Chains of Recurrences - a method to expedite the evaluation of closed-form functions *

Olaf Bachmann
Dept. of Math. and Computer Science
Kent State University
Kent, Ohio 44242-0001 USA
obachman@mcs.kent.edu

Paul S. Wang†
Distributed Computing Department
Sandia National Laboratories
P.O.Box 969 Mail Stop 9214
Livermore, CA 94551-0969
pwang@mcs.kent.edu

Eugene V. Zima
Dept. of Computational Mathematics and Cybernetics (BMK)
Moscow State University
Moscow, 119899, Russia
zima@cs.msu.su

Abstract

Chains of Recurrences (CR's) are introduced as an effective method to evaluate functions at regular intervals. Algebraic properties of CR's are examined and an algorithm that constructs a CR for a given function is explained. Finally, an implementation of the method in MAXIMA/Common Lisp is discussed.

1 Introduction

Given a closed-form function $G(x)$, a common computational task is to evaluate the function at a number of points in an interval. More precisely, given a starting point x_0 and an increment h, the task is to compute $G(x_0 + ih)$ for $i = 0, 1, \ldots, n-1$. Such computations occur frequently in practice: plotting curves of functions, computing finite sums and products, calculating integrals, and solving differential equations. Straightforward evaluations of G at all n points can be very inefficient and may sometimes even become the bottleneck of a given system. The SIG [5] graphing system is such an example.

One way to speed up this type of evaluation is to compute the next point of G by using results from earlier steps, i.e. by utilizing recurrence properties of G. Take $G(x) = 3x + 1$ for example. Instead of computing $3x_0+1$, $3(x_0+h)+1$, and so on, we may compute $G(x_0) = 3x_0 + 1$, $c = 3h$ and then generate the desired evaluations by adding c to the previous value of G. The latter approach is obviously faster. Of course, it is common programming practice to use simple recurrences like this or like $x^{hi} = x^{h(i-1)}x^h$ to economize certain computations. More complicated is a technique known by numerical analysists as "forward differencing" [4][1] which reduces the process of evaluating an nth degree polynomial to just n additions after the first few steps. In this method a system of $n+1$ recursive functions (also called a finite difference table) is set up such that its successive evaluation yields the desired values of the polynomial in question. It is also known that many other useful functions satisfy specific recurrence relations. For example, in trigonometry we have

$$\begin{aligned}\cos(i\theta) &= 2\cos(\theta)\cos([i-1]\theta) - \cos([i-2]\theta)\\ \sin(i\theta) &= 2\sin(\theta)\sin([i-1]\theta) - \sin([i-2]\theta).\end{aligned}$$

Instead of finding recurrences for a particular class of functions, our goal is to automatically generate recurrence representations for a wide variety of common functions in order to obtain more efficient computational procedures for their evaluation. More precisely, for a given function $G(x)$, we construct an appropriate Chain of Recurrences (CR) Φ such that $\Phi(i) = G(x_0 + ih)$ and the evaluation of Φ is potentially much more efficient than the straightforward evaluation of G. Techniques described here can, for example, generate a CR for the function G_1 (Fig. 1) that reduces the evaluation time (for 1000 points) from 1614 ms to 155 ms.

The Chains of Recurrences investigation here extends Zima's studies of Systems of Recurrence Relations [6]:

- New definitions and concepts (sections 2 and 3) for describing CR's are developed.
- Algebraic properties of CR's are systematically examined (section 3).
- An algorithm for constructing CR's is given and analyzed (section 4).
- Improvements of the original algorithm are described (section 4).
- A general implementation of the CR method is described in detail and actual machine timings are presented (section 5).
- The CR method as implemented is applied to speed up a graphing package (section 5).

*Work reported herein has been supported in part by the National Science Foundation under Grant CCR-9201800, in part by the Army Research Office under Grant DAAL03-91-G-0149 and in part by the RFFI (Russia) under Grant 94-01-01743.

†On sabbatical leave from Kent State University

ISAAC 94 - 7/94 Oxford England UK
© 1994 ACM 0-89791-638-7/94/0007..$3.50

[1] Knuth describes this phenomenon as "tabulating polynomial values" [3, p.469]

j	$G_j(x)$	CR-expression Φ_j such that $\Phi_j(i) = G_j(x_0 + i * h)$
1	$\frac{e^{x^3+3x^2-3x+1}}{2^{x^2-2x+1}}$	$\left\{\frac{e^{x_0^3+3x_0^2-3x_0+1}}{2^{x_0^2-2x_0+1}}, *, \frac{e^{3hx_0^2+3x_0(h^2+2h)+h^3+3h^2-3h}}{2^{2hx_0-2h+h^2}}, *, \frac{e^{6(h^2x_0+h^3+h^2)}}{2^{2h^2}}, *, e^{6h^3}\right\}$
2	$a_0 + a_1x + \ldots + a_nx^n$	$\{c_0(x_0,h), +, c_1(x_0,h), +, \ldots, +, c_n(x_0,h)\}$ where the c_i's are certain constructible polynomials in x_0, h
3	$\cos(20x) * e^{x^2}$	$\cos(\{20x_0, +, 20h\}) * \{e^{x_0^2}, *, e^{2hx_0+h^2}, *, e^{2h^2}\}$
4	$\frac{(x!)^2}{(n-x)!}$ $x, n \in \mathbf{N}$	$\{\frac{1}{n!}, *, n, +, 3n-4, +, 2n-10, +, -6\}$ $x_0 = 0, h = 1$
5	$\frac{(x!)^2\, 2^{x^2-1}}{e^{2x^3+4x+2}\,(n-x)!}$ $x, n \in \mathbf{N}$	$\left\{\frac{1}{2e^2n!}, *, \{\frac{2}{e^6}, *, \frac{4}{e^{12}}, *, \frac{1}{e^{12}}\} * \{n, +, 3n-4, +, 2n-10, +, -6\}\right\}$ $x_0 = 0, h = 1$

Figure 1: Examples of Chains of Recurences

2 Chains of Recurrences

First let us introduce basic concepts for representing functions using recurrences.

Given a constant φ_0, a function f_1 defined over the natural numbers $\mathbf{N}$, and an operator $\odot$ equal to either + or *, a *Basic Recurrence (BR)* f, represented by the tuple

$$f = \{\varphi_0, \odot, f_1\},$$

is defined as a function $f(i)$ over $\mathbf{N}$ by

$$\{\varphi_0, +, f_1\}(i) = \varphi_0 + \sum_{j=0}^{i-1} f_1(j)$$

$$\{\varphi_0, *, f_1\}(i) = \varphi_0 \prod_{j=0}^{i-1} f_1(j).$$

For example, consider the BR $f = \{3x_0+1, +, 3h\}$. Then $f(i) = \{3x_0+1, +, 3h\}(i) = G(x_0+ih)$ where $G(x) = 3x+1$.

Notice that for any BR $f = \{\varphi_0, \odot, f_1\}$ and $i > 0$, $f(i) = f(i-1) \odot f_1(i-1)$. Hence, BR's are special cases of first-order linear recurrences. Obviously, only simple functions can be expressed with BR's. In order to represent more complicated functions using recurrences, we introduce the concept of a *Chain of Recurrences (CR)*.

Given constants $\varphi_0, \ldots, \varphi_{k-1}$, a function f_k defined over $\mathbf{N}$, and operators $\odot_1, \ldots, \odot_k$ equal to either + or *, a *Chain of Recurrences (CR)* Φ, represented by the tuple

$$\Phi = \{\varphi_0, \odot_1, \varphi_1, \odot_2, \varphi_2, \ldots, \odot_k, f_k\},$$

is defined recursively as a function over $\mathbf{N}$ by

$$\Phi(i) = \{\varphi_0, \odot_1, \{\varphi_1, \odot_2, \varphi_2, \ldots, \odot_k, f_k\}\}(i) \quad (1)$$

For a given $G(x)$, we are interested in constructing a CR Φ such that $\Phi(i) = G(x_0 + i * h)$. For example, for $G(x) = e^{x^2}$, we obtain the CR

$$\Phi = \{e^{x_0^2}, *, e^{2x_0h+h^2}, *, e^{2h^2}\}$$

such that $\Phi(i) = G(x_0 + ih)$.

For increased readability we will consistently denote BR's by lowercase letters (like f and g), CR's by uppercase Greek letters (like Φ and Ψ), and constants appearing in BR's or CR's by indexed lowercase Greek letters (like φ_0 and ψ_1). The shorthand notations $\Phi = \Phi(i)$ and $\varphi_i = \varphi_i(x_0, h)$ will also be used.

Given a CR $\Phi = \{\varphi_0, \odot_1, \varphi_1, \odot_2, \varphi_2, \ldots, \odot_k, f_k\}$ it is easy to verify that there exists BR's $f_0, f_1, \ldots, f_{k-1}$ such that for $0 \le j < k$

$$f_j(i) = \begin{cases} \varphi_j, & \text{if } i = 0 \\ f_j(i-1) \odot_{j+1} f_{j+1}(i-1), & \text{if } i > 0 \end{cases} \quad (2)$$

and $f_0(i) = \Phi(i)$. A similar scheme was originally used by Zima to define a *System of Recurrence Relations* (SRR). The new definition (1), basically equivalent but less cumbersome, involves a "chain" of recurrences, hence the name CR.

Given a CR $\Phi = \{\varphi_0, \odot_1, \varphi_1, \odot_2, \ldots \odot_k, f_k\}$ we call

- $k = L(\Phi)$ the *length* of Φ
- Φ a *pure-sum* CR if $\odot_1 = \odot_2 = \ldots = \odot_k = +$
- Φ a *pure-product* CR if $\odot_1 = \odot_2 = \ldots = \odot_k = *$
- Φ a *simple* CR if f_k is a constant.

Before proceeding further, let us look at figure 1 which shows some more sample CR's.

Observe that the pure-product CR Φ_1 is simple and of length 3. Φ_2 indicates that we can obtain a simple, pure-sum CR of length n for any given nth degree polynomial. The CR-expression in row 3 involves two simple CR's: a pure-sum CR of length 1 and a pure-product CR of length 2. Restricting x to be a natural number and setting $x_0 = 0$, $h = 1$ we obtain Φ_4 and Φ_5. Notice that the CR Φ_5 is of length 1 and that its function f_1 is the product of two other CR's.

2.1 Evaluation of CR's

Once a CR Φ is constructed for a function, it can provide a very efficient way to evaluate the function at regular intervals. To generate the function values, Φ is first initialized by evaluating all constant expressions in the tuple notation of the CR with given values for x_0, h, and other parameters. The initialization produces values for $\varphi_0, \ldots, \varphi_{k-1}, f_k$, and $G(x_0) = \Phi(0)$. Values of G at successive points ($\Phi(1), \Phi(2)$, ...) are then generated in a loop. Each iterative step updates the values in the tuple notation of Φ appropriately and thereby computes the next $\Phi(i)$.

It is easy to see that the number of arithmetic operations needed to compute the next value of a simple CR is equal to the length of the CR. Therefore, the length of a CR gives an indication of its evaluation cost. Thus, using Φ_1 (Fig. 1), only three multiplications are needed to obtain the value of G_1 at each successive evaluation point.

3 Algebraic Properties of BR's and CR's

We have seen examples of CR's and their effectiveness for evaluating functions at regular intervals. The question is how to construct an appropriate CR for any given $G(x)$.

As a first approach we will examine the reverse question: Given a CR of a certain kind, what is its value as a closed-form formula?

Recall that for $n \in \mathbf{N}$, *factorials* of the form $x^{(n)}$ (which are also known as falling factorials) are defined by

$$x^{(n)} = x(x-1)(x-2)\ldots(x-n+1) \quad \text{with } x^{(0)} = 1$$

and that polynomials in x represented by

$$a_0 + a_1 x^{(1)} + a_2 x^{(2)} + \ldots a_n x^{(n)}$$

are called *factorial polynomials*. Factorials such as $x^{(n)}$ can be reduced to polynomials in x and vice versa using

$$x^{(n)} = s_{n1}x + s_{n2}x^2 + \ldots + s_{nn}x^n$$

$$x^n = t_{n1}x^{(1)} + t_{n2}x^{(2)} + \ldots + t_{nn}x^{(n)}$$

where the numbers s_{ij} (t_{ij}) are the Stirling numbers of the first (second) kind [1].

Using these concepts, we can easily prove the following lemma by induction on k.

Lemma 1 *Let* $\{\varphi_0, \odot, \varphi_1, \odot, \ldots, \odot, \varphi_k\}$ *be a simple CR. Then,*

$$\begin{aligned}
&\{\varphi_0, +, \varphi_1, +, \varphi_2, +, \ldots, +, \varphi_k\}(i) \\
&= \varphi_0 + \varphi_1 i^{(1)} + \frac{\varphi_2}{2!} i^{(2)} + \ldots + \varphi_k \frac{i^{(k)}}{k!} \qquad (3) \\
&= \varphi_0 + i\sum_{j=1}^{k} \frac{\varphi_j}{j!} s_{j1} + i^2 \sum_{j=2}^{k} \frac{\varphi_j}{j!} s_{j2} + \ldots + i^k \frac{\varphi_k}{k!} s_{kk}
\end{aligned}$$

$$\begin{aligned}
&\{\varphi_0, *, \varphi_1, *, \varphi_2, *, \ldots, *, \varphi_k\}(i) \\
&= \varphi_0 * \varphi_1^{i^{(1)}} * \varphi_2^{\frac{i^{(2)}}{2!}} * \ldots * \varphi_k^{\frac{i^{(k)}}{k!}} \qquad (4) \\
&= \exp\left(\log(\varphi_0) + \log(\varphi_1) i^{(1)} + \frac{\log(\varphi_2)}{2!} i^{(2)} + \ldots + \frac{\log(\varphi_k)}{k!} i^{(k)}\right)
\end{aligned}$$

In other words, a simple, pure-sum CR of length k is a polynomial in i of degree k. And, a simple, pure-product CR is e raised to a polynomial exponent. For example,

$$\begin{aligned}
1 &= \{1\} \\
x &= \{x_0, +, h\} \\
x^2 &= \{x_0^2, +, 2hx_0 + h^2, +, 2h^2\} \\
x^3 &= \{x_0^3, +, 3hx_0^2 + 3h^2 x_0 + h^3, +, 6h^2 x_0 + 6h^3, +, 6h^3\} \\
x^n &= \{c_{0n}, +, c_{1n}, +, c_{2n}, +, \ldots, +, c_{nn}\} \qquad (5) \\
e^{x^n} &= \{e^{c_{0n}}, *, e^{c_{1n}}, *, e^{c_{2n}}, *, \ldots, *, e^{c_{nn}}\}
\end{aligned}$$

where[2] $c_{in}(x_0, h) = i! \sum_{j=i}^{n} \binom{n}{j} h^j x_0^{n-j} t_{ji}$

Given a polynomial $P(x)$ (or $e^{P(x)}$), it is therefore conceivable to devise an algorithm which, based on lemma 1, constructs an equivalent CR. However, this "head-on" construction approach is not practical because it is expensive and of limited applicability. A recursive CR construction technique, superior in many ways, will be presented below.

Our general strategy to construct a CR for a given formula $G(x)$ is the following:

[2] assuming that $t_{00} = 1$

1. On the parse tree of G, first replace the trivial subexpression x by the basic recurrence $\{x_0, +, h\}$.
2. Recursively apply algebraic properties of recurrences to combine CR's. This process proceeds from the bottom to the top of the parse tree of G and simplifies expressions involving CR's.

The following sections examine important algebraic properties of BR's and CR's. These properties allow us to devise a general CR construction algorithm (section 4).

3.1 Algebraic Properties of BR's

BR's lay at the heart of our construction method since a CR is a chain of BR's. Therefore, we first examine algebraic properties of BR's before applying them to CR's in the following subsection.

To begin, arithmetic operations involving a single BR are considered:

Lemma 2 *Let* $f = \{\varphi_0, \odot, f_1\}$ *be a BR and* c *be a constant. Then,*

$$\begin{aligned}
c + \{\varphi_0, +, f_1\} &= \{c + \varphi_0, +, f_1\} && (6) \\
c * \{\varphi_0, +, f_1\} &= \{c * \varphi_0, +, c * f_1\} && (7) \\
c^{\{\varphi_0, +, f_1\}} &= \{c^{\varphi_0}, *, c^{f_1}\} && (8) \\
c * \{\varphi_0, *, f_1\} &= \{c * \varphi_0, *, f_1\} && (9) \\
\{\varphi_0, *, f_1\}^c &= \{\varphi_0^c, *, f_1^c\} && (10) \\
\log(\{\varphi_0, *, f_1\}) &= \{\log(\varphi_0), +, \log(f_1)\}. && (11)
\end{aligned}$$

The following lemma examines algebraic properties of arithmetic operations of two BR's:

Lemma 3 *Let* $f = \{\varphi_0, \odot, f_1\}$ *and* $g = \{\psi_0, \otimes, g_1\}$ *be BR's. Then,*

$$\begin{aligned}
&\{\varphi_0, +, f_1\} + \{\psi_0, +, g_1\} = \{\varphi_0 + \psi_0, +, f_1 + g_1\} && (12) \\
&\{\varphi_0, +, f_1\} * \{\psi_0, +, g_1\} = \{\varphi_0\psi_0, +, fg_1 + gf_1 + f_1 g_1\} && (13) \\
&\{\varphi_0, *, f_1\} * \{\psi_0, *, g_1\} = \{\varphi_0\psi_0, *, f_1 g_1\} && (14) \\
&\{\varphi_0, *, f_1\}^{\{\psi_0, +, g_1\}} = \{\varphi_0^{\psi_0}, *, f^{g_1} * f_1^g * f_1^{g_1}\}. && (15)
\end{aligned}$$

Proof: The equalities (6) - (15) are proved using the definition of BR's and basic arithmetic identities. As an example, the proof of (13) is shown here:

$$\begin{aligned}
&\{\varphi_0, +, f_1\} * \{\psi_0, +, g_1\} = [\varphi_0 + \sum_{j=0}^{i-1} f_1(j)] * [\psi_0 + \sum_{j=0}^{i-1} g_1(j)] \\
&= \varphi_0\psi_0 + \sum_{j=0}^{i-1} \Big[g_1(j)[\varphi_0 + \sum_{l=0}^{j-1} f_1(l)] + f_1(j)[\psi_0 + \sum_{l=0}^{j-1} g_1(l)] \\
&\qquad + f_1(j) g_1(j) \Big] \\
&= \varphi_0\psi_0 + \sum_{j=0}^{i-1} \Big[g_1(j) \{\varphi_0, +, f_1\}(j) + f_1(j) \{\psi_0, +, g_1\}(j) \\
&\qquad + f_1(j) g_1(j) \Big] \\
&= \{\varphi_0\psi_0, +, g_1 f + f_1 g + f_1 g_1\} \qquad \#
\end{aligned}$$

Let us illustrate how these properties can be used to construct CR's. Consider $G(x) = e^{x^2}$, for example. First, we replace x^2 by $\{x_0, +, h\} * \{x_0, +, h\}$ and perform the following transformations:

$$\begin{aligned}
&\{x_0, +, h\} * \{x_0, +, h\} \\
&= \{x_0^2, +, h * \{x_0, +, h\} + h * \{x_0, +, h\} + h^2\} \text{ by (13)} \\
&= \{x_0^2, +, \{2hx_0 + h^2, +, 2h^2\}\} \text{ by (7) and (12)} \\
&= \{x_0^2, +, 2hx_0 + h^2, +, 2h^2\} \text{ by (1)}
\end{aligned}$$

By substituting the latter CR for x^2 in e^{x^2} we continue:

$$
\begin{aligned}
& e^{\{x_0^2, +, 2hx_0+h^2, +, 2h^2\}} \\
&= e^{\{x_0^2, +, \{2hx_0+h^2, +, 2h^2\}\}} \text{ by (1)} \\
&= \{e^{x_0^2}, *, \{e^{2hx_0+h^2}, *, e^{2h^2}\}\} \text{ by (8)} \\
&= \{e^{x_0^2}, *, e^{2hx_0+h^2}, *, e^{2h^2}\} \text{ by (1)}
\end{aligned}
$$

It should be noted that similar transformation rules can not be found for BR-expressions like $c + \{\varphi_0, *, f_1\}$ or $\{\varphi_0, +, f_1\} * \{\psi_0, *, g_1\}$. An analysis of the closed formulas of such BR-expressions shows that they can not be transformed into useful recursive representations independent of i. In other words, it is due to arithmetic properties of such expressions that they can not be represented by a single BR and therefore, no useful transformation rules can be given for such BR expressions.

3.2 Algebraic Properties of CR's

Before applying BR properties to CR's, it is useful to define the concept of a CR-expression more precisely by generalizing the definition of a CR in the natural way.

We call an expression Φ to be a *CR-expression* if it represents one of the following functions over $\mathbf{N}$:

1. a constant
2. a CR $\{\varphi_0, \odot_1, \varphi_1, \odot_2, \ldots, \odot_k, f_k\}$ where f_k is a CR-expression
3. $F(\Phi_1, \Phi_2, \ldots, \Phi_m)$, where F is a function of m arguments and $\Phi_1, \Phi_2, \ldots, \Phi_m$ are CR-expressions.

Generalizing from the length of a CR, we define the *Cost Index (CI)* of a CR-expression Φ to be

$$
CI(\Phi) = \begin{cases} 0, & \text{if } \Phi \text{ is a constant} \\ k + CI(f_k), & \text{if } \Phi \equiv \{\varphi_0, \odot_1, \varphi_1, \ldots, \odot_k, f_k\} \\ 1 + \sum_{j=1}^{m} CI(\Phi_j), & \text{if } \Phi \equiv F(\Phi_1, \Phi_2, \ldots, \Phi_m) \end{cases}
$$

Similar to the length of a CR, the cost index of a CR-expression gives an indication of its evaluation cost (it counts the number of operations needed to evaluate a CR-expression at one point)[3].

Using these concepts, we return to CR's by considering simplification properties of CR-expressions. First, properties involving a single CR are considered.

Proposition 1 *Let $\Phi = \{\varphi_0, \odot_1, \varphi_1, \odot_2, \ldots, \odot_k, f_k\}$ be a CR and c be a constant. Then the following relations hold:*

$$\odot_1 \equiv + \;\Rightarrow\; c + \Phi = \{c+\varphi_0, +, \varphi_1, \odot_2, \ldots, \odot_k, f_k\} \quad (16)$$

$$\odot_1 \equiv * \;\Rightarrow\; c * \Phi = \{c*\varphi_0, *, \varphi_1, \odot_2, \ldots, \odot_k, f_k\} \quad (17)$$

Φ is pure-sum

$$\Rightarrow\; c * \Phi = \{c*\varphi_0, +, c*\varphi_1, +, \ldots, +, c*f_k\} \quad (18)$$

$$\text{and}\quad c^{\Phi} = \{c^{\varphi_0}, *, c^{\varphi_1}, *, \ldots, *, c^{f_k}\} \quad (19)$$

Φ is pure-product

$$\Rightarrow\; \Phi^c = \{\varphi_0^c, *, \varphi_1^c, *, \ldots, *, f_k^c\} \quad (20)$$

$$\text{and}\quad \log(\Phi) = \{\log(\varphi_0), +, \log(\varphi_1), +, \ldots, +, \log(f_k)\}. \quad (21)$$

These relations are proved by induction on the length of Φ using properties of BR's. If Φ is a CR-expression, we have $CI(LHS) > CI(RHS)$ for (16) and (17). Also $CI(LHS) \geq CI(RHS)$ for (18)–(21) (In fact, if Φ is simple than $CI(LHS) > CI(RHS)$ for (18)–(21), as well). Hence, the right-hand sides of Eq. (16)–(21) are usually less costly to evaluate.

Next, we apply the BR properties (12) and (14) to CR's.

Proposition 2 *Let $\Phi = \{\varphi_0, \odot, \varphi_1, \odot, \ldots, \odot, \varphi_k\}$ and $\Psi = \{\psi_0, \odot, \psi_1, \odot, \ldots, \odot, \psi_l\}$ be two simple CR's where $k \geq l$. For pure-sum Φ and Ψ we have*

$$\Phi + \Psi = \{\varphi_0+\psi_0, +, \ldots, \varphi_l+\psi_l, +, \varphi_{l+1}, \ldots, +, \varphi_k\}. \quad (22)$$

And for pure-product Φ and Ψ, we have

$$\Phi * \Psi = \{\varphi_0*\psi_0, *, \ldots, \varphi_l*\psi_l, *, \varphi_{l+1}, \ldots, *, \varphi_k\}. \quad (23)$$

These equalities are shown by induction on $CI(\Phi \odot \Psi)$ using lemma 2 and 3. Note also that $CI(LHS) = CI(RHS) + l + 1$. Hence, applying (22) or (23) to a CR-expression reduces its cost index by $l + 1$.

It takes some more detailed analysis to apply the BR properties (13) and (15) to CR's. Note that, if considered as CR-expressions, $CI(LHS) < CI(RHS)$ for (13) and (15). In other words, the right-hand sides of these equalities are more complex arithmetic expressions than the left-hand sides. The following proposition examines under which conditions and how an application of (13) and (15) leads to useful simplifications.

Proposition 3 *Let Φ, Ψ be simple CR's of length k and l.*

(i) If Φ and Ψ are pure-sum, then there exists a simple pure-sum CR Σ of length $k + l$ such that $\Phi * \Psi = \Sigma$. Σ can be constructed using the algorithm **CRProd** *given below, i.e. $\Sigma =$* **CRProd**(Φ, Ψ).

(ii) If Φ is pure-product and Ψ is pure-sum, then there exists a simple, pure-product CR Π of length $k + l$ such that $\Phi^{\Psi} = \Pi$. Π can be constructed using the algorithm **CRExpt** *discussed below, i.e. $\Pi =$* **CRExpt**(Φ, Ψ).

By lemma 1, we know immediately that Σ and Π exist. The important question is how to construct Σ and Π efficiently and whether they speed up the computations.

Let us first consider the algorithm **CRProd**:

Algorithm CRProd: Let $\Phi = \{\varphi_0, +, \varphi_1, +, \ldots, +, \varphi_k\}$ and $\Psi = \{\psi_0, +, \psi_1, +, \ldots, +, \psi_l\}$ with $k \geq l$, this algorithm returns a a simple, pure-sum CR Σ of length $k + l$ such that $\Phi * \Psi = \Sigma$.

P1 [Base case]
If $l = 0$ return $\{\varphi_0\psi_0, +, \varphi_1\psi_0, +, \ldots, +, \varphi_k\psi_0\}$

P2 [Prepare recursive calls] Let
$f_1 = \{\varphi_1, +, \varphi_2, +, \ldots, +, \varphi_k\}$
$g_1 = \{\psi_1, +, \psi_2, +, \ldots, +, , \psi_l\}$
$\Psi' = \Psi + g_1 = \{\psi_0 + \psi_1, +, \psi_1 + \psi_2, +, \ldots, +, \psi_l\}$

P3 [Recursive Calls based on (13)] Set
$\{\xi_1', +, \xi_2', +, \ldots, +, \xi_{k+l}'\} \leftarrow$**CRProd**$(\Phi, g_1)$
$\{\xi_1'', +, \xi_2'', +, \ldots, +, \xi_{k+l}''\} \leftarrow$**CRProd**$(f_1, \Psi')$

P4 [Simplify and return result using (22)]
return $\{\phi_0\psi_0, +, \xi_1' + \xi_1'', +, \ldots, +, \xi_{k+l}' + \xi_{k+l}''\}$

The algorithm **CRExpt** can be easily obtained from **CRProd** by appropriately replacing multiplications (additions) with exponentiations (multiplications).

Finally, to prove proposition 3 we must verify the correctness of the algorithms **CRProd** and **CRExpt**. This is easily done by induction on $CI(\Phi * \Psi)$, using (13), (18), (22) and noticing that in step P3, $CI(\Phi * g_1) = CI(f_1 * \Psi') = CI(\Phi * \Psi) - 1$. Also, Σ and Π are less costly to evaluate because $CI(\Phi * \Psi) > CI(\Sigma)$ and $CI(\Phi^{\Psi}) > CI(\Pi)$.

[3] For simplicity, let us count F as one operation and assume that if $F = +$ or $F = *$ then $m = 2$

We conclude this section with the following interesting property about the factorial function:

Proposition 4 *Let* $\varphi_1 \in \mathbf{Z}$. *Then*

$\varphi_1 > 0$

$$\Rightarrow \{\varphi_0, +, \varphi_1\}! = \{\varphi_0!, *, \xi_0, +, \xi_1, +, \ldots, +, \xi_{\varphi_1}\} \quad (24)$$

$\varphi_1 < 0$

$$\Rightarrow \{\varphi_0, +, \varphi_1\}! = \{\varphi_0!, *, \frac{1}{\{\xi_0, +, \xi_1, +, \ldots, +, \xi_{|\varphi_1|}\}}\} \quad (25)$$

for some constructible constants $\xi_0, \xi_1, \ldots, \xi_{|\varphi_1|}$.

This claim is proved using proposition 3 and the fact that

$$\{\varphi_0, +, |\varphi_1|\}! = (\varphi_0 + i\,|\varphi_1|)! = \varphi_0! * \prod_{j=0}^{i-1} \prod_{l=1}^{|\varphi_1|} (\varphi_0 + j|\varphi_1| + l)$$

$$= \{\varphi_0!, *, \prod_{l=1}^{|\varphi_1|} \{\varphi_0 + l, +, |\varphi_1|\}\}$$

Using (24) we obtain, for example, the following CR's:

$$i! = \{1, *, 1, +, 1\} \qquad i!^2 = \{1, *, 1, +, 3, +, 2\}$$

$$(n+i)! = \{n!, *, n+1, +, 1\} \qquad (n-i)! = \{n!, *, \frac{1}{\{n, +, -1\}}\}$$

$$\frac{1}{(n+i)!} = \{\frac{1}{n!}, *, \frac{1}{\{n+1, +, 1\}}\} \qquad \frac{1}{(n-i)!} = \{\frac{1}{n!}, *, n, +, -1\}$$

4 The CR construction algorithm

Summarizing the results from the last section, we devise the algorithm **CRMake** to construct a CR-expression Φ for a given function $G(x)$.
Some remarks about the algorithm:

- The correctness of **CRMake** follows immediately from the results of section 3. Also, no further simplifications based on the lemmas and propositions of section 3 can be performed on the returned CR-expression Φ, i.e. $CI(\Phi)$ is minimized with respect to the results of Section 3.
- The only requirement on the input formula G is that it is a closed-form function. No special preknowledge about the characteristics or representation of G is needed. The algorithm automatically recognizes structures which can be represented by a CR, independently of their syntactical representation.
- The domain of the input function G may remain unspecified. In fact, the algorithm returns correct results as long as G is defined over a commutative ring. Therefore, the algorithm can be used, for example, to construct CR's for natural, real or complex functions G.
- The algorithm returns symbolic CR's which can be initialized and reused for different values of x_0 and h. However, if we use actual values for x_0 and h during the CR construction, then the algorithm returns an already initialized CR's.

Let us estimate briefly the time complexity of the algorithm. It is evident from step S2 that **CRMake** "travels" along the entire parse-tree of a given input formula G. We then can estimate the cost of the algorithm for each inner node of G as follows:

- If **CRMake** returns as a result of the steps S3, S4.1 or S4.2, then the additional cost is of at most linear order in the length of the involved CR's. For example, if the property (18) is applied in step S3, then $L(\Phi)$ additional multiplications are performed at this step.

Algorithm CRMake: This algorithm constructs and returns a CR-expression Φ for a given formula $G(x)$, such that $\Phi(i) = G(x_0 + ih)$ for $i \geq 0$. (Here G is a closed form function with x as variable.)

S1 [Trivial Cases]
If G is a constant or a CR then return G
If $G \equiv x$ then return the BR $\{x_0, +, h\}$

S2 [Main Recursion]
Apply the algorithm recursively to all arguments of G, replace the original arguments of G with their constructed CR-expressions and set $\mathcal{G}$ to the resulting CR-expression.

S3 [Check for basic properties of CR's]
If $\mathcal{G}$ satisfies a LHS-assumption of one of the equalities of proposition 1 or 2 then simplify $\mathcal{G}$ using that equality and return the result of the applied transformation.

S4 [Check for recursive properties of CR's]
If $\mathcal{G}$ is equivalent to:

S4.1 $\{\varphi_0, +, f_1\} + \{\psi_0, +, g_1\}$ for some $\varphi_0, \psi_0, f_1, g_1$ return $\{\varphi_0 + \psi_0, +, \mathbf{CRMake}(f_1 + g_1)\}$

S4.2 $\{\varphi_0, *, f_1\} * \{\psi_0, *, g_1\}$ for some $\varphi_0, \psi_0, f_1, g_1$ return $\{\varphi_0 * \psi_0, *, \mathbf{CRMake}(f_1 * g_1)\}$

S4.3 $\Phi * \Psi$ and Φ, Ψ are simple, pure-sum CR's return $\mathbf{CRProd}(\Phi, \Psi)$

S4.4 Φ^{Ψ} and Φ is a simple, pure-product, Ψ a simple, pure-sum CR return $\mathbf{CRExpt}(\Phi, \Psi)$

S4.5 Φ^n and Φ is a simple,pure-sum CR, $n \in \mathbf{N}$ return $\mathbf{CRMake}(\Phi * \Phi^{n-1})$

S5 [Check for factorial case]
If $\mathcal{G} \equiv \{\varphi_0, +, \varphi_1\}!$, $\varphi_1 \in \mathbf{Z}$ and
$\varphi_1 > 0$ return $\{\varphi_0!, *, \mathbf{CRMake}(\prod_{l=1}^{\varphi_1} \{\varphi_0 + l, +, \varphi_1\})\}$
$\varphi_1 < 0$ return $\{\varphi_0!, *, \mathbf{CRMake}(\prod_{l=1}^{|\varphi_1|} \{\varphi_0 + l, +, \varphi_1\})^{-1}\}$

S6 [All else failed] Return $\mathcal{G}$.

Figure 2: The general CR construction algorithm

- Before **CRMake** returns as a result of the steps S4.3, S4.4, S4.5 or S5, the algorithm **CRProd** (resp. **CRExpt**) is subsequently called. Hence the additional cost depends on the complexity of **CRProd**.
- If $k = L(\Phi)$, $l = L(\Psi)$, $k \geq l$ then $\mathbf{CRProd}(\Phi, \Psi)$ (resp **CRExpt**) performs at least $k^2 2^l$ arithmetic operations, i.e. the additional cost of **CRProd** is $\Omega(k^2 2^l)$.

It can be concluded from this complexity analysis that the algorithm needs $\Omega(n^4)$ time in order to construct a CR for a dense polynomial of degree n or even needs exponential time for constructing a CR for products of polynomials. Such a time behavior leads naturally to very long CR construction times for more complex input formulas. Therefore, it is desirable to reduce the complexity of the CR construction algorithm. In the following subsection we propose an improvement of **CRMake** which drastically decreases the construction time for some common cases.

4.1 An improvement of CRMake

The results of this subsection are based on the following proposition:

Proposition 5 *Given two simple, pure-sum CR's* $\Phi = \{\varphi_0, +, \varphi_1, +, \ldots, +, \varphi_k\}$ *and* $\Psi = \{\psi_0, +, \psi_1\}$, *we have*

$$\Phi * \Psi = \{\bar{\varphi}_0, +, \bar{\varphi}_1, +, \ldots, +, \bar{\varphi}_k, +, \bar{\varphi}_{k+1}\} \quad (26)$$

where $\bar{\varphi}_0 = \varphi_0\psi_0, \bar{\varphi}_{k+1} = (k+1)\varphi_k\psi_1$, *and* $\bar{\varphi}_i = i\varphi_{i-1}\psi_1 + (\psi_0 + i\psi_1)\varphi_i$ *for* $i = 1, 2, \ldots, k$.

Proof: (By induction on k) Suppose $k = 1$:

$\{\varphi_0, +, \varphi_1\} * \{\psi_0, +, \psi_1\}$
$= \{\varphi_0\psi_0, +, \varphi\psi_1 + (\psi + \psi_1)\varphi_1\}$ by (13)
$= \{\varphi_0\psi_0, +, \varphi_0\psi_1 + (\psi_0 + \psi_1)\varphi_1, +, 2\varphi_1\psi_1\}$ by (6), (7)

Now let $k > 1$:

$\{\varphi_0, +, \varphi_1, +, \ldots, \varphi_k\} * \{\psi_0, +, \psi_1\}$
$= \{\varphi_0\psi_0, +, \{\varphi_0\psi_1, +, \varphi_1\psi_1, +, \ldots, +, \varphi_k\psi_1\} +$
$\{\varphi_1, +, \ldots, +, \varphi_k\} * \{\psi_0 + \psi_1, +, \psi_1\}\}$
by (6), (13) and (18)
$= \{\varphi_0\psi_0, +, \{\varphi_0\psi_1, +, \varphi_1\psi_1, \ldots, \varphi_k\psi_1\} +$
$\{(\psi_0+\psi_1)\varphi_1, +, \varphi_1\psi_1+(\psi_0+2\psi_1)\varphi_2, \ldots, k\varphi_k\psi_1\}\}$
by induction hypothesis
$= \{\varphi_0\psi_0, +, \varphi_0\psi_1+\varphi_1(\psi_0+\psi_1), +, 2\varphi_1\psi_1+(\psi_0+2\psi_1)\varphi_2,$
$+, \ldots, +, (k+1)\varphi_k\psi_1\}$ #

It is interesting to observe that proposition 5 can be both, stated and proved entirely using the concepts of factorial polynomials and lemma 1. In fact, in light of identity (5), we can apply proposition 5 to obtain the following recurrence formula for the constant expressions $c_{in}(x_0, h)$:

$$c_{in} = \begin{cases} x_0^n, & \text{if } i = 0; \\ n!\,h^n, & \text{if } i = n; \\ i\,h\,c_{i-1,n-1} + (x_0 + i\,h)c_{i,n-1}, & \text{if } 0 < i < n; \end{cases} \quad (27)$$

In other words, CR's for x^j for all $j \leq n$ can be constructed in $o(n^2)$ time using a simple loop which realizes (27).

Based on the results above, we propose to add the following two steps to **CRMake**

S1.1 If $G \equiv x^j, j \in \mathbf{N}$ then
return the symbolic CR $\{c_{0j}, +, c_{1j}, +, \ldots, +, c_{jj}\}$

S3.1 If $\mathcal{G}$ satisfies the assumption of proposition 5 then
return the result of the application of (26)

and the following step to **CRProd**

P1.1 If $l = 1$ then
return the result of the application of (26).

The idea behind step S1.1 can be illustrated by an example. Consider the polynomial $p(x) = x^9 + x^8$. The original algorithm first constructs a pure-sum CR of length 9 for x^9 using the time-expensive algorithm **CRProd**, then a pure-sum CR of length 8 for x^8, again using **CRProd** and finally adds the two CR's using property (22) to output a simple, pure-sum CR of length 9. As a result of step S1.1, the modified algorithm returns the CR $\{c_{09}+c_{08}, +, \ldots, +, c_{89}+c_{88}, +, c_{99}\}$ after only three steps.

In general, the modified CR construction algorithm returns a CR-expression which may not only contain the symbols x_0 and h but also symbols c_{ij} with $0 \leq i \leq j \leq n$ for some $n > 0$. Subsequently, the CR initialization procedure has to be modified appropriately by computing the values of the c_{ij}'s in the way described above before the CR can be initialized as we discussed in section 2.1.

In summary, proposition 5 provides the following improvements of **CRMake**:

1. CR's for x^n, and hence for nth degree polynomials, can be constructed and initialized in $o(n^2)$ time.
2. The number of recursive calls of **CRMake** and **CRProd** in steps S4.3, S4.5, and S5 is significantly reduced.

5 Implementing the CR method

To realize the CR method, we need to implement three distinct mechanisms: CR construction, CR initialization, and CR evaluation. These are described here together with actual timings. We did our work on the MAXIMA system. The same code can easily be adopted on other systems with little change. The CR mechanisms are used to interface the SIG graphing package components **mgraph** and **xgraph** resulting in significant speed improvements.

5.1 CR construction and initialization

The routines **CRMake** and **CRInit** for constructing and initializing CR's are written in Common Lisp (CL) accessible from either the Lisp or the top level of MAXIMA. The function **CRMake** takes four arguments: a symbolic function $G(x)$, the name of the variable of G, and the parameters x_0, h. If x_0, h are symbols, then **CRMake** returns a symbolic CR, otherwise the values of x_0, h are used to directly construct an initialized CR. The function **CRInit** takes a symbolic CR and equalities of the form **variable=value** as input and initializes the CR by evaluating all constant expressions using the given values (for x_0, h, etc.). A symbolic CR can be initialized with any valid MAXIMA value resulting in a CR whose evaluation yields the respective value type. For example, we can initialize a CR to produce floating point, rational, complex or bigfloat values.

Construction and initialization timings[4] for the five examples (Fig. 1) are shown in Table 1. For G_2 we choose an 11th degree polynomial with rational coefficients whose CR construction yields a simple, pure-sum CR of length 11.Each table entry shows two timings: the first deals with the original CR construction and initialization algorithm and the second shows the same data for the improved **CRMake** algorithm. The table examines constructing a symbolic CR (column 1), initializing the CR with rational numbers (column 2), and constructing the same initialized CR directly (column 3).

	Symbolic **CRMake**	Rational **CRInit**	Rational **CRMake**
1	52/12	10/6	45/9
2	265/33	90/25	110/33
3	3/2	1/1	2/1
4	4/2	1/1	2/1
5	12/5	6/3	8/6

Table 1: CR Construction Timings in ms

The above timings clearly illustrate the improvement gained by the modified CR construction and initialization algorithms. Furthermore, CR's with symbolic parameters are clearly more expensive to construct than those with numeric parameters. However, once a symbolic CR is constructed, its initialization is still the fastest way to obtain a "ready-to-evaluate" CR.

[4] All timings of this section were obtained using a lightly loaded Sun SPARCstation2 with 32MB main memory

5.2 Evaluation of CR's

Once a CR for $G(x)$ is constructed and initialized, it can be used to produce the values $G(x_0 + ih)$ very efficiently. Evaluation can proceed using a *direct interpretation* or a *code generation* approach.

In the *direct interpretation* approach, we treat the tuple representation of a CR as a *program*. A general routine is used to *interpret* any CR tuple. Each interpretation pass produces the next evaluation value and modifies the tuple for the next pass. To illustrate this approach, let us consider the evaluation of a simple CR. Suppose the CR tuple is stored as a list `CR`. Then the following MAXIMA function `CREval` evaluates `CR` for n points, yielding the result values in the array `Result`. This approach obviously works in general.

```
CREval(CR,l,n,Result):=
  for i:0 thru n-1 do
  (  Result[i] = CR[1],
     for j:2 thru 2*l step 2 do
        CR[j-1]: if equal("+",CR[j])
                   then CR[j-1]+CR[j+1]
                 else CR[j-1]*CR[j+1]
  );
```

Note that the function `CREval` is a rendition of a much more efficient and complete implementation in CL.

To make the CR evaluation even faster, we can generate compilable code whose execution yields the values of a specific CR. We implemented the MAXIMA function `CRCodeGen` which, given an initialized CR tuple, produces efficient C code to calculate the evaluations. For example, with $x_0 = 0, h = 0.01$ `CRCodeGen` generates the following C function `CCREval` for Φ_1.

```
void CCREval(int n, double Result)
{   double CR0 = 1.3591409142295225;
    double CR1 = 0.98422045134067937;
    double CR2 = 1.0004674797985269;
    double CR3 = 1.000006000018;
    for (int i=0 ; i<n ; i++)
    {   Result[i] = CR0;
        CR0 *= CR1;
        CR1 *= CR2;
        CR2 *= CR3;
    }
}
```

Table 2 shows timings with the interpretation approach. Again, we use the examples 1-5. The evaluation of examples 1-3 was done for 1000 points using floating point numbers, whereas the evaluation of examples 4 and 5 was done for 100 points using rational numbers. Each table entry shows three timings: normal MAXIMA evaluation of the original formula (column 1), Horner's rule applied whenever possible (column 2), and the CR interpretation (column 3).

	NormalEval	HornerEval	CREval	Speedups
1	1614	1325	155	10.5 / 8.5
2	3074	1248	357	8.6 / 3.5
3	428	428	160	2.7
4	708	708	31	20.5
5	3103	2567	209	14.5/12.3

Table 2: Timings of MAXIMA evaluation routines in ms

Table 3 shows timings of automatically generated evaluation procedures in C. Only examples 1-3 are considered here. The respective routines were generated by the above described MAXIMA routine `CRCodeGen` and compiled using the gcc compiler. All evaluations were done for 1000 points using `double` values.

	NormalEval	HornerEval	CREval	Speedups
1	3240	2980	50	64 / 59
2	680	240	90	7.6 / 2.7
3	1990	1990	480	4.2

Time for program generation and compilation: $\approx$ 3 seconds

Table 3: Timings of compiled C-evaluation procedures in μs

Obviously, the speedups gained by either CR evaluation methods are substantial – even for such relatively simple formulas as G_3 and even in comparison with some "faster" evaluation techniques, such as the Horner evaluation for polynomials. We can furthermore notice that the speedup depends on the evaluation method, the data type of the values, and the nature of the closed-form formula.

One might wonder if there are cases where a CR takes longer to evaluate then the naive approach. Our experiments indicate that the additional evaluation overhead of the CR is almost negligible in comparison to the total evaluation time and, in the worst case, the CR evaluation is basically equivalent to the naive evaluation. In other words, a CR evaluation may not be faster in certain cases but neither is it significantly slower.

A CR evaluation yields algebraically the same result as a normal evaluation of the original formula. However, this changes when floating point numbers are used (as is illustrated in table 4). Because a CR uses a different sequence of arithmetic operations, it is clear that its numeric error properties are different. Potentially, a CR evaluation can result in bigger numerical errors, because errors can accumulate from point to point. Therefore, if the numeric error exceeds a certain limit, we might have to "refresh" the CR evaluation by reinitializing the symbolic CR.

The timings obtained also clearly illustrate the advantages and disadvantages of the two different evaluation methods. There is certainly no general "recipe" that one can apply to determine which of the methods to use. This depends very much on the characteristics of the specific application and on the sizes of the problems involved. We discuss next a typical application of the above methods.

5.3 Application in SIG

SIG is a graphics system [5] for the display of curves and surfaces defined by mathematical formulas in a symbolic system. It uses MAXIMA/CL for straightforward evaluations of symbolic formulas to generate a set of points for the desired curve or surface. Then, a stand-alone C-X11-based graphics facility does the display. The point generation part of SIG is modified to employ the CR method. Before a formula is evaluated, it is transformed into a CR-expression which is then interpreted (`CREval`) to generate the points. Table 4 shows the improved timings for three curves (examples 1-3) in the interval [-5,5] and a stepwidth of 0.05.

	CRMake	CREval	SIG	Speedup	Rel. Error
1	3	31	322	9.4	10^{-9}
2	5	71	614	8.0	10^{-11}
3	1	32	86	2.6	10^{-12}

Table 4: Timings of modified SIG evaluation in *ms*

The results indicate clearly that the initial cost of constructing a CR is very small in comparison to the total evaluation time and that any possible extra numerical errors introduced by the CR evaluation are tolerable. Therefore, the CR technique can be applied successfully in practice. The impact will be more significant after we extend the CR method to functions of more than one variable.

6 Summary

The CR technique can be very effective for the evaluation of a function at regular intervals. The CR construction algorithm is simple to implement and executes efficiently. Once constructed, a CR tuple can be interpreted as a program to produce function values. This approach is suitable for dynamic applications. The CR technique can also be used to generate efficient code to be compiled and therefore be applied to automatically optimize source code of numerical and algebraic programs.

Our MAXIMA implementation and the SIG application demonstrate the practical value of the CR method. Much future work lies ahead: to extend the general concept of CR's to second or higher order recurrences, to treat functions in more than one variable, and to handle trigonometric functions. Furthermore, the numeric properties of the CR evaluation should be analyzed in more detail.

Finally, CR's are evaluated in a vector-like fashion, whereas functions are normally evaluated in a tree-like fashion. Therefore, CR's also promise to provide new and more efficient means to parallelize function evaluations on modern high-performance computers.

Acknowledgments

We would like to thank all referees for their very helpful suggestions and comments. We are especially grateful to one of the referees who suggested to use the "chain" feature for defining CR's and who pointed out the close relationship of factorial polynomials and CR's.

References

[1] William H. Beyer. *CRC Standard Mathematical Tables.* CRC Press Inc., Boca Raton, Florida, 27th edition, 1984.

[2] The MATHLAB Group. *Macsyma Reference Manual, version nine.* Laboratory for Computer Science, M.I.T., Cambridge Mass., 1977.

[3] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming.* Reading Mass.:Addison Wesley, 1981.

[4] Peter A. Stark. *Introduction to Numerical Methods.* The Macmillan Company, New York, 1970.

[5] Paul S. Wang. A system independent graphing package for mathematical functions. In *Design and and implementation of symbolic computation systems : International Symposium, DISCO '90*, Capri, Italy, April 1990. LNCS 429, Springer Verlag.

[6] Eugene V. Zima. Recurrent relations and speed-up of computations using computer algebra systems. In *Design and and implementation of symbolic computation systems : International Symposium , DISCO '92*, Bath, U.K., April 1992. LNCS 721, Springer Verlag.