating-point computations) or of coefficient size (when working over $\QQ$) require further work.

Acknowledgments. We thank ANR Gecko, the joint Inria-Microsoft Research Lab and NSERC for financial support.
Table 3: Polynomials with conversion in $O(M(n))$

| polynomial | generating series | $u(x)v(t)$ | $f(z)$ | $g(x)$ | $h(t)$ |
|------------|-------------------|-------------|--------|--------|--------|
| Laguerre $L_n$ | $\sum_{n\geq 0} L_n(x) t^n$ | $(1-t)^{-\alpha}$ | $\exp(z)$ | $-x$ | $t(1-t)^{-1}$ |
| Hermite $H_n$ | $\sum_{n\geq 0} \frac{1}{n!} H_n(x) t^n$ | $\exp(-t^2)$ | $\exp(z)$ | $2x$ | $t$ |
| Jacobi $P_{n(\alpha,\beta)}$ | $\sum_{n\geq 0} \frac{1}{(\alpha+1)n!(\beta+1)n!} P_{n(\alpha,\beta)}(x) t^n$ | $(1+t)\alpha-\beta-1$ | $2F_1\left(\frac{\alpha+\beta+1}{2}, \frac{\alpha+\beta+2}{2}; \beta+1; z\right)$ | $1+x$ | $2(1+t)^{-2}$ |
| Fibonacci $F_n$ | $\sum_{n\geq 0} F_n(x) t^n$ | $(1-t)^{-1}$ | $(1-z)^{-1}$ | $x$ | $t(1-t)^{-1}$ |
| Euler $E_n$ | $\sum_{n\geq 0} \frac{1}{n!} E_n(x) t^n$ | $2^n(e^t-1)^{-\alpha}$ | $\exp(z)$ | $x$ | $t$ |
| Bernoulli $B_n$ | $\sum_{n\geq 0} \frac{1}{n!} B_n(x) t^n$ | $t^n(e^t-1)^{-\alpha}$ | $\exp(z)$ | $x$ | $t(1-t)^{-2}$ |
| Mott $M_n$ | $\sum_{n\geq 0} \frac{1}{n!} M_n(x) t^n$ | $1$ | $z(1+2z)^{-1}$ | $x$ | $t(1-t)^{-2}$ |
| Spread $S_n$ | $\sum_{n\geq 0} S_n(x) t^n$ | $1$ | $\exp(z)$ | $-x$ | $1-\sqrt{1-4t}/t$ |
| Bessel $p_n$ | $\sum_{n\geq 0} \frac{1}{n!} p_n(x) t^n$ | $1$ | $\exp(z)$ | $x$ | $\log(1+t)$ |

Table 4: Polynomials with conversion in $O(M(n)\log(n))$

| polynomial | generating series | $u(x)v(t)$ | $f(z)$ | $g(x)$ | $h(t)$ |
|------------|-------------------|-------------|--------|--------|--------|
| Falling factorial $(x)_n$ | $(*)$ | $\sum_{n\geq 0} (x)_n t^n$ | $1$ | $\exp(z)$ | $x$ |
| Bell $\phi_n$ | $\sum_{n\geq 0} \frac{1}{n!} \phi_n(x) t^n$ | $t\log(1+t)$ | $\exp(z)$ | $x$ | $\log(1+t)$ |
| Bernoulli, 2nd kind $b_n$ | $\sum_{n\geq 0} \frac{1}{n!} b_n(x) t^n$ | $\exp(-t)$ | $\exp(z)$ | $x$ | $\log(1+t/n)$ |
| Poisson-Charlier $c_{n}(x; a)$ | $\sum_{n\geq 0} \frac{1}{n!} c_{n}(x; a) t^n$ | $\exp(bt)$ | $\exp(z)$ | $-x$ | $\exp(t) - 1$ |
| Actuarial $a_{n(\beta)}$ | $\sum_{n\geq 0} \frac{1}{n!} a_{n(\beta)}(x) t^n$ | $\exp(\beta t)$ | $\exp(z)$ | $-x$ | $\exp(t) - 1$ |
| Narumi $N_{n(\beta)}$ | $\sum_{n\geq 0} \frac{1}{n!} N_{n(\beta)}(x) t^n$ | $t^n\log(1+t)^{-\alpha}$ | $\exp(z)$ | $x$ | $\log(1+t)$ |
| Peters $P_n^{(\lambda,\mu)}$ | $\sum_{n\geq 0} \frac{1}{n!} P_n^{(\lambda,\mu)}(x) t^n$ | $(1+(1+t)^{\lambda})^{-\mu}$ | $\exp(z)$ | $x$ | $\log(1+t)$ |
| Meixner-Pollaczek $P_{n(\phi)}(x; \phi)$ | $(**)$ | $\sum_{n\geq 0} \frac{1}{n!} P_{n(\phi)}(x; \phi) t^n$ | $1+(1+2t\cos\phi)^{-\alpha}$ | $\exp(z)$ | $ix \log(1-\cos\phi)^{-\alpha}$ |
| Meixner $m_n(x; \beta, c)$ | $(**)$ | $\sum_{n\geq 0} \frac{1}{n!} m_n(x; \beta, c) t^n$ | $(1-t)^{-\beta}$ | $\exp(z)$ | $x$ |
| Krawtchouk $K_n(x; p, N)$ | $(**)$ | $\sum_{n\geq 0} \frac{1}{n!} K_n(x; p, N) t^n$ | $(1+t)^N$ | $\exp(z)$ | $\log(1+t)^N(1+t)^{N-1}$ |

7. REFERENCES

[1] A. V. Aho, K. Steiglitz, and J. D. Ullman. Evaluating polynomials at fixed sets of points. SIAM J. Comp., 4(4):533–539, 1975.
[2] B. K. Alpert and V. Rokhlin. A fast algorithm for the evaluation of Legendre expansions. SIAM J. Sci. Statist. Comp., 12(1):158–179, 1991.
[3] G. Andrews, R. Askey, and R. Roy. Special functions. Cambridge University Press, 1999.
[4] R. Barrio and J. Peña. Basis conversions among univariate polynomial representations. C. R. Math. Acad. Sci. Paris, 339(4):293–298, 2004.
[5] D. J. Bernstein. Composing power series over a finite ring in polynomial representations. Comp. Aided Geom. Design, 15(6):637–642, 1998.
[6] D. Potts, G. Steidl, and M. Tasse. Fast algorithms for discrete orthogonal polynomials. In SNC’07, pages 117–124, New York, NY, USA, 2007. ACM.
[7] Y.-M. Li and X.-Y. Zhang. Basis conversion among Bézier, Tchebyshev and Legendre. Comput. Aided Geom. Design, 15(6):637–642, 1998.
[8] V. Y. Pan. New fast algorithms for polynomial interpolation and evaluation on the Chebyshev node set. Computers and Mathematics with Applications, 35(3):125–129, 1998.
[9] M. Frumkin. A fast algorithm for expansion over spherical harmonics. Appl. Algebra Engrg. Comm. Comp., 6(6):333–343, 1995.
[10] J. g. Gathen and J. Gerhard. Modern computer algebra. Cambridge University Press, 1999.
[11] J. Gerhard. Modular algorithms for polynomial basis conversion and greatest factorial factorization. In RWCA’00, pages 125–141, 2000.
[12] G. Hanrot, M. Quecira, and P. Zimmermann. The Middle Product Algorithm. I. Appl. Algebra Engrg. Comm. Comp., 14(6):415–438, 2004.
[13] G. Hanrot and P. Zimmermann. Newton iteration revisited. http://www.loria.fr/~zimmerma/papers, 2002.
[14] T. Pelin, T. Karpus, and M. Tasse. Challenges of symbolic computation: my favorite open problems. J. Symb. Comp., 29(6):891–919, 2000.
[15] M. Kaltofen and V. Lakshman. Improved sparse multivariate polynomial interpolation algorithms. In ISSAC’88, volume 358 of LNCS, pages 467–474. Springer Verlag, 1989.
[16] J. Keiner. Computing with expansions in Gegenbauer polynomials. Preprint AMR07/10, U. New South Wales, 2007.
[17] G. Leibon, D. Rockmore, and G. Chirikjian. A fast Hermite transform with applications to protein structure determination. In SNC’07, pages 117–124, New York, NY, USA, 2007. ACM.
[18] J. g. Hoeven. Relax, but don’t be too lazy. J. Symb. Comput., 34(6):479–542, 2002.
[19] J. J. Hoeven. Relax, but don’t be too lazy. J. Symb. Comput., 34(6):479–542, 2002.
[20] E. Kaltofen. Challenges of symbolic computation: my favorite open problems. J. Symb. Comp., 29(6):891–919, 2000.
[21] E. Kaltofen and Y. Lakshman. Improved sparse multivariate polynomial interpolation algorithms. In ISSAC’88, volume 358 of LNCS, pages 467–474. Springer Verlag, 1989.
[22] D. Potts, G. Steidl, and M. Tasse. Fast algorithms for discrete polynomial transforms. Math. Comp., 67(244):1577–1590, 1998.
[23] S. Roman. The umbral calculus. Dover publications, 2005.
[24] A. Schönhage and V. Strassen. Schnelle Multiplikation großer Zahlen. Computing, 7:281–292, 1971.
[25] V. Shoup. A new polynomial factorization algorithm and its implementation. J. Symb. Comp., 20(4):363–397, 1995.