Multidimensional Chains of Recurrences*

V. Kislenkov V. Mitrofanov

Dept. of Comp. Math. & Cybernetics, MSU
Moscow 119899, Russia
{kvv, mitrofan}@cs.msu.su

E. Zima

Symbolic Computation Group
University of Waterloo
Waterloo, Canada
ezima@daisy.uwaterloo.ca

Abstract

A technique to expedite iterative computations which is based on multidimensional chains of recurrences (MCR) is presented. Algorithms for MCR construction, interpretation and MCR-based code generation are discussed. The notion of delayed MCR simplification introduced here for the first time often leads to reduced times for both the MCR construction and MCR interpretation phases of this technique. Three different implementations of the MCR technique (in Maple, C and Java) are described.

1 Introduction

We consider the problem of expediting computational tasks like this: given a closed form function $G(x_1, \ldots, x_m)$, initial points $x_{01}, \ldots, x_{0m}$ and steps $h_1, \ldots, h_m$, compute values $G(x_1, \ldots, x_m)$ for $x_1 = x_{01}, x_{01} + h_1, \ldots, x_{01} + n_1 h_1; \ldots; x_m = x_{0m}, x_{0m} + h_m, \ldots, x_{0m} + n_m h_m$. Many problems appearing in such applications as plotting (explicit, parametric, implicit) in different coordinate systems, animation, signal processing etc. can be reduced to this kind of computation. In order to expedite the computations a technique based on chains of recurrences (CR) can be used. The application of this technique consists of two steps: (1) algebraic conversion of the initial computational scheme into chains of recurrences and (2) numerical interpretation of the chain based computational scheme. Algorithms to construct and interpret linear and two-dimensional chains of recurrences have been considered in [4, 10, 11, 12, 14, 16] together with implementations of both steps mentioned above within different computer algebra systems (CAS) and as standalone software. It was shown in the linear case ([11]) that a CR-based representation of expressions to be evaluated in loops is a canonical representation of polynomials and rational functions. For polynomials this is an analog of the usual dense form ([6]).

A general algorithm for constructing an optimal computational scheme for any given expression G is impossible to develop. We consider an approach to getting a "reasonable" solution, which is based on the technique of multidimensional chains of recurrences (MCRs). To understand what we mean by "reasonable" solutions consider a simple example. Exercise 4.6.3-32 from [1]: given real x and natural m, compute the set of values $x, x^4, x^9, \ldots, x^{m^2}$ in the minimal number of multiplications. From [2] we know that the lower bound for the number of multiplications is larger than $m + m^{2/3-\epsilon}$ for any $\epsilon > 0$ (see also [3]). But for many practical purposes one can agree with the "reasonable" solution given by the CR technique:

```
y:=x; y2:=y*x; y1:=y2*x;
for i:=2 to m do y:=y*y1; y1:=y1*y2 od
```

The number of multiplications here is $2m$. This solution is reasonable because it is simple, easy to implement, and easy to extend on similar problems such as $G(i) = x^{3i^2-i+1}$.

The main goals of this paper are:
— to define multidimensional chains of recurrences (MCR) in a general way;
— to give the general interpreting scheme for MCRs;
— to describe Maple, C and Java implementations of the MCR technique;
— to discuss in the context of MCR evaluation such transformations as delayed simplifications, variable ordering etc., which are useful for improving the characteristics of CR-based computations.
The generality applied here allows us to use CR-based computations without restrictions on the dimension of computational tasks, domain of computations or the type of functions involved in the expression G. For example, the initial points and steps can be rational, floating point, complex, algebraic, or symbolic. The expression G can be arbitrary: it may include user-defined, piecewise or undefined functions.

The CR technique is based on a special form of representation of polynomials. We will explain each new concept in the domain of polynomial evaluation (independent of the evaluation algorithm) and then apply it to the MCR case.

2 Multidimensional chains of recurrences

2.1 Preliminaries

Before defining MCRs we reformulate (substituting $x_j = x_{0j} + i_j h_j$) the initial computational task to the equivalent one:

$$\begin{array}{l}\text{Compute } F(i_1, \ldots, i_m) \\ \text{for } i_1 = 0, 1, \ldots, n_1; \ldots; i_m = 0, 1, \ldots, n_m;\end{array} \quad (1)$$

*This work was supported in part by National Science and Engineering Research Council of Canada.
 ISSAC'98, Rostock, Germany.

and keep $x_{0j}, h_j, j = 1, \ldots, m$ as parameters. It will be shown at the end of this section how to proceed with the initial computational task.

Generally speaking the problem considered here is one of optimization of nested loops

```
for i1  := 0 to n1   do
 .....
  for im-1 := 0 to nm-1 do
    for im  := 0 to  nm   do
       compute_and_store( F(i1,...,im) )          (*)
    od;
  od;
   .....
od;
```

Optimization of such loops has to involve different known optimizing techniques, such as loop fusion, common subexpressions elimination, strength reduction, loop unfolding etc. The MCR technique is shown to be complementary to all of these transformations. Therefore, concentrating on MCRs we will make some natural assumptions about the expression F. Let l_j be maximal degree of polynomial in i_j subexpressions of F. One of the assumptions is that $l_j << n_j$, $j = 1, \ldots, m$. This means that it is unreasonable to apply the MCR technique to a task like $\sin(i^{98})/(1 + i^{100}), i = 0, 1, 2, 3$ (here $n = 3, l = 100$). For tasks like this it is better to unfold the corresponding loop in i, or to apply the mixed binary powering scheme. However a task such as $\sin(i^2)/(1 + i^4), i = 0, 1, \ldots, 100$ is quite amenable to CR techniques. It is interesting that the `plot3d` and `animate3d` help pages in Maple, MuPad, Mathematica contain only examples for which this assumption holds. Computer algebra systems can generally provide honest plotting [7] only in these reasonable cases. There is another reason to bound the degrees of polynomial subexpressions of F when CR techniques are used — the weak numeric stability of the CR scheme. Approaches to analysis of the numeric stability can be found in [15, 18]. It was shown in [18] that the MCR technique can be used to analyse and improve the numeric stability of MCR schemes on the fly. One of the possible solutions is to increase the dimension of the computational task with a simultaneous decrease of the values of n_j.

However, even for "reasonable" tasks there are still many factors affecting the MCR scheme. The discussion of some of the approaches to obtaining reasonably fast MCR schemes in a reasonably short time, is one of the main purposes of this paper.

Linear CR techniques [4, 10, 11] can be applied in a straightforward way to `(*)`, proceeding from the innermost loop. There are obvious drawbacks to the straightforward application: an explosion in the number of intermediate assignments and the necessity to reparse new expressions which in fact were constructed from atomic elements in the previous steps of the optimization process. These observations lead to a need for defining MCRs and construction algorithms which require only one parse of the expression tree.

2.2 Definitions

Consider computational task (1). Let the order $\prec$ be defined over the set of variables $i_1, i_2, \ldots, i_m$ such that $i_1 \prec i_2 \prec \ldots \prec i_m$. Considering any expression which does not depend on $i_1, \ldots, i_m$ as a 0-dimensional chain of recurrences, we define an m-dimensional chain of recurrences in $i_1, \ldots, i_m$ recursively: given functions $\varphi_0(i_1, \ldots, i_{m-1}), \ldots, \varphi_{k-1}(i_1, \ldots, i_{m-1})$, defined over $(N \cup \{0\})^{m-1}$, a function $f_k(i_1, \ldots, i_m)$ defined over $(N \cup \{0\})^m$, and operators $\odot_1, \ldots, \odot_k$ which equal either $+$ or $*$, we call an *m-dimensional chain of recurrences in* $i_1, \ldots, i_m$ the set of functions $f_0(i_1, \ldots, i_m), f_1(i_1, \ldots, i_m), \ldots, f_k(i_1, \ldots, i_m)$ connected in such a way that for $0 \le j < k$

$$f_j(i_1, \ldots, i_m) = \begin{cases} \varphi_j(i_1, \ldots, i_{m-1}), \\ \quad \text{if } i_m = 0, \\ f_j(i_1, \ldots, i_{m-1}, i_m - 1) \odot_{j+1} \\ \quad f_{j+1}(i_1, \ldots, i_{m-1}, i_m - 1), \\ \quad \text{if } i_m > 0. \end{cases} \tag{2}$$

Further, to denote MCRs (2), we will use the linear notation $f_0(i_1, \ldots, i_m) = \Phi(i_1, \ldots, i_m) = \{_{i_m} \varphi_0, \odot_1, \varphi_1, \odot_2, \varphi_2, \ldots \odot_k, f_k\}_{i_m}$ and call an expression Φ an m-dimensional CR-expression in $i_1, \ldots, i_m$ if it represents one of the following functions over $(N \cup \{0\})^m$:

- an $(m-1)$-dimensional CR-expression in $i_1, \ldots, i_{m-1}$,
- an m-dimensional CR $\{_{i_m} \varphi_0, \odot_1, \varphi_1, \ldots \odot_k, f_k\}_{i_m}$, where f_k is an m-dimensional CR-expression in $i_1, \ldots, i_m$ and $\varphi_0, \ldots, \varphi_{k-1}$ are $(m-1)$-dimensional CR-expressions in $i_1, \ldots, i_{m-1}$,
- a function $P(\Phi^{(1)}, \ldots, \Phi^{(s)})$, where $\Phi^{(1)}, \ldots, \Phi^{(s)}$ are m-dimensional CR-expressions.

Example 1. Function $2^{j^3 - 5j^2 + 1}\left(i^2 - 1\right)$ has the following representation as a two-dimensional CR in j, i:

$$\begin{array}{l} \Phi(j,i) = \{_i \{_j -2, *, \frac{1}{16}, *, \frac{1}{16}, *, 64\}_j, +, \\ \{_j 2, *, \frac{1}{16}, *, \frac{1}{16}, *, 64\}_j, +, \{_j 4, *, \frac{1}{16}, *, \frac{1}{16}, *, 64\}_j \}_i . \end{array} \tag{3}$$

For an m-dimensional CR

$$\Phi(i_1, \ldots, i_m) = \{_{i_m} \varphi_0, \odot_1, \varphi_1, \ldots \odot_k, f_k\}_{i_m} \tag{4}$$

we call i_m the *main variable* ($\mathtt{mvar}(\Phi) = i_m$, $\mathtt{mvar}(C) \prec i_1$ for any constant expression C) and keep most of the notation from the theory of linear CRs [11] with the additional words "w.r.t. the main variable" in the appropriate places. In particular for MCR (4) we call

- $k = L_{i_m}(\Phi)$ the *length* of Φ w.r.t. i_m,
- Φ a *pure-sum* MCR w.r.t. the main variable, if $\odot_1 = \ldots = \odot_k = +$,
- Φ a *pure-product* MCR w.r.t. the main variable, if $\odot_1 = \ldots = \odot_k = *$,
- Φ a *simple* MCR w.r.t. i_m, if f_k is $(m-1)$-dimensional MCR (i.e. if f_k does not depend on the main variable),
- $\mathtt{Comp}(j, \Phi) = \varphi_j$ the *j-th component* of a simple CR Φ,
- $\Phi_r = \{_{i_m} \varphi_r, \odot_{r+1}, \varphi_{r+1}, \ldots \odot_k, f_k\}_{i_m}$, $(0 \le r \le k)$ – *the r-order subchain* of the MCR Φ w.r.t. i_m,

and so on. Therefore, in (3) $\mathtt{mvar}(\Phi(j,i)) = i$; $\Phi(j,i)$ is simple pure-sum MCR of length 2 w.r.t. i; $\mathtt{Comp}(1, \Phi(j,i)) = \{_j 2, *, \frac{1}{16}, *, \frac{1}{16}, *, 64\}_j$ is a one-dimensional CR in j, and it is simple pure-product of length 3.

Of course changing the order of variables $i_1, \dots, i_m$ will change the internal representation (similarly to the polynomial case). Return to the function $2^{j^3-5*j^2+1}(i^2-1)$. It has the following representation as two-dimensional CR in i, j:

$$\Psi(i,j) = \{_j \{_i -2, +, 2, +, 4\}_i, *, \frac{1}{16}, *, \frac{1}{16}, *, 64\}_j . \qquad (5)$$

This differs essentially from (3): $\Psi(i,j)$ is a simple pure-product MCR of length 3 w.r.t. the main variable j and $\mathtt{Comp}(0,\Psi)$ is a simple pure-sum one-dimensional CR of length 2 w.r.t. the main variable i.

From here we will denote MCR (4) by $\Phi = \{_{i_m} \varphi_0, \odot_1, \Phi_1\}_{i_m}$ where Φ_1 is a first-order subchain of Φ. This allows us to formulate MCR construction and interpretation rules in condensed form.

2.3 MCR construction

As operations on multivariate polynomials in recursive form are implemented using operations on univariate polynomials [6], the algorithm for constructing the MCRs by a given function $F(i_1, \dots, i_m)$ can be obtained easily from the algorithm to construct linear CRs [11]. The algorithm begins by replacing all occurrences of i_l by the simple MCR $\{_{i_l} 0, +, 1\}_{i_l}$. Then MCR simplifying rules are applied to the obtained MCR expression recursively. These rules are just slight modifications of the linear CR-simplifying rules described in [11]. For example, given two MCRs Φ, Ψ and $u = \mathtt{mvar}(\Phi), v = \mathtt{mvar}(\Psi)$, the rule to simplify the MCR expression $\Phi + \Psi$ is:

$$\mathtt{Simplify}(\Phi+\Psi) = \begin{cases} \{_v \mathtt{Simplify}(\Phi+\psi_0), +, \Psi_1\}_v, \\ \quad \text{if } u \prec v \text{ and } \Psi = \{_v \psi_0, +, \Psi_1\}_v \ , \\ \{_u \mathtt{Simplify}(\Psi+\varphi_0), +, \Phi_1\}_u, \\ \quad \text{if } v \prec u \text{ and } \Phi = \{_u \varphi_0, +, \Phi_1\}_u \ , \\ \{_u \mathtt{Simplify}(\varphi_0+\psi_0), +, \mathtt{Simplify}(\Phi_1+\Psi_1)\}_u, \\ \quad \text{if } u = v \text{ and } \Phi = \{_u \varphi_0, +, \Phi_1\}_u \ , \ \Psi = \{_v \psi_0, +, \Psi_1\}_v \ , \\ \Phi + \Psi, \text{ otherwise.} \end{cases}$$

One observes that when $\mathtt{mvar}(\Phi) \prec \mathtt{mvar}(\Psi)$ and $\Psi = \{_v \psi_0, +, \Psi_1\}_v$, MCR Φ behaves as a constant expression (in the corresponding simplification rule for linear CRs [11]) with respect to MCR Ψ. For example, $j^2 = \{_j 0, +, 1, +, 2\}_j$, $\exp(i) = \{_i 1, *, \exp(1)\}_i$ and $\{_j 0, +, 1, +, 2\}_j + \{_i 1, *, \exp(1)\}_i$ is simplified to $\{_j \{_i 1, *, \exp(1)\}_i, +, 1, +, 2\}_j$, if $i \prec j$.

All other simplification rules are constructed from linear simplification rules in the same way.

Example 2. Consider step by step the process of constructing the MCR w.r.t. j, i for the function $i^2 \exp(2j+1)$:
1. after the first (substitution) step we have the MCR expression $\{_i 0, +, 1\}_i^2 \exp(2\{_j 0, +, 1\}_j + 1)$
2. after simplifying $\{_i 0, +, 1\}_i^2$ and $\exp(2\{_j 0, +, 1\}_j + 1)$ we obtain $\{_i 0, +, 1, +, 2\}_i \{_j \exp(1), *, \exp(2)\}_j$
3. (up to this step pure linear simplifications of CRs in i and j respectively were involved) then 2-dimensional simplification gives
$\{_i 0, +, 1 * \{_j \exp(1), *, \exp(2)\}_j, +, 2 * \{_j \exp(1), *, \exp(2)\}_j\}_i$
4. after linear simplification of the last expression we get
$\{_i 0, +, \{_j \exp(1), *, \exp(2)\}_j, +, \{_j 2\exp(1), *, \exp(2)\}_j\}_i$.
Here the MCR in j behaves as a constant expression with respect to the MCR in i, and recursive calls of the `Simplify` procedure are used.

Remark. The difference in construction of the MCRs for the initial problem only appears in the first (substitution) step, when all occurrences of x_l are replaced by $\{_{i_l} x_{0l}, +, h_l\}_{i_l}$.

2.4 Interpreting tools for multidimensional CRs

Once the MCR $\Phi(i_1, \dots, i_m)$ for $F(i_1, \dots, i_m)$ is constructed, one can interpret Φ in shift-and-operate fashion. In order to describe the general MCR interpreting scheme we need two auxiliary functions. Let $u \in \{i_1, \dots, i_m\}$. We define function $\mathtt{Value}_u(\Phi)$ as

$$\begin{cases} \Phi, & \text{if } \mathtt{mvar}(\Phi) \prec u, \\ \varphi_0, & \text{if } \Phi = \{_u \varphi_0, \odot_1, \Phi_1\}_u, \\ \{_v \mathtt{Value}_u(\varphi_0), \odot_1, \mathtt{Value}_u(\Phi_1)\}_v, & \text{if } \Phi = \{_v \varphi_0, \odot_1, \Phi_1\}_v \\ & \quad \& \ u \prec v, \\ P(\mathtt{Value}_u(\Phi^{(1)}), ..., \mathtt{Value}_u(\Phi^{(s)})), & \text{if } \Phi = P(\Phi^{(1)}, ..., \Phi^{(s)}); \end{cases}$$

and the result of applying the shift operator E_u [1] to Φ as

$$\begin{cases} \Phi, & \text{if } \mathtt{mvar}(\Phi) \prec u, \\ \{_u \varphi_0 \odot_1 \mathtt{Value}_u(\Phi_1), \odot_1, E_u(\Phi_1)\}_u, & \text{if } \Phi = \{_u \varphi_0, \odot_1, \Phi_1\}_u, \\ \{_v E_u(\varphi_0), \odot_1, E_u(\Phi_1)\}_v, & \text{if } \Phi = \{_v \varphi_0, \odot_1, \Phi_1\}_v \\ & \quad \& \ u \prec v, \\ P(E_u(\Phi^{(1)}), \dots, E_u(\Phi^{(s)})), & \text{if } \Phi = P(\Phi^{(1)}, \dots, \Phi^{(s)}). \end{cases}$$

Returning to Example 2 we can write:

$$\begin{aligned} &\Phi(j,i) = \\ &= \{_i 0, +, \{_j \exp(1), *, \exp(2)\}_j, +, \{_j 2\exp(1), *, \exp(2)\}_j\}_i, \\ &\mathtt{Value}_i(\Phi(j,i)) = 0, \\ &\mathtt{Value}_j(\Phi(j,i)) = \{_i 0, +, \exp(1), +, 2\exp(1)\}_i, \\ &E_j(\Phi(j,i)) = \\ &= \{_i 0, +, \{_j \exp(3), *, \exp(2)\}_j, +, \{_j 2\exp(3), *, \exp(2)\}_j\}_i \end{aligned}$$

and so on. One can see, that when $u = \mathtt{mvar}(\Phi)$, then $\mathtt{Value}_u(\Phi)$ returns the current value of the MCR expression Φ (which is an MCR expression of smaller dimension in general); when $u \prec \mathtt{mvar}(\Phi)$, then $\mathtt{Value}_u(\Phi)$ replaces all MCRs with the main variable u by their values in MCR expression Φ (and therefore decreases the dimension of Φ as well); finally, when $\mathtt{mvar}(\Phi) \prec u$, then $\mathtt{Value}_u(\Phi)$ returns Φ unchanged. At the same time $E_u(\Phi)$ shifts an MCR expression Φ in u. Observe that if Φ is a simple MCR, then only operations $\odot_l, l = 1, \dots, k$ from (2) are evaluated at the time of shifting in u.

Now we will describe the straightforward scheme which solves the computational task (1). The nested loop to compute and to write the values of $\Phi(i_1, \dots, i_m)$ is:

```
Initialize(Φ);
for i_1  := 0 to n_1   do
  Φ^(1) := Value_{i_1}(Φ) ;
  .....
      for i_{m-1}  := 0 to n_{m-1}   do
        Φ^(m-1) := Value_{i_{m-1}}(Φ^(m-2)) ;
        for i_m  := 0 to n_m   do
          write( Value_{i_m}(Φ^(m-1)) );          (**)
          Φ^(m-1) := E_{i_m}(Φ^(m-1))
        od;
        Φ^(m-2) := E_{i_{m-1}}(Φ^(m-2))
      od;
  .....
  Φ := E_{i_1}(Φ)
od;
```

Here `Initialize(Φ)` initializes components of Φ with actual values of x_{0l}, h_l and other variables involved (if needed).

When the general interpreting scheme of MCRs is defined it is easy to implement an MCR-based code generator.

[1] $E_u(F(u)) = F(u+1)$.

For this, MCR expression Φ has to be mapped on the set of local variables (each component of every MCR in Φ corresponds to a single variable). The main loop in the generated procedure is almost identical to the loop `(**)`. The difference is that every assignment $\Phi^{(j-1)}$ `:=` $\texttt{Value}_{i_j}(\Phi^{(j-1)})$ and $\Phi^{(j-1)}$ `:=` $E_{i_j}(\Phi^{(j-1)})$ is replaced by an inline implementation of the corresponding operation. For example, for expression (3) our Maple code generator produces the following code:

```
cc3 := -2; cc4 := 1/16; cc5 := 1/16; cc6 := 64;
cc7 := 2; cc8 := 1/16; cc9 := 1/16; cc10 := 64;
cc11 := 4; cc12 := 1/16; cc13 := 1/16; cc14 := 64;
for _cc1 from 0 to nn1 do
 cc0 := cc3; cc1 := cc7; cc2 := cc11;
 for _cc2 from 0 to nn2 do
  aa1[_cc1,_cc2]:= cc0; cc0:= cc0+cc1; cc1:= cc1+cc2
 od;
 cc3 := cc3*cc4; cc4 := cc4*cc5; cc5 := cc6*cc5;
 cc7 := cc7*cc8; cc8 := cc8*cc9; cc9 := cc9*cc10;
 cc11:= cc11*cc12; cc12:= cc12*cc13; cc13:= cc13*cc14
od
```

2.5 A remark on trigonometric MCRs

In the linear case, if $P(i) = \{\xi_0, +, \xi_1, +, \ldots, \xi_p\}$ (i.e., $P(i)$ is a polynomial of degree p in i) then for $i = 0, 1, \ldots$, values of $s_0(i) = \sin P(i)$, $c_0(i) = \cos P(i)$, $\frac{s_0(i)}{c_0(i)} = \tan P(i)$, and so on, can be defined ([4, 8]) by a special form of chains of recurrences of *length* p:

$$s_{t-1}(i) = \begin{cases} \phi_{t-1}, & i = 0 \\ s_{t-1}(i-1)c_t(i-1) + \\ \qquad c_{t-1}(i-1)s_t(i-1), & i > 0 \end{cases} \tag{6}$$

$$c_{t-1}(i) = \begin{cases} \psi_{t-1}, & i = 0 \\ c_{t-1}(i-1)c_t(i-1) - \\ \qquad s_{t-1}(i-1)s_t(i-1), & i > 0 \end{cases}$$

$t = 1, 2, \ldots, p$; where $s_p(i) = \phi_p, c_p(i) = \psi_p$ and $\phi_j = \sin\xi_j$, $\psi_j = \cos\xi_j$, $j = 0, 1, \ldots, p$. Chains of this form are completely defined by sequences $\phi_0, \ldots, \phi_p$, $\psi_0, \ldots, \psi_p$ and can be presented compactly as a triple $\{tag, spart, cpart\}$, where $tag \in \{\sin, \cos, \tan, \cot, \sec, \csc\}$, $spart = [\phi_0, \ldots, \phi_p]$ and $cpart = [\psi_0, \ldots, \psi_p]$. Let $\mathtt{A} = \{\alpha_0, *, \alpha_1, *, \ldots, *, \alpha_k\}$ and $tag \in \{\sin, \cos\}$. The only additional simplification rule for constructing linear CR-expressions involving CRs (6) is:

$$\{tag, [\phi_0, \ldots, \phi_p], [\psi_0, \ldots, \psi_p]\} * \mathtt{A} =$$

$$= \begin{cases} \{tag, [\phi_0\alpha_0, \ldots, \phi_k\alpha_k, \phi_{k+1}, \ldots, \phi_p], \\ \quad [\psi_0\alpha_0, \ldots, \psi_k\alpha_k, \psi_{k+1}, \ldots, \psi_p]\}, & \text{if } k \le p, \\ \{tag, [\phi_0\alpha_0, \ldots, \phi_{p-1}\alpha_{p-1}, \Phi_p], \\ \quad [\psi_0\alpha_0, \ldots, \psi_{p-1}\alpha_{p-1}, \Psi_p]\}, & \text{if } k > p, \\ \quad \text{where } \Phi_p = \{\phi_p\alpha_p, *, \mathtt{A}_{p+1}\}, \\ \quad \Psi_p = \{\psi_p\alpha_p, *, \mathtt{A}_{p+1}\}. \end{cases}$$

This rule of course includes the case of multiplying CR (6) by a constant expression α_0. Replacing subtraction by addition in (6) we obtain a chained representation for hyperbolic trigonometric functions of polynomials. An analog of the above simplification rule holds for them too, when $tag \in \{\sinh, \cosh\}$.

In order to be able to interpret trigonometric CRs it is enough to add some more functionality to functions `Value` and E. Let Φ be a linear trigonometric CR. Addition functionality of E is defined by (6). Additions to `Value` are

$$\texttt{Value}(\Phi) = \begin{cases} \phi_0, & \text{if } tag = \sin, \\ \psi_0, & \text{if } tag = \cos, \\ \phi_0/\psi_0, & \text{if } tag = \tan, \\ \ldots \end{cases}$$

No other changes to CR construction or interpretation rules are needed. Using this representation of trigonometric CRs allows us to apply this technique to computational tasks in the complex domain. Representations of trigonometric CRs [14] which use properties such as
$\sin(\{\xi_0, +, \xi_1, + \ldots, \xi_p\}) =$
$Im(\{\cos\xi_0 + I\sin\xi_0, *, \ldots, *, \cos\xi_p + I\sin\xi_p\})$,
do not work in the general case.

We finish this remark with the note that the multidimensional version of trigonometric CR simplification and interpretation is obvious.

2.6 Estimating the complexity of MCR computations

In order to be able to compute the complexity of the interpretation scheme, we use the function CI_u (where $u \in \{i_1, \ldots, i_m\}$) which is called *the Cost Index* of the MCR expression Φ with respect to u :

$$CI_u(\Phi) = \begin{cases} 0, \text{ if } \texttt{mvar}(\Phi) \prec u, \\ 1 + CI_u(\Phi_1), \text{ if } \Phi = \{_u\, \varphi_0, \odot_1, \Phi_1\}_u, \\ CI_u(\varphi_0) + CI_u(\Phi_1), \text{ if } \Phi = \{_v\, \varphi_0, \odot_1, \Phi_1\}_v \,\&\, u \prec v, \\ 6p, \\ \quad \text{if } \Phi \text{ is trigonometric MCR in } u \text{ of the length } p, \\ \sum_{j=0}^{p}(CI_u(spart_j) + CI_u(cpart_j)), \\ \quad \text{if } \Phi \text{ is trig. MCR in } v \text{ of the length } p,\ u \prec v, \\ CI_u(\Phi^{(1)}) + \ldots + CI_u(\Phi^{(s)})) + q, \\ \quad \text{if } \Phi = P(\Phi^{(1)}, \ldots, \Phi^{(s)}), \end{cases}$$

where q is the number of operations in the expression P.

This definition follows from definitions of E_u and $\texttt{Value}_u(\Phi)$: $CI_u(\Phi)$ is equal to the number of operations to be evaluated in order to shift Φ in u and to compute $\texttt{Value}_u(\Phi)$. By using this function we can express the computational complexity of the general interpreting scheme as

$$\sum_{j=1}^{m} CI_{i_j}(\Phi) \prod_{l=1}^{j}(n_l + 1).$$

For our two-dimensional example (3) we have: $CI_i(\Phi) = 2$, $CI_j(\Phi) = 9$ and the complexity of MCR-based computations is equal to $9(n_2 + 1) + 2(n_1 + 1)(n_2 + 1)$. Consideration of the same example with changed order of variables (5) gives $CI_i(\Phi) = 2$, $CI_j(\Phi) = 3$ and the complexity of MCR-based computations is equal to $2(n_1 + 1) + 3(n_2 + 1)(n_1 + 1)$.

In addition to the cost index, we introduce another complexity function $l_u(\Phi)$ which is defined as follows:

$$\begin{cases} 0, \text{ if } \texttt{mvar}(\Phi) \prec u, \\ 1 + l_u(\Phi_1), \text{ if } \Phi = \{_u\, \varphi_0, \odot_1, \Phi_1\}_u, \\ \max(l_u(\varphi_0), l_u(\Phi_1)), \text{ if } \Phi = \{_v\, \varphi_0, \odot_1, \Phi_1\}_v \,\&\, u \prec v, \\ 6p, \\ \quad \text{if } \Phi \text{ is trigonometric MCR in } u \text{ of the length } p, \\ \max_{j=0,\ldots,p}(l_u(spart_j), l_u(cpart_j)), \\ \quad \text{if } \Phi \text{ is trig. MCR in } v \text{ of the length } p,\ u \prec v, \\ l_u(\Phi^{(1)}) + \ldots + l_u(\Phi^{(s)})), \\ \quad \text{if } \Phi = P(\Phi^{(1)}, \ldots, \Phi^{(s)}). \end{cases}$$

It is easy to see from their definitions, that $l_u(\Phi) \le CI_u(\Phi)$ and the value $l_u(\Phi)$ does not depend on the order of variables (we will use this fact later). Returning to expression

(3), we have $l_i(\Phi) = 2, l_j(\Phi) = 3$. For expression (5) we still have $l_i(\Phi) = 2, l_j(\Phi) = 3$.

During conversion of a given computational task (1) into an MCR expression Φ it is straightforward to compute vectors $\mathbf{l} = (l_1, \ldots, l_m)$, $\mathbf{c} = (c_1, \ldots, c_m)$ such that $l_j = l_j(\Phi)$, $c_j = CI_j(\Phi)$. We can use the fact that any constant expression can be associated with vectors $\mathbf{l} = (0, \ldots, 0)$, $\mathbf{c} = \mathbf{l}$. Each variable i_j can be associated with $\mathbf{l} = (\delta_{1j}, \ldots, \delta_{mj})$, $\mathbf{c} = \mathbf{l}$. Given $\Phi', \mathbf{l}', \mathbf{c}'$ and $\Phi'', \mathbf{l}'', \mathbf{c}''$ it is easy to compute $\mathbf{l}, \mathbf{c}$ for Φ at the time of application of the simplification rule $\Phi' \odot \Phi'' \to \Phi$. From now on we asssume that whenever we have an MCR expression Φ we also have the associated complexity vectors $\mathbf{l}, \mathbf{c}$. Our complexity measure depends only on the number of operations required. It does not distinguish differences in complexity between operations, for example the differences in complexity of addition and multiplication. This approximation is adequate for hardware floating point arithmetic. For more complicated computations one can easily expand all the definitions using weights of operations (as in [17]).

3 Delayed simplifications and variable ordering

We start this section with the observation that the earlier-described algorithm to construct MCRs solves one of the problems mentioned in Section 2.1. Namely, it solves the problem of constructing CR representations in one parse of the input expression. However this algorithm is still susceptible to an exploding number of intermediate expressions. We address that problem in this section. It can be seen from examples that other approaches to improvement of the MCR scheme are possible. If straightforward MCR construction is used, there are many CRs with common subchains in the resulting MCR-scheme (as in equation (3)). Most of these common subchains can be found even without any additional search (as in [11]), because after some transformations we know in advance that resulting CRs have common subchains. However, additional overheads to handle all cases like this are unavoidable. Here we consider an approach which leads to roughly the same complexity of the resulting scheme, but often reduces the time of MCR construction. This approach also allows us to obtain easily an approximate solution of the variable ordering problem.

3.1 Delayed MCR simplifications

The notion of delayed computations (lazy computations, mixed evaluation) is widely used in the theory and practice of computer programming. We first demonstrate an idea of delayed simplification on the example of the Horner scheme for bivariate polynomials.

Consider a polynomial $P(x, y)$, $\deg(P, x) = s$, $\deg(P, y) = q$, represented by $(s + 1) \cdot (q + 1)$ matrix of coefficients. It's easy to write a 2-dimensional Horner scheme to compute values $P(x, y)$ over $n_x n_y$ points $x = x_1, \ldots, x_{n_x}$, $y = y_1, \ldots, y_{n_y}$. If x corresponds to the inner loop, the complexity of these computations will be $c_1 = n_x n_y 2s + n_y(s + 1)2q$. What if this polynomial is of the form $P(x, y) = f(x) * g(y)$? We can expand it and again use the matrix of coefficients of the expanded polynomial, and obtain the scheme of the complexity c_1.

Another possible solution is to not expand, but use two linear arrays to represent these polynomials, which gives a scheme like:

```
for  y = y_1, ..., y_{n_y}  do
   a := g(y);     {Horner scheme for g(y)}
   for  x = x_1, ..., x_{n_x}  do
     store  a * f(x);     {Horner scheme for f(x)}
od od
```

with complexity $c_2 = n_x n_y 2s + n_y 2q + n_x n_y$ (the last term occurs because we keep performing multiplication by a in the innermost loop).

Alternatively we can delay expansion to the point in the computational scheme where the value of $g(y)$ is already computed and still represent polynomials $f(x)$ and $g(y)$ by linear arrays:

```
for  y = y_1, ..., y_{n_y}  do
  a := g(y);     {Horner scheme for g(y)}
  multiply all coefficients of f(x) by  a;
  for  x = x_1, ..., x_{n_x}  do
    store  f(x);     {Horner scheme for f(x)}
od od
```

This scheme with delayed expansion has complexity:

$$c_3 = n_x n_y 2s + n_y 2q + n_y(s + 1).$$

The last term here is the complexity of delayed expansion. It is easy to see that $c_3 < c_2$ when $s + 1 < n_x$ and $c_3 < c_1$ when $1 < (2q - 1)s$, i.e. delayed expansion gives the best solutions for all polynomials of the form $f(x)g(y)$ of degree greater than 1 if the corresponding computational task is "reasonable". This last scheme is a compromise between the first two: it keeps the advantage of expanded representation for the inner loop, and reduces the number of operations in the outer loop.

MCR representation of polynomials is analogous to the expanded form. A similar approach to the above gives an MCR-scheme with delayed simplifications (expansions). Looking closely into CR-simplification rules [11] we find a small number of rules which lead to the replication of constant subexpressions. In the linear CR case, these rules have to be applied unconditionally to decrease the complexity of the linear CR-scheme. In the MCR case these rules turn into rules which can replicate arbitrary (not necessarily constant) expressions. Unconditional application of these rules decreases the complexity of the innermost loop but can lead to an increase in the complexity of outer loops (as we have seen in examples). Let us consider all rules of the form

$$\Phi \odot \Psi \to \Delta, \ (u = \mathtt{mvar}(\Phi), v = \mathtt{mvar}(\Psi), v \prec u),$$

which lead to replication of MCR expression Ψ:
1. $\odot = *$
a) Φ is pure-sum in u, Ψ is pure-sum in v;
b) Φ is pure-sum, Ψ is an arbitrary MCR expression;
c) Φ is a trigonometric CR in u, Ψ is an arbitrary MCR expression;
2. $\odot = \hat{}$
a) Φ is pure-product in u, Ψ is pure-sum in v;
b) Φ is pure-product, Ψ is an arbitrary MCR expression;

During MCR construction we have to check all of these rules before simplification, and as the result of the check we can possibly delay simplification to the interpretation time. The decision about the delay is made on the base of information from higher levels of an expression to be simplified. It allows possible future simplification in u. This means that

simplification has to be delayed only if any future simplifications (say addition of pure-sum in u MCRs) are impossible. Since the procedure of MCR construction is recursive, we can easily keep information about higher levels of the expression to the moment of making concrete decision. This strategy allows us to keep innermost loop as simple as possible and prevent replication of MCR expressions in variables with least precedence.

Technically it means that as the result of simplification of, for example, the expression
$\{_u \varphi_0, +, \varphi_1, +, \ldots + , \varphi_k\}_u * \Psi$
instead of
$\{_u \varphi_0\Psi, +, \varphi_1\Psi, +, \ldots + , \varphi_k\Psi\}_u$
we can have
$\{_u \varphi_0, +, \varphi_1, +, \ldots + , \varphi_k\}_u \odot_v \Psi$
where $\odot$ denotes delayed multiplication and v is the *delay* variable. The MCR construction time is reduced in this case, because the delay stops recursive calls of simplifications of MCRs in v. During interpretation, as soon as we enter into the loop in variable v and replace all CRs in v by their values, we can perform simplifications delayed by variable v. In order to obtain an interpreter for the delayed MCR scheme it is enough to add only one option to the definition of function $\texttt{Value}_u$:

$\texttt{Simplify(}\ P_u(\texttt{Value}_u(\Phi^{(1)}), ..., \texttt{Value}_u(\Phi^{(s)}))$,

if $\Phi = P_u(\Phi^{(1)}, ..., \Phi^{(s)})$ and P_u is a delayed operation;

The procedure `Simplify` used here is not the same as is used on the phase of MCR construction. It is a hardcoded procedure which performs simplification over MCR with numerical components and numerical constants. In the case of code generation delayed simplifications are taken into account by inserting actual code to perform this simplification into generated procedure.

Let's consider the expression from Example 1. If we use delayed MCR construction rules, we obtain the MCR expression with delayed by j multiplication:

$$\{_i -1, +, 1, +, 2\}_i \odot_j \{_j 2, *, \frac{1}{16}, *, \frac{1}{16}, *, 64\}_j .$$

The result of code generation will look, in this case, like:

```
cc0:=2; cc1:=1/16; cc2:=1/16; cc3:=64; # CR in j
cc4:=-1; cc5:=1; cc6:=2;               # CR in i
for _cc1 from 0 to nn1 do
  cc7:=cc4*cc0; cc8:=cc5*cc0;          # delayed CR-
  cc9:=cc6*cc0;                        # multiplication
  for _cc2 from 0 to nn2 do
    aa1[_cc1,_cc2] := cc7;
    cc7:=cc7+cc8; cc8:=cc8+cc9         # shift in i
  od;
  cc0 := cc0*cc1; cc1 := cc1*cc2;
  cc2 := cc2*cc3                       # shift in j
od
```

It is easy to see that the outer loop becomes simpler than before. At the same time the inner loop remains of the same complexity as before.

3.2 Variable ordering

It was already shown in an earlier example that the complexity of the MCR scheme depends greatly on variable order. The problem of choosing the "optimal" order in the general case needs exhaustive search. It also can turn out that different variable orders are necessary for different subexpressions of the given expression F in order to reach the optimal scheme. It was shown in [17] for the 2-dimensional case, that some heuristics and reconstruction of the general interpreting scheme (splitting loop `(*)`) allow keeping the "optimal" order for different subexpressions. However, in the general case this approach leads to rather complicated memory management. Even in the 2-dimensional case, this approach destroys some of the CR computational features (e.g., some operations and function calls of $\exp, \sin$ etc. which could be removed with a straightforward scheme, remain in the innermost loop). We propose an approximate solution of the ordering problem which is easy to implement to making on the fly ordering decisions and keeps the general MCR interpreting scheme (`(**)`) unchanged.

We start with a simple polynomial example. Let F from (1) be a polynomial with $l_j = \deg_{i_j} F << n_j$ and we use a Horner scheme to compute values of F. Under these conditions an easy solution exists. We sort degrees l_j and reorder variables $i_1, \ldots, i_m$ into $i_{k_1}, \ldots, i_{k_m}$ such that $l_{k_1} \geq l_{k_2} \geq \ldots \geq l_{k_m}$. This gives an m-dimensional Horner scheme of the smallest complexity.

In the case of a delayed MCR scheme, the values $l_u(\Phi)$ from vector $\mathbf{l} = (l_1, \ldots, l_m)$ defined in section 2.6:
– do not depend on the order of variables i_j from computational task (1),
– offer good approximation to values of $CI_u(\Phi)$ (in fact $l_u(\Phi)$ is the lower bound for $CI_u(\Phi)$ which does not depend on the order of variables).
We can use these values for an approximate solution to the ordering problem. The solution is determined using three steps:
1. Construct an MCR expression with delays w.r.t. an arbitrary order (and compute vector $\mathbf{l}$ at the same time).
2. Sort values l_j and change the order of variables i_j so that $l_{k_1} \geq l_{k_2} \geq \ldots \geq l_{k_m}$.
3. Reconstruct the MCR expression w.r.t. the new variable order.

It turns out that due to the delayed MCR simplification scheme, the third step does not involve reconstruction from scratch of MCRs w.r.t. the new order. It may require only a few additional simplifications. Return to Example 2. After step 2 in this example we have an MCR expression, which does not depend on the order of variables i, j. If the order is changed we can start reconstruction from this step. Technically it is easy to implement, especially in Maple due to *remember tables*. We just once more call the procedure to construct MCRs w.r.t. the new order and the remember table mechanism prevents reconstruction from scratch. In our C and Java implementations we use a weaker, but similar approach. Each node of the expression F parse tree after the first stage of the MCR construction contains a pointer to MCR expression which corresponds to the sub-tree with root in this node.

4 Implementation

4.1 Maple MCR package

Our Maple implementation is a general purpose library level package which provides the MCR construction procedure, the set of MCR interpreting procedures and the MCR code generator. The implementation allows user-defined functions in the input expression (list of expressions), an arbitrary computational domain etc. The MCR construction procedure together with all parameters of an initial computational task, accepts optional arguments which turn on

(off) the delays-based scheme and/or the choice of an appropriate variable ordering scheme (as described earlier). Whenever some of the parameters of the computational task (initial values, steps) involve symbols, the code generator assumes that the generated procedure will be used several times. The generated procedure is correspondingly parametrized. The code generator calls standard Maple optimize facilities to handle common subexpressions.

Example 3. For the computational task $\sin(x)\exp(y)+f(2x)$, $x = a, a+I, \ldots$; $y = b, b+I, \ldots$ (f is a user defined or undefined function, $I = \sqrt{-1}$) we obtain the following procedure:

```
proc (aa1, nn1, nn2, b, a)
 local cc0, cc1, cc2, cc3, cc4, cc5, cc6, cc7, cc8,
       cc9, cc10, _cc1, _cc2, t1, t2, t3, t4;
 cc2 := 2*a; cc3 := 2*I; t1 := sin(a); t2 := exp(b);
 cc6 := t1*t2; t3 := sinh(1); cc7 := I*t3; t4 := cos(a);
 cc8 := t4*t2; cc9 := cosh(1); cc5 := exp(I);
 for _cc1 from 0 to nn1 do
   cc1 := f(cc2); cc4 := cc6;
   for _cc2 from 0 to nn2 do cc0 := cc1+cc4;
       aa1[_cc1,_cc2] := cc0; cc4 := cc4*cc5 od;
   cc2 := cc2+cc3; cc10 := cc6*cc9+cc7*cc8;
   cc8 := cc8*cc9-cc7*cc6; cc6 := cc10
 od
end
```

One can use the Maple C function in order to generate C code based on the MCR scheme.

Consider the following computational tasks:

1. $\sin(x+y), x = -1..1, y = -1..1$
2. $(1.3)^x \sin(y), x = -1..2\pi, y = 0..\pi$
3. $x\exp(-x^2-y^2), x = -2..2, y = -2..2$
4. $1.3^{1.2x-1}\cos(1.5x)\sin(1.5y), x = -2..2, y = -2..2$
5. $(1.3)^x \sin(uy), x = -1..2\pi, y = 0..\pi, u = 1..8$
6. $[x, y, (1.3)^x \sin(uy)], x = 1..3, y = 1..4, u = 1..2$
7. $\sin(\cos(\sin(\exp(x^{15}-y^{15}+x^3-x^{15}+y^{15}-x^3)-1)(x^5-5y))), x = 0..1, y = 0..1$

For all two-dimensional examples we used grid $[100, 100]$ and for all three-dimensional examples we used grid $[50, 50, 20]$.

Table 1 represents the timing results for computational tasks 1–7 using Maple V.4 (R4), Maple V.5 (R5) and MCR (construction, code generation and running time) on DEC Alpha 1000/800s with 256 Mb RAM. When `Digits` was set to 10, the code was run under `evalhf`. For timing comparison purposes, only the computational parts of `plot3d` and `animate3d` were run. After recent improvements in Maple `plot3d`, `animate3d` and hardware float arrays, the speed gained by an MCR-generated procedure under `evalhf` is not very impressive. But there is still room for progress, since the current slowdown of the MCR-scheme under `evalhf` is caused by `evalhf` architectural features. Our C implementation (see next subsection) using an internal representation very similar to Maple's internal representation, shows a factor 4-8 speedup in comparison with the same MCR scheme in Maple under `evalhf`. However, when we switch to `evalf` (increasing the accuracy of computation to 30 digits), the MCR generated scheme is much faster then Maple's `evalf`.

	D	R4	R5	MCR constr.	MCR codeg.	MCR run
1	10	0.946	0.378	0.033	0.033	0.217
	30	8.269	9.698	0.049	0.032	6.82
2	10	1.087	0.532	0.06	0.02	0.107
	30	15.317	16.708	0.076	0.021	1.93
3	10	1.074	0.428	0.041	0.024	0.193
	30	35.739	37.756	0.061	0.025	5.02
4	10	1.452	0.646	0.086	0.033	0.229
	30	27.135	26.407	0.124	0.032	7.302
5	10	7.815	7.497	0.076	0.039	0.559
	30	360.146	336.587	0.097	0.040	18.043
6	10	20.482	20.356	0.107	0.073	1.621
	30	478.558	485.819	0.141	0.073	35.599
7	10	0.907	0.396	0.004	0.009	0.059
	30	10.185	10.248	0.016	0.009	0.748

Table 1: Maple timings in seconds ($D \equiv$ `Digits` here).

4.2 Standalone MCR engines

In this section, we describe briefly two standalone implementations of the MCR technique in C and Java and demonstrate their practical use.

4.2.1 C implementation

A standalone C MCR engine, built on the top of small symbolic kernel, is implemented in ANSI C. This engine can be linked to any application (under UNIX, MSDOS or Windows) as the library or can be used as an executable program accepting computational task from standard input. This engine is able to handle multivariate computational tasks and uses a complete set of MCR simplifications including a delayed MCR construction technique.

Table 2 presents the timings of the examples from the previous subsection on the same machine (we repeat Maple timings here for comparison):

	Maple (constr. + codegen)	Maple (evalhf)	MCR engine (constr.)	MCR engine (running)
1	0.066	0.217	< 0.01	0.05
2	0.08	0.107	< 0.01	0.017
3	0.065	0.193	< 0.01	0.034
4	0.119	0.229	< 0.01	0.034
5	0.115	0.559	< 0.01	0.200
6	0.180	1.621	< 0.01	0.434
7	0.013	0.059	0.05	0.017

Table 2: Maple and C timings in seconds.

4.2.2 Java implementation

Our Java implementation is a package of classes which can be easily included in any Java project. It provides basic methods for MCR construction and interpretation. Input expressions can be either in the usual infix form, or in OpenMath SGML/XML encoding [19]. The evaluation domain can be floating point or Java BigIntegers.

In order to show the ease of use of the JMCR engine we used the SurfacePlotter [20] available with sources on the Gamelan web site and replaced the computational part of this plotter. The timing comparison of the native SurfacePlotter interpreter and the JMCR engine is shown in the Table 3. These results were obtained on a Pentium 200 with 64 Mb RAM (we include Maple timings on the same machine for scaling).

	Maple constr.+ codegen	Maple (evalhf)	Surface plotter	JMCR engine (constr.)	JMCR engine (run)
1	0.041	0.120	3.856	0.100	0.030
2	0.050	0.131	4.016	0.100	0.020
3	0.040	0.070	4.326	0.070	0.040
4	0.070	0.110	3.455	0.050	0.130
5	0.100	0.300	—	0.040	0.341
6	0.111	0.320	—	0.070	0.510
7	0.020	0.030	3.875	0.090	0.040

Table 3: Java timings in seconds.

The live demo of the JMCR engine is available on `http://scg1.uwaterloo.ca/JMCR.html`. It is possible to run and compare times for different computational tasks with the SurfacePlotter interpreter and JMCR engine. As well, the SurfacePlotter with its computational part replaced with JMCR can be run to compare the plotting quality.

There is a special-purpose 2-dimensional implementation of the MCR technique (MPCR-server [17]), which combines some variable ordering heuristics with very careful implementation of the interpretation scheme. Observe that this MPCR-server can not handle 3-dimensional examples, and it is valid only over the real number computational domain. Moreover, it does not support proper simplifications: for computational task 7 (section 4.1) the MPCR-server generates 103 lines of C code, when the expression is equivalent to sin(1). Heuristic procedures in [17, 15] are based on a "weak" definition of the degree of polynomial, which lead to such inadequate results. However, some ideas from this implementation can be combined with the delayed simplification technique, which may lead to further speedup of CR-based computations.

Acknowledgments

Authors would like to thank Chris Howlett (Web Pearls Inc.), Ha Quang Le (University of Waterloo) and Alastair Rough (ATI Technologies Inc.) for their help during preparation of this paper.

References

[1] Knuth D.E. The art of computer programming. V.2. Seminumerical Algorithms. First edition. Addison-Wesley, 1969.

[2] Knuth D.E. The art of computer programming. V.2. Seminumerical Algorithms. Second edition. Addison-Wesley, 1981.

[3] Dobkin D., Lipton R. Addition chain method for the evaluation of specific polynomials. SIAM J, Computing **9**, 1980, pp.121–125.

[4] Zima E.V. Automatic Construction of Systems of Recurrence Relations. USSR Comput. Maths. Math. Phys., Vol.24, N 6, 1984, pp. 193–197.

[5] Zima E.V. System of Recurrence Relations and Loops Optimization. PhD thesis, Department of Computational Mathematics and Cybernetics, Moscow State University, 1985.

[6] Davenport J., Siret Y., Tournier E. Calcul formel, Masson, 1987.

[7] Fateman R. Honest Plotting, Global Extrema, and Interval Arithmetic. Proc. ISSAC'92, Berkley, USA, July 1992, ACM Press, pp. 216-223.

[8] Zima E.V. Recurrent relations and speed-up of computations using computer algebra systems. Proc. DISCO'92, Bath, U.K., 1993, LNCS 721, Springer-Verlag, pp.152–161.

[9] Zima E.V. Numeric code optimization in computer algebra systems and recurrent relations technique. Proc. ISSAC'93, Kiev, Ukraine, 1993, ACM Press, pp. 242-249.

[10] Bachmann O., Wang P.S., Zima E.V. Chains of Recurrences – a method to expedite the evaluation of closed-form functions. Proc. ISSAC'94, Oxford, UK, July 1994, ACM Press, pp. 242-249.

[11] Zima E.V. Simplification and Optimization Transformations of Chains of Recurrences. Proc. of ISSAC'95, Montreal, Canada, ACM Press, pp. 42–50.

[12] Avitzur R., Bachmann O., Kajler N. From Honest to Intelligent Plotting. in the Proceedings of ISSAC'95, Montreal, Canada, ACM Press, 32–41.

[13] Casavant T., Vadivelu K., Zima E. Mapping Techniques for Parallel Evaluation of Chains of Recurrences. *1996 International Parallel Processing Symposium*, April 1996, pp. 620–624.

[14] Bachmann O. Chains of Recurrences for Functions of Two Variables and their Application to Surface Plotting. In N.Kajler, ed., *Human Interaction for Symbolic Computation*. Springer-Verlag, 1996.

[15] Bachmann O. Chains of recurrences. PhD thesis, Kent State Univ., Kent, OH - 44240, USA, December 1996.

[16] Viazmitinov G.L., Zima E.V. A Portable Interpreter of Multidimensional Chains of Recurrences. Programmirovanie N 1, 1997, pp. 4–11.

[17] Bachmann O. MPCR: An Efficient and Flexible Chains of Recurrences Server. SIGSAM Bulletin, vol. 31, N 1, March 1997, pp. 15–21.

[18] Zima E.V. Safe Numerical Computations with Chains of Recurrences. Programmirovanie N 3, 1997, pp. 36–42.

[19] `http://www.openmath.org`

[20] `http://dsaka20.kushiro-ct.ac.jp/~yanto/java/surface/`