Symbolic Evaluation of Chains of Recurrences for Loop Optimization

Robert A. van Engelen
Dept. of Computer Science
Florida State University
Tallahassee, Florida 32306
e-mail: `engelen@cs.fsu.edu`

Keywords: Symbolic Evaluation, Recurrence Relations, Loop Optimization, Normal Forms, Intermediate Program Representation, Program Verification.

Abstract

This paper presents a novel method for loop optimization that exploits symbolic evaluation of chains of recurrences (CRs). The method generalizes loop induction expression recognition, loop parallelization by induction variable substitution, loop strength reduction, and loop invariant expression elimination (code motion). Symbolic differencing of loops has been extensively studied by Haghighat for these type of transformations. The differencing method detects generalized loop induction variables and approximates the corresponding closed-form functions by polynomials bounded to a predetermined maximum order. We will show that compared to differencing methods, symbolic evaluation with CRs is safe, more powerful, and simpler to implement. We prove that CRs are unique normal forms, which is crucial for the use of CRs as intermediate program representations of recurrences. Based on the algorithms developed by Bachmann, Zima, and Wang, we develop a CR-based algorithm for loop analysis. We present a set of symbolic transformations that define a partial mapping of CRs to closed-form functions. We also demonstrate the potential of the method for analyzing power series computations in loops. Finally, we demonstrate that the CR intermediate program representation provides a very effective means for non-linear loop dependence testing.

1 Introduction

In this paper we develop a new method for loop analysis and optimization based on chains of recurrences (CRs) [6]. The CR-based loop analysis method can be applied in the areas of

- **Program optimization.**
  The CR framework is a firm algebraic basis for loop analysis and optimization that generalizes ad-hoc compiler optimization techniques related to loop induction variable recognition [1, 2, 14, 21]. Furthermore, CR-based loop analysis and optimization is

more powerful compared to the automatic differencing method of Haghighat [12], currently the most powerful method known. The CR loop optimization method is simple to implement in compilers and in code generators of symbolic computing environments.

- **Program equivalence determination.**
  The CR-based analysis algorithm constructs an SSA-like intermediate program representation [8] in which CR-expressions represent recurrences and closed-form functions of loop counter variables. Because CRs are unique normal forms, the CR-based intermediate program representation can be used in systems that need to determine the semantic equivalence of two syntactically distinct programs.

- **Program analysis and verification.**
  The correctness of certain numerical algorithms can be verified by showing that (parts of) an algorithm correspond to specific closed-form functions of the input parameters, where the closed-form functions are determined by the CR loop analysis algorithm.

In this paper we focus on the application of the CR-based loop analysis in loop optimization. We recently added CR-based loop optimization to the CTADEL code-generation system [17]. The CTADEL system is a symbolic environment and code generator for weather forecast models. The system generates efficient vector and parallel codes from higher-level PDE specifications of forecast models. We have shown that the generated codes outperform handwritten codes [16]. The PDEs are solved on regular grids which naturally leads to numerical codes containing many loops.

1.1 Related Work

For the methods presented in this paper we adopt the chains of recurrences formalism. *Chains of recurrences* (CRs) are developed by Bachmann, Zima, and Wang [6] by reformulation of Zima's *systems of recurrence relations* (SRRs) [19]. SSRs expedite the evaluation of polynomials, exponentials, factorials, and trigonometric functions on regular grids. This reformulation enabled the definition of an algebra for the construction and simplification of the recurrence relations, which allowed the automation of the construction process by symbolic manipulation, for example using a computer algebra system. Elementary expressions can be symbolically transformed to their mathematically equivalent CRs [4].

CRs are an effective method to accelerate the evaluation of closed-form functions on regular grids by reusing values calculated at previous points. For example, loops can be generated for the efficient tabulation of closed-form functions on grids. Applications range from plotting curves [5], computing finite sums and products, calculating integrals, and solving differential equations [4]. Performance results show that the actual evaluation of the recurrence relations in generated Fortran code is substantially faster than evaluating the original function at each grid point separately [6].

The methods presented in this paper are closely related to Haghighat's symbolic differencing method [12]. The differencing method recognizes generalized induction expressions [3, 9, 10, 11, 18] that form polynomial and geometric progressions described by the characteristic function χ defined as $\chi(n) = \varphi(n) + ar^n$, where n is an integer variable identifying the loop iteration number, φ is an integer-valued polynomial of finite degree in n whose coefficients are loop-invariant expressions, and a and r are loop-invariant expressions. The method is currently the most powerful known method for induction variable recognition and has been adopted by the Parafrase-2 optimizing compiler [15]. Performance results [12] on the Perfect Benchmarks$^{(R)}$ demonstrate the performance impact on numerical algorithms. However, the differencing method is problematic, because

- **The method can result in incorrectly transformed programs.**
  The differencing method recognizes polynomials and geometric series by the tabulation of symbolic difference tables. Closed-form expressions of induction variables are derived by symbolic interpolation. The Parafrase-2 compiler, for example, adopts a user-specified maximum degree m for interpolating polynomials. A loop is executed at most $m + 2$ time iterations of the loop symbolically to recognize induction expressions of degree m. The claim that "the more iterations executed, the higher degree interpolating polynomials can be discovered" [12] is error prone, because the differencing method truncates high-order induction expressions to polynomials of lower order when the maximum degree m is set too low (see also Section 3.1.3). On the other hand, setting m too high results in expensive analysis costs since the tabulation requires $O(m^2)$ time. The problem is that there is no guarantee that m is high enough for an accurate analysis and the responsibility is left to the user to define an appropriate value for m.

- **Analysis is restricted to expressions that are differentiable.**
  Consider for example the expression `FLOOR((i*i+5)/2.0)` which is not differentiable with respect to a loop counter variable `i`. However, the argument of `FLOOR` is a differentiable polynomial in `i`. Differencing methods cannot recognize differentiable subexpressions. In contrast, the CR-based analysis method includes subexpressions in the analysis.

- **Analysis is limited to polynomials and geometric series.**
  The differencing method detects generalized loop inductions variables whose progressions form polynomials and geometric series. The CR-based analysis method is more general, because polynomials, exponentials, factorials, and compositions of these are recognized by the method. Although compositions are not very common in numerical applications, expressions that are a mix are more common and can be optimized by the CR-based analysis method as well.

- **The method uses abstract interpretation.**
  Abstract interpretation is a costly technique and requires substantial programming to implement in an optimizing compiler. In contrast, the CR-based analysis method is a static analysis method that operates directly on an intermediate program representation.

This paper is organized as follows. In Section 2 we introduce the basic concepts of CRs, we prove that CRs are normal forms, and we extend the framework with a mapping of CRs to closed-form functions. Section 3 presents the CR-based loop analysis and optimization methods. A loop dependence testing method that exploits CRs is discussed in Section 4. Finally, in Section 5 we summarize our results.

2 Chains of Recurrences

Any closed-form function f (not only polynomials) evaluated on $x = x_0 + i\,h,\ i = 0,\ldots,n$ has a CR if f can be rewritten as a mathematical equivalent SRR $f_0(i), f_1(i), \ldots, f_k(i)$, where the functions $f_j(i)$ for $j = 0, \ldots, k-1$ are *linear recurrences* of the form

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

with $\odot \in \{+, *\}$ and i the index to which the recurrence is evaluated. The function f_k is either a constant value (an expression independent of i) or a similar recurrence system. For the system it holds that $f(x_0 + i\,h) = f_0(i)$.

A linear recurrence Eq. (1) is called a *Basic Recurrence* (BR), denoted by

$$f_j(i) = \{\phi_j, \odot_{j+1}, f_{j+1}\}_i\ , \tag{2}$$

where i denotes the recurrence index. The BR notation allows a system defined by Eq. (1) to be written as a single recursive expression

$$\Phi_i = \{\phi_0, \odot_1, \{\phi_1, \odot_2, \cdots, \{\phi_{k-1}, \odot_k, f_k\}_i\}_i\}_i\ . \tag{3}$$

When flattened to a single tuple

$$\Phi_i = \{\phi_0, \odot_1, \phi_1, \odot_2, \cdots, \odot_k, f_k\}_i \tag{4}$$

it is called a *Chain of Recurrences* (CR) with $k = L(\Phi_i)$ the length of the CR.

A CR is called *simple* if f_k is a constant. It is called a *pure sum* or *polynomial* CR if $\odot_j = +$ for $1 \le j \le k$, and it is called a *pure-product* or *exponential* CR if $\odot_j = *$ for $1 \le j \le k$. A simple polynomial CR of length k represents a k-order polynomial in variable x with $x = x_0 + i\,h$.

2.1 CR Construction

The CR representation Eq. (4) enables the development of an algebra, such that the process of construction and simplification of CRs for arbitrary closed-form functions can be automated within a symbolic computing environment [4]. The algebra is defined by the rewrite rules shown in Fig. 1 taken from [6, 4], where the variables E, ϕ, ψ, f, and g in the rules match arbitrary expressions and i is an index variable. The test $i \notin FV[E]$ is true when expression E does not contain variable i, where the *set of free variables* $FV[E]$ of expression E is recursively defined on the structure of E by

$$\begin{array}{lcll} FV[\{\phi_0, \odot_1, \phi_1, \ldots, \phi_{k-1}, \odot_k, f_k\}_i] & = & \left(\bigcup_{j=0}^{k-1} FV[\phi_j]\right) \cup FV[f_k] \cup \{i\} & \\ FV[f(E_1, \ldots, E_k)] & = & \bigcup_{j=1}^{k} FV[E_j] & \text{for } k\text{-ary function symbol } f \\ FV[v_E] & = & \{v\} \cup FV[E] & \text{for variable } v \text{ indexed by } E \\ FV[v] & = & \{v\} & \text{for variable } v \\ FV[c] & = & \emptyset & \text{for constant } c \end{array}$$

Rule	LHS		RHS	
1	$E + \{\phi_0, +, f_1\}_i$	$\Rightarrow$	$\{E + \phi_0, +, f_1\}_i$	when $i \notin FV[E]$
2	$E * \{\phi_0, +, f_1\}_i$	$\Rightarrow$	$\{E * \phi_0, +, E * f_1\}_i$	when $i \notin FV[E]$
3	$E * \{\phi_0, *, f_1\}_i$	$\Rightarrow$	$\{E * \phi_0, *, f_1\}_i$	when $i \notin FV[E]$
4	$E^{\{\phi_0, +, f_1\}_i}$	$\Rightarrow$	$\{E^{\phi_0}, *, E^{f_1}\}_i$	when $i \notin FV[E]$
5	$\{\phi_0, *, f_1\}_i^E$	$\Rightarrow$	$\{\phi_0^E, *, f_1^E\}_i$	when $i \notin FV[E]$
6	$\{\phi_0, +, f_1\}_i + \{\psi_0, +, g_1\}_i$	$\Rightarrow$	$\{\phi_0 + \psi_0, +, f_1 + g_1\}_i$	
7	$\{\phi_0, +, f_1\}_i * \{\psi_0, +, g_1\}_i$	$\Rightarrow$	$\{\phi_0\psi_0, +, \{\phi_0, +, f_1\}_i * g_1 + \{\psi_0, +, g_1\}_i * f_1 + f_1 * g_1\}_i$	
8	$\{\phi_0, *, f_1\}_i * \{\psi_0, *, g_1\}_i$	$\Rightarrow$	$\{\phi_0\, \psi_0, *, f_1\, g_1\}_i$	
9	$\{\phi_0, *, f_1\}_i^{\{\psi_0, +, g_1\}_i}$	$\Rightarrow$	$\{{\phi_0}^{\psi_0}, *, \{\phi_0, *, f_1\}_i^{g_1} * f_1^{\{\psi_0, +, g_1\}_i} * f_1^{g_1}\}_i$	
10	$\{\phi_0, +, f_1\}_i!$	$\Rightarrow$	$\begin{cases} \{\phi_0!, *, \left(\prod_{j=1}^{f_1} \{\phi_0 + j, +, f_1\}_i\right)\}_i & \text{if } f_1 \geq 0 \\ \{\phi_0!, *, \left(\prod_{j=1}^{\lvert f_1 \rvert} \{\phi_0 + j, +, f_1\}_i\right)^{-1}\}_i & \text{if } f_1 < 0 \end{cases}$	
11	$\log\{\phi_0, *, f_1\}_i$	$\Rightarrow$	$\{\log \phi_0, +, \log f_1\}_i$	

Figure 1: $\mathcal{CR}$

The term rewriting system (TRS) $\mathcal{CR}$ shown in Fig. 1 extended with rules for trigonometric functions forms $\mathcal{CR}^t$ shown in Fig. 2. The rules require complex arithmetic, see also [6]. In this paper we will use the "simpler" $\mathcal{CR}$ only.

CR construction proceeds by replacing every occurrence of variable x (where $x = x_0 + i\, h$) in an expression by its corresponding BR $\{x_0, +, h\}_i$ and applying the rules $\mathcal{CR}$ to exhaustion.

Theorem 2.1 *$\mathcal{CR}$ is complete.*

Proof. See Appendix A. □

Thus, any rule application order will result in a unique CR normal form corresponding to the form in Eq. (3) which can be rewritten more concisely as in Eq. (4).

The computational time of CR construction using the system $\mathcal{CR}^t$ is linear in the length of the expression, except for polynomials. For constructing CRs for polynomials, a simple rule of the form

$$x^n \Rightarrow x^{n-1} * x \qquad \text{for integer constant } n > 1$$

can be employed, where x is an arbitrary expression. However, this rule is very inefficient in practice. Bachmann [6] developed a fast algorithm for simplifying a n-th degree polynomial in $o(n^2)$ time.

Rule	LHS		RHS
12	$\sin\{\phi_0, +, f_1\}_i$	$\Rightarrow$	$\Im(e^{\imath\{\phi_0, +, f_1\}_i})$
13	$\cos\{\phi_0, +, f_1\}_i$	$\Rightarrow$	$\Re(e^{\imath\{\phi_0, +, f_1\}_i})$
14	$\tan\{\phi_0, +, f_1\}_i$	$\Rightarrow$	$\frac{\Im}{\Re}(e^{\imath\{\phi_0, +, f_1\}_i})$

Figure 2: $\mathcal{CR}^t = \mathcal{CR}$ + Rules 12–14 (where $\imath = \sqrt{-1}$)

$$
\begin{aligned}
&(x^2+x)-(x^2-x)\\
&\quad\Rightarrow\quad (x*x+x)+(-1)*(x*x+(-1)*x)\\
&\quad\Rightarrow\quad (\{0,+,h\}*\{0,+,h\}+\{0,+,h\})+(-1)*(\{0,+,h\}*\{0,+,h\}+(-1)*\{0,+,h\})\\
&\quad\stackrel{2}{\Rightarrow}\quad (\{0,+,h\}*\{0,+,h\}+\{0,+,h\})+(-1)*(\{0,+,h\}*\{0,+,h\}+\{0,+,-h\})\\
&\quad\stackrel{7}{\Rightarrow}\quad (\{0,+,\{0,+,h\}*h+\{0,+,h\}*h+h*h\}+\{0,+,h\})+(-1)*(\{0,+,\{0,+,h\}*h\\
&\qquad\qquad +\{0,+,h\}*h+h*h\}+\{0,+,-h\})\\
&\quad\stackrel{2}{\Rightarrow}\quad (\{0,+,\{0,+,h*h\}+\{0,+,h*h\}+h*h\}+\{0,+,h\})+(-1)*(\{0,+,\{0,+,h*h\}\\
&\qquad\qquad +\{0,+,h*h\}+h*h\}+\{0,+,-h\})\\
&\quad\stackrel{6}{\Rightarrow}\quad \{0,+,\{0,+,2*h*h\}+h*h\}+\{0,+,h\}+(-1)*\{0,+,\{0,+,2*h*h\}+h*h\}+\{0,+,-h\}\\
&\quad\stackrel{1}{\Rightarrow}\quad \{0,+,\{h*h,+,2*h*h\}\}+\{0,+,h\}+(-1)*\{0,+,\{h*h,+,2*h*h\}\}+\{0,+,-h\}\\
&\quad\stackrel{6}{\Rightarrow}\quad \{0,+,\{h*h,+,2*h*h\}+h\}+(-1)*\{0,+,\{h*h,+,2*h*h\}-h\}\\
&\quad\stackrel{1}{\Rightarrow}\quad \{0,+,\{h*h+h,+,2*h*h\}\}+\{0,+,\{-h*h+h,+,-2*h*h\}\}\\
&\quad\stackrel{6}{\Rightarrow}\quad \{0,+,\{2*h,+,0\}\}
\end{aligned}
$$

Figure 3: Example Polynomial CR Derivation (Each Step Involves Multiple Rewritings)

For some expressions, exhaustive application of the rules results in an expression with CRs as subexpressions. A special type of such mixed expressions are *CR-expressions* and these are still subject to optimization. Any closed-form function can be transformed into its equivalent CR-expression. For more details, the reader is referred to [6, 20].

2.2 Normalized CRs

$\mathcal{CR}$ (with associativity and commutativity of $+$ and $*$) as defined in [4] does not guarantee the derivation of normal forms for polynomials. Consider for example the mathematically identical polynomials $2x$ and $(x^2+x)-(x^2-x)$ with $x = i\,h$. The CR of polynomial $2x$ is $\{0,+,2*h\}_i$ after constructions and simplification using the system $\mathcal{CR}$, but the CR of $(x^2+x)-(x^2-x)$ is $\{0,+,2*h,+,0\}_i$ having a redundant trailing zero stride (see Fig. 3).

We call a CR $\{\phi_0,\odot_1,\phi_1,\ldots,\phi_{k-1},\odot_k,f_k\}_i$ *normalized* if $f_k \neq 0$ when $\odot_k = +$, and $f_k \neq 1$ when $\odot_k = *$, and $\phi_j \neq 0$ when $\odot_{j+1} = *$ for all $0 \leq j < k$. A normalized CR has no trailing zero stride (for $+$) or unit stride (for $*$) and (embedded) product CRs have no initial zero values. Fig. 4 depicts $\mathcal{CR}^+$, the extension of $\mathcal{CR}$ with three cancelation rules for deriving normalized CRs and arithmetic simplification rules. For example, the CR $\{2,+,0,*,1\}_i$ is normalized to constant 2 using rewrite system $\mathcal{CR}^+$ on its nested representation $= \{2,+,\{0,*,1\}_i\}_i$.

The system $\mathcal{CR}^+$ derives normal forms for polynomials.

Theorem 2.2 $\mathcal{CR}^+$ *with associativity, commutativity, and arithmetic rules is complete for univariate CRs.*

Proof. See Appendix B. □

Rule	LHS		RHS
15	$\{\phi_0, +, 0\}_i$	$\Rightarrow$	ϕ_0
16	$\{\phi_0, *, 1\}_i$	$\Rightarrow$	ϕ_0
17	$\{0, *, f_1\}_i$	$\Rightarrow$	0
18	$0 + E$	$\Rightarrow$	E
19	$0 * E$	$\Rightarrow$	0
20	$1 * E$	$\Rightarrow$	E
21	0^E	$\Rightarrow$	0
22	1^E	$\Rightarrow$	1
23	E^0	$\Rightarrow$	1
24	$\log 1$	$\Rightarrow$	0

Figure 4: $\mathcal{CR}^+ = \mathcal{CR}$ + Rules 15–24

The CR normalization of polynomials is closed under polynomial addition, subtraction, multiplication, and positive integer exponentiation after the addition of rules for scalar addition, subtraction, multiplication, and division to $\mathcal{CR}^+$. For example, $(x-2)(x+2)$ and $x^2 - 4$ have the same CR normal form.

2.3 Indeterminate CRs

A potential problem in the derivation of CRs is the meaning of 0^0. The problem occurs when deriving the CR of 0^i which is $\{0^0, *, 0\}_i$ using rule 4 of $\mathcal{CR}$. This suggests that we can define $0^0 = 1$ thereby obtaining $\{1, *, 0\}_i$, which represents a zero-valued function with a unit spike at $i = 0$. However, we can also define $0^0 = 0$, obtaining $\{0, *, 0\}_i = 0$.

We say that $\{\phi_0, \odot_1, \ldots, \phi_{k-1}, *, 0\}_i$ with $\phi_{k-1} \neq 0$ is an *indeterminate* CR, because the meaning of 0^0 is undefined. Indeterminate CRs are computable recurrences but in general have no corresponding closed-form function. An *indeterminate CR-expression* is a CR-expression that contains at least one indeterminate CR.

Indeterminate CR-expressions are very useful for the analysis of wraparound variables in loops. The indeterminate CR representation of wraparound variables enables the generalization of CRs as normal form representations of loop induction variables.

2.4 Multivariate CRs

Consider the CRs $\{\phi_0, +, f_1\}_i$ and $\{\psi_0, +, g_1\}_j$ exhibiting different index variables i and j. The CRs represent two functions evaluated on a two-dimensional grid in independent i and j directions, respectively. The sum of the functions forms a two-dimensional function, but adding the CRs using the formation rule 1 with commutativity (see Fig. 1) gives two possible multivariate CRs: $\{\{\phi_0, +, f_1\}_i + \psi_0, +, \{\phi_0, +, f_1\}_i + g_1\}_j$ where j is considered a dominant variable and $\{\{\psi_0, +, g_1\}_j + \phi_0, +, \{\psi_0, +, g_1\}_j + f_1\}_i$ where i is considered a dominant variable.

The system $\mathcal{CR}^+$ with associativity and commutativity is not complete under the formation of multivariate CRs. However, given a total order on variables, we can construct a unique multivariate CR normal form for a multidimensional function, where the total order determines the dominating variables. The multivariate CR representation is similar to the recursive representation of multivariate polynomials, where a multivariate polynomial is treated as a univariate polynomial in a dominant variable whose coefficients are polynomials in other variables.

2.5 Integer Divisible CRs

The integer divisibility test on (multivariate) polynomials is important to guarantee a canonical representation [12]. The divisibility test in [12] requires a separate algorithm. Based on the observation that the recursive representation of a multivariate polynomial lends itself well for an inductive proof of the divisibility of the polynomial by an integer, the integer divisibility algorithm relies on the recursive determination of the divisibility of $\Delta p = p(x_0 + (i+1)\,h) - p(x_0 + i\,h)$ for a given polynomial p, which effectively requires a tabulation of the Newton series for p.

In contrast, the CR-based analysis method does not require a separate test. Polynomial CRs are equivalent to Newton series whose evaluation is restricted to integer arguments (i.e. grid points) [6]. For integer divisibility testing of polynomials, rule 2 of Fig. 1 can be applied on the CR, where $E = 1/n$ for given integer n. If the CR coefficients after application of rule 2 are integer valued (the coefficients in turn can be polynomials which are tested using the same technique), then the (multivariate) polynomial is divisible by n. This method also works for CRs representing arbitrary closed-form functions, not only polynomials, by using rules 2 and 3 (see Fig. 1).

2.6 Symbolic Evaluation of CRs

A computer algebra system can be employed to derive the CR-expression of a closed-form function after which the function can be efficiently evaluated on a regular grid. For symbolic evaluation of a CR-expression, three symbolic functions are required. These functions are also used for the symbolic evaluation of CRs for the loop optimizations discussed in Section 3. Function $\mathcal{V}(\Phi_i)$ returns the current value of a CR-expression Φ_i and is defined as

$$\begin{array}{lcll} \mathcal{V}(\{\phi_0, \odot_1, \ldots, \odot_k, f_k\}_i) & = & \phi_0 & \\ \mathcal{V}(f(\Phi_i^1, \ldots, \Phi_i^k)) & = & f(\mathcal{V}(\Phi_i^1), \ldots, \mathcal{V}(\Phi_i^k)) & \text{for } k\text{-ary function } f \\ \mathcal{V}(v) & = & v & \text{for variable } v \\ \mathcal{V}(c) & = & c & \text{for constant } c \end{array} \qquad (5)$$

Function $\mathcal{F}(\Phi_i)$ shifts Φ_i "forward" and returns $\Phi(i+1)$. It is defined as

$$\begin{array}{lcll} \multicolumn{4}{l}{\mathcal{F}(\{\phi_0, \odot_1, \phi_1, \ldots, \phi_{k-1}, \odot_k, f_k\}_i)} \\ \multicolumn{4}{l}{\qquad = \{\phi_0 \odot_1 \phi_1, \odot_1, \phi_1 \odot_2 \phi_2, \odot_2, \ldots, \odot_{k-1}, \phi_{k-1} \odot_k \mathcal{V}(f_k), \odot_k, \mathcal{F}(f_k)\}_i} \\ \mathcal{F}(f(\Phi_i^1, \ldots, \Phi_i^k)) & = & f(\mathcal{F}(\Phi_i^1), \ldots, \mathcal{F}(\Phi_i^k)) & \text{for } k\text{-ary function } f \\ \mathcal{F}(v) & = & v & \text{for variable } v \\ \mathcal{F}(c) & = & c & \text{for constant } c \end{array} \qquad (6)$$

$$
\begin{aligned}
&cr_0 = \phi_0\\
&cr_1 = \phi_1\\
&\quad\vdots\\
&cr_k = f_k\\
&\texttt{DO i = } 0, n\\
&\quad f[\texttt{i}] = cr_0\\
&\quad cr_0 = cr_0 \odot_1 cr_1\\
&\quad cr_1 = cr_1 \odot_2 cr_2\\
&\qquad\vdots\\
&\quad cr_{k-1} = cr_{k-1} \odot_k cr_k\\
&\texttt{ENDDO}
\end{aligned}
$$

Figure 5: Template for the Numerical Evaluation of $\Phi_i = \{\phi_0, \odot_1, \phi_1, \ldots, \phi_{k-1}, \odot_k, f_k\}_i$

Function $\mathcal{B}(\Phi_i)$ shifts Φ_i "backward" and returns $\Phi(i-1)$. It is defined as

$$
\begin{array}{lcll}
\mathcal{B}(\{\phi_0, \odot_1, f_1\}) & = & \{\phi_0 \odot_1^{-1} \mathcal{V}(\mathcal{B}(f_1)), \odot_1, \mathcal{B}(f_1)\}_i & \\
\mathcal{B}(f(\Phi_i^1, \ldots, \Phi_i^k)) & = & f(\mathcal{B}(\Phi_i^1), \ldots, \mathcal{B}(\Phi_i^k)) & \text{for } k\text{-ary function } f \\
\mathcal{B}(v) & = & v & \text{for variable } v \\
\mathcal{B}(c) & = & c & \text{for constant } c
\end{array} \tag{7}
$$

where $\odot^{-1}$ is the inverse operator of $\odot$, i.e. $x \odot^{-1} y = x - y$ for $\odot = +$ and $x \odot^{-1} y = x/y$ for $\odot = *$. Note that $\mathcal{B}$ is defined for nested CR tuples $\{\phi_0, \odot, f_1\}_i$ where f_1 is a CR-expression as in Eq. (3). Function $\mathcal{B}$ is undefined for indeterminate CRs, because the application of $\mathcal{B}$ on an indeterminate CR results in an division by zero operation. Multivariate versions of the functions $\mathcal{V}$, $\mathcal{F}$, and $\mathcal{B}$ require slight modifications of the definitions above.

The proofs of correctness of functions $\mathcal{V}$ and $\mathcal{F}$ can be found in [4] (proof of correctness of algorithm CREval). The proof of correctness of function $\mathcal{B}$ can be found in Appendix C.

2.7 Numerical Evaluation of CRs

If a function has to be efficiently evaluated on a grid, code can be generated to numerically evaluate the CR of the function on a grid [6]. Fig. 5 shows a simple template of the code generated for a CR $\Phi_i = \{\phi_0, \odot_1, \phi_1, \ldots, \phi_{k-1}, \odot_k, f_k\}_i$. The template is a numerical version of the symbolic $\mathcal{V}$ and $\mathcal{F}$ functions. The code requires a total of $k+1$ scalar temporary variables cr_j, $j = 0, \ldots, k$, one for each CR coefficient ϕ_j. At most k additions or multiplications need to be performed at each loop iteration. Similar source code can be generated for the more general CR-expressions and multivariate CRs, see [4].

2.8 Numerical Stability

Bachmann [4] demonstrates that the rounding errors in a floating-point numerical evaluation of a polynomial CR accumulate at most linearly with a slope that is proportional to the length of the CR (degree of the polynomial). Rounding errors of exponential and trigonometric CRs

```
cr0 = 1.0
cr1 = EXP(h**2-4*h)
cr2 = EXP(2*h**2)
DO i = 0,n
  s[i] = s[i]+cr0
  cr0 = cr0*cr1
  cr1 = cr1*cr2
ENDDO
sum = 0.0
DO i = 0,n
  sum = sum+s[i]
ENDDO
```

(1)

```
sum = 0.0
cr0 = 1.0
cr1 = EXP(h**2-4*h)
cr2 = EXP(2*h**2)
DO i = 0,n
  sum = sum+cr0
  cr0 = cr0*cr1
  cr1 = cr1*cr2
ENDDO
```

(2)

```
sum = n*EXP(n*h**2-4*n*h
            +2*n**2*h**2)
```

(3)

Figure 6: Three Codes for the Numerical Evaluation of $\sum_{i=0}^{n} e^{(ih)^2-4ih}$

accumulate polynomially with a degree that is proportional to the length of the CR. The conclusion is that evaluations of polynomial CRs are numerically stable whereas evaluations of exponential and trigonometric CRs are not. Bachmann also developed algorithms for symbolically deriving upper bounds on the accumulated errors of CRs and algorithms for numerically evaluating CRs thereby ensuring that the user-specified error bounds are not exceeded. The algorithms are based on the derived error bounds and, wherever necessary, divide the evaluation intervals into smaller sections, until either the requested accuracy can be met or until the evaluation intervals become too small.

Haghighat's differencing method is restricted to the analysis of integer-valued induction expressions. Similarly, the CR loop analysis method presented in this paper can be restricted to avoid problems with numerical roundoff.

2.9 An Example

Consider the summation

$$\sum_{i=0}^{n} f(i\,h)\ , \quad f(x) = e^{x^2-4x}\ .$$

The CR of $f(x)$ is $\{1, *, e^{h^2-4h}, *, e^{2h^2}\}_i$ with $x = i\,h$ and $i = 0, \ldots, n$. Using the template Fig. 5 we can easily construct a program to evaluate this summation numerically as shown in Fig. 6 (1). In Fig. 6 (2) a slightly more efficient version of the code with respect to memory usage is shown.

Observe that in code Fig. 6 (2) the values of variable `sum` in the loop can be represented by another recurrence relation: $\{0, +, 1, *, e^{h^2-4h}, *, e^{2h^2}\}_i$. The closed-form function of `sum` at the i^{th} iteration of the loop is $s(i) = i\,e^{i\,h^2-4\,i\,h+2\,i^2\,h^2}$. The final value of `sum` at the end of the loop is $s(n) = n\,e^{n\,h^2-4\,n\,h+2\,n^2\,h^2}$. This means that an even more efficient code can be constructed that assigns $s(n)$ to `sum`, shown in Fig. 6 (3) [1]

[1]Set `sum` to $s(\max(n, 0))$ when taking into account that n can be negative resulting in a zero-trip loop.

Rule	LHS		RHS	
1	$\{\phi_0, +, f_1\}_i$	$\Rightarrow$	$\phi_0 + \{0, +, f_1\}_i$	when $\phi_0 \neq 0$
2	$\{\phi_0, *, f_1\}_i$	$\Rightarrow$	$\phi_0 * \{1, *, f_1\}_i$	when $\phi_0 \neq 1$
3	$\{0, +, -f_1\}_i$	$\Rightarrow$	$-\{0, +, f_1\}_i$	
4	$\{0, +, f_1 + g_1\}_i$	$\Rightarrow$	$\{0, +, f_1\}_i + \{0, +, g_1\}_i$	
5	$\{0, +, f_1 * g_1\}_i$	$\Rightarrow$	$f_1 * \{0, +, g_1\}_i$	when $i \notin FV[f_1]$
6	$\{0, +, \log f_1\}_i$	$\Rightarrow$	$\log\{1, *, f_1\}_i$	
7	$\{0, +, 1^i\}_i$	$\Rightarrow$	$\{0, +, 1\}$	
8	$\{0, +, f_1^i\}_i$	$\Rightarrow$	$\frac{f_1^{i+1}-1}{f_1-1}$	when $i \notin FV[f_1]$ and $f_1 \neq 1$
9	$\{0, +, f_1^{g_1+h_1}\}_i$	$\Rightarrow$	$\{0, +, f_1^{g_1} * f_1^{h_1}\}_i$	
10	$\{0, +, f_1^{g_1*h_1}\}_i$	$\Rightarrow$	$\{0, +, (f_1^{g_1})^{h_i}\}_i$	when $i \notin FV[f_1] \cup FV[g_1]$
11	$\{0, +, f_1\}_i$	$\Rightarrow$	$i * f_1$	when $i \notin FV[f_1]$
12	$\{0, +, i\}_i$	$\Rightarrow$	$\frac{1}{2}i^2 - \frac{1}{2}i$	
13	$\{0, +, i^n\}_i$	$\Rightarrow$	$\sum_{k=0}^{n} \frac{\binom{n+1}{k}}{n+1} B_k \, i^{n-k+1}$	for $n \in \mathbb{N}$
14	$\{1, *, -f_1\}_i$	$\Rightarrow$	$(-1)^i\{1, *, f_1\}_i$	
15	$\{1, *, \frac{1}{f_1}\}_i$	$\Rightarrow$	$\{1, *, f_1\}_i^{-1}$	
16	$\{1, *, f_1 * g_1\}_i$	$\Rightarrow$	$\{1, *, f_1\}_i * \{1, *, g_1\}_i$	
17	$\{1, *, f_1^{g_1}\}_i$	$\Rightarrow$	$f_1^{\{1, *, g_1\}_i}$	when $i \notin FV[f_1]$
18	$\{1, *, g_1^{f_1}\}_i$	$\Rightarrow$	$\{1, *, g_1\}_i{}^{f_1}$	when $i \notin FV[f_1]$
19	$\{1, *, f_1\}_i$	$\Rightarrow$	f_1^i	when $i \notin FV[f_1]$
20	$\{1, *, i\}_i$	$\Rightarrow$	0^i	
21	$\{1, *, i + f_1\}_i$	$\Rightarrow$	$\frac{(i+f_1-1)!}{(f_1-1)!}$	if $i \notin FV[f_1]$
22	$\{1, *, f_1 - i\}_i$	$\Rightarrow$	$(-1)^i * \frac{(i-f_1-1)!}{(-f_1-1)!}$	if $i \notin FV[f_1]$

Figure 7: $\mathcal{CR}^{-1}$

The idea of the automatic detection of recurrences in loops forms the basis of the loop optimization method explained in Section 3. Note that the one-line code Fig. 6 (3) requires the derivation of the closed-form function of the CR for $s(i)$, which will be explained in the next subsection.

2.10 CR Inverse

Fig. 7 depicts $\mathcal{CR}^{-1}$, a TRS to derive closed-form functions for CR-expressions. Exhaustive applications of the rules on a CR (which must be in recursive form Eq. (3)) results in a closed form function not containing any CRs as subexpressions if the CR has such a closed form function. The $\mathcal{CR}^{+}$ rules form a mapping from closed form functions to CRs that is neither injective nor surjective. This means that the inverse system $\mathcal{CR}^{-1}$ can only derive a subset of closed form functions corresponding to CRs. This subset is formed by certain compositions of polynomials, exponentials of polynomials, and factorials. The transformation strategy for applying $\mathcal{CR}^{-1}$ rules is to reduce the innermost redexes first on nested CR tuples of the form Eq. (3). This results in closed expressions for the increments in CR tuples which can be matched by other rules. After the application of the inverse transformations $\mathcal{CR}^{-1}$, the resulting closed form expression can be factorized to compress the representation.

The rules of $\mathcal{CR}^{-1}$ have been derived from axiomatic identities on BRs [4] and by using the CR and BR definitions of $\Phi_i = \{\phi_0, +, f_1\}_i$

$$\Phi_i = \phi_0 + \sum_{j=0}^{i-1} (f_1)_j$$

and the definitions of $\Phi_i = \{\phi_0, *, f_1\}_i$

$$\Phi_i = \phi_0 \prod_{j=0}^{i-1} (f_1)_j$$

Rules 9 and 10 bring CRs into a form such that other rules can apply. In rule 13 the Binomial formula is used for the closed form function of $\{0, +, i^n\}_i$ where B_k denotes the k^{th} Bernoulli number. The rule has been included to complete the TRS for deriving closed-form polynomials of nested polynomial CRs. A faster and more accurate algorithm to convert a simple polynomial CR $\{\phi_0, +, \phi_1, +, \ldots, +, \phi_k\}_i$ into a closed-form polynomial p in i of degree k exploits Newton series conversion, see also [4], using

$$p(i) = \sum_{j=0}^{k} \phi_j \binom{i}{j}$$

which is also known as Newton's forward formula for interpolation. By precomputing Sterling numbers, Bachmann shows that the conversion takes only $O(k^2)$ operations.

Application of rules 8, 12, 13, 21 and 22 requires rational arithmetic to avoid numerical roundoff in the translation to closed-form functions.

The system $\mathcal{CR}^{-1}$ is not defined for indeterminate CRs. Consider rule 8 of $\mathcal{CR}^{-1}$ with redex $\{0, +, 0^i\}_i$. Contraction yields $\frac{0^{i+1}-1}{-1}$ which simplifies to 1, because $0^{i+1} = 0$ and $i \geq 0$. This must be an invalid closed-form function, as $\mathcal{V}(\{0, +, 0^i\}_i) = 0$.

3 Loop Analysis and Loop Optimization

The method of loop optimization using symbolic evaluation of CRs proceeds in two phases. The first phase consists of an analysis phase for detecting loop induction variables and normalization of induction expressions. The second phase is a synthesis phase to generate optimized code using the CR normal forms calculated in the first phase. In the synthesis phase, loop optimizations such as strength reduction or induction variable substitution and code motion takes place.

3.1 Loop Analysis

Analysis proceeds with the determination of loop induction variables in a source code, which are replaced by their BR counterparts. When these have been substituted, normalized CR-expressions are constructed for every side-effect free expression occurring in the loop. This constraint ensures that functions will not assign values to the loop induction variables. A more advanced loop analysis scheme includes interprocedural analysis to determination which functions may change induction variables, cf. [12].

```
DO i = n,m
  a = b+c
  p = q+r
  S1(b, p)
  b = 2*b
  S2(b, b-c)
  c = h*i
  q = p
  r = r+c
  c = a
ENDDO
```

(1)

```
 DO i = n,m
(1) a_new = b_i+c_i
(2) p_new = q_i+r_i
(3) S1(b_i, q_i+r_i)
(4) b_new = 2*b_i
(5) S2(2*b_i, 2*b_i-c_i)
(6) q_new = q_i+r_i
(7) r_new = r_i+h*i
(8) c_new = b_i+c_i
 ENDDO
```

(2)

```
DO i = n,m
  a_new = b_i+c_i
  p_new = q_i+r_i
  S1(b_i, q_i+r_i)
  b_new = 2*b_i
  S2(2*b_i, 2*b_i-c_i)
  q_new = q_i+r_i
  r_new = r_i+h*{n,+,1}_i
  c_new = b_i+c_i
ENDDO
```

(3)

```
DO i = n,m
  a_new = b_i+c_i
  p_new = q_i+r_i
  S1(b_i, q_i+r_i)
  b_new = 2*b_i
  S2(2*b_i, 2*b_i-c_i)
  q_new = q_i+r_i
  r_new = r_i+{h*n,+,h}_i
  c_new = b_i+c_i
ENDDO
```

(4)

```
DO i = n,m
  a_new = {b,*,2}_i+c_i
  p_new = q_i+{r,+,h*n,+,h}_i
  S1({b,*,2}_i, q_i+{r,+,h*n,+,h}_i)
  b_new = {b,*,2}_i
  S2({2*b,*,2}_i, {2*b,*,2}_i-c_i)
  q_new = q_i+{r,+,h*n,+,h}_i
  r_new = {r,+,h*n,+,h}_i
  c_new = {b,*,2}_i+c_i
ENDDO
```

(5)

```
DO i = 0,m-n
  S1({b,*,2}_i, {q+r,+,r+h*n,+,h*n+h,+,h}_i)
  S2({2*b,*,2}_i, {2*b,*,2}_i-{c,+,b,*,2}_i)
  a_new = {b,*,2}_i+{c,+,b,*,2}_i
  b_new = {b,*,2}_i
  c_new = {c,+,b,*,2}_i
  p_new = {q+r,+,r+h*n,+,h*n+h,+,h}_i
  q_new = {q,+,r,+,h*n,+,h}_i
  r_new = {r,+,h*n,+,h}_i
ENDDO
```

(6)

Figure 8: Example Loop Analysis

Next, the CR-expressions are constructed for expressions by application of the $\mathcal{CR}^+$ simplification rules (Fig. 1 and Fig. 4). For nested loops, the loop analysis is applied recursively starting from the innermost loop.

More specifically, loop analysis proceeds by creating an intermediate form of the loop as follows

1. **The Derivation of an SSA (Single Static Assignment) Representation.** A hybrid SSA form [8] of the loop body is constructed by renaming scalar variables `x` to $\mathtt{x}_{\mathrm{new}}$ on the left-hand sides of assignment statements. The renaming is done only for variables in assignments of side-effect free expressions. After the variable renaming, data-flow analysis is used to propagate the right-hand sides forward to all of the respective variable uses (ignoring the flow and anti-dependencies). See Fig. 8 (1) (assignments are boxed). The result is shown in Fig. 8 (2). In the example code Fig. 8, it is assumed that the execution of statements `S1` and `S2` does not modify any of the variables in the loop. This may require interprocedural analysis to handle more general cases.

   Multiple assignments to the same variable can be removed, because the propagation in this case results in dead stores except for the last assignment statement which is kept in

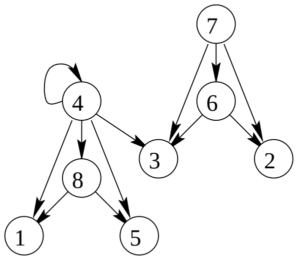

Figure 9: Dependency Graph of the Loop Body of the Example in Fig. 8 (2)

the loop body. All variables `x` assigned in the loop body are marked as i-loop-dependent variables $\mathtt{x}_i$.

2. **Determination of the Primary Induction Variables.**
We distinguish two types of primary induction variables: the *basic induction variables* consisting of the loop counter variables and the *dependent induction variables.*

   - The basic induction variable is the loop counter variable which is assumed to have no assignments in the loop body. The BR of a basic induction variable is $\{start, +, stride\}_i$, where i is the loop index variable, *start* is the start-value of the loop counter variable, and *stride* is the loop stride expression (both of which are obviously loop invariant). Every occurrence of the basic induction variable is replaced by its BR in all expressions occurring in the loop body, see Fig. 8 (2) and (3) where the box in (2) indicates the replacement of i with its BR; The expressions are simplified by applying the CR simplification rules $\mathcal{CR}^+$, resulting in Fig. 8 (4). Note that in the simplification process variable `h` is treated as loop invariant while $\mathtt{r}_i$ is dependent on i.
   - The dependent induction variables are found by an in-order traversal of the dependence graph spanned by arcs pointing in the direction of the flow dependencies and arcs pointing in the *reverse direction* of anti-dependencies between assignments (an example graph is shown in Fig. 9). For this algorithm to work the graph has to be acyclic, ignoring self-cycles of nodes (see also Section 3.1.3). The self-cycles in the graph are restricted to nodes that represent update assignments of dependent induction variables. The BR of a dependent induction variable $\mathtt{x}_{\text{new}}$ is $\{x_0, \odot, h\}_i$ derived from the assignment statement $\mathtt{x}_{\text{new}} = \mathtt{x}_i \odot h$ with $\odot \in \{+, *\}$ or algebraic equivalents[2]. Expression h can be a loop-invariant expression or a CR. The isolation of expression h from the right-hand side of an assignment statement may require the application of the associative and commutative laws. The initial value x_0 is equal to the value of the variable at the start of the loop. If data-flow analysis of the source code cannot determine this value, the initial value is set to the variable, i.e. we can set $x_0 = \mathtt{x}$ as the initial value of the BR. The uses of induction variable $\mathtt{x}_{\text{new}}$ in all expressions in the loop are replaced by the corresponding

[2]For example, bitwise shift operators and inverse operators, e.g. $\odot = -$ (subtraction) $\odot$ can be replaced by $+$ and negating h. For $\odot = /$ (division) $\odot$ is set to $*$ and h is inverted.

```
j = m
DO i = m,n
  a(j) = b(k)
  j = i+1
  k = j
ENDDO
```
(1)

```
j = m
DO i = m,n
  a(j_i) = b(k_i)
  j_new = {m+1,+,1}_i
  k_new = {m+1,+,1}_i
ENDDO
```
(2)

```
j = m
DO i = 0,n-m
  a({m,+,1}_i) = b({k-m,*,0}_i + {m,+,1}_i)
  j_new = {m+1,+,1}_i
  k_new = {m+1,+,1}_i
ENDDO
```
(3)

Figure 10: Analysis of Wraparound Variables

BR of the variable after which the expression is simplified by applying the CR simplification rules $\mathcal{CR}^+$. The in-order traversal of the dependence graph ensures that the updates h in assignments are first analyzed for CRs. The assignments to variables `b` and `r` in Fig. 8 (4) have no dependence (flow or anti), so they are analyzed first. Primary induction variable `b` has the CR $\{\mathtt{b}, *, 2\}_i$ and `r` has the CR $\{\mathtt{r}, +, \mathtt{h} * \mathtt{n}, +, \mathtt{h}\}_i$ which are substituted in the loop resulting in Fig. 8 (5). The assignments to variables `c` and `q` in Fig. 8 (5) are now analyzed. Variable `c` has the CR $\{\mathtt{c}, +, \mathtt{b}, *, 2\}_i$ and `q` has the CR $\{\mathtt{q}, +, \mathtt{r}, +, \mathtt{h} * \mathtt{n}, +, \mathtt{h}\}_i$ which are substituted in the loop. The expressions are simplified and assignments can be grouped in a lexicographical order resulting in the intermediate representation shown in Fig. 8 (6).

3. **Recognition of Indirect Induction Variables.**
   Application of the CR simplification rules on expressions in the loop reveals *indirect induction variables*. These variables are indirect recurrences and their values in the loop solely depend on the values of other induction variables. For example, variables `a` and `p` in Fig. 8 (6) are indirect induction variables. Note that the closed-form function of induction variable `a` is represented by a CR-expression, not a pure CR, which demonstrates the ability of the method to analyse induction variables whose progressions are described by arbitrary closed-form functions.

4. **Wraparound Variable Analysis.**
   Wraparound variables are indirect induction variables with assignment statements that induce anti-dependences between statements in the loop body. The CR-expressions that were derived by CR loop analysis for these variables can be propagated forward to all of the variables uses, see the example in Fig. 10 (1) and (2). The example is taken from [12] (page 52). A wraparound variable has at least one use of the variable before its (first) assignment statement in the loop body, as is the case for variables `j` and `k` in Fig. 10 (1). For wraparound variable analysis, each use of the variable `x` before its definition is replaced by the indeterminate CR-expression

$$\{x_0 - \mathcal{V}(\mathcal{B}(\Phi_i)), *, 0\}_i + \mathcal{B}(\Phi_i) \tag{8}$$

   where Φ_i is the CR-expression of the indirect induction variable `x` and x_0 is the initial value of `x` at the start of the loop or `x` itself if the initial value is not known. Fig. 10 (3)

```
a = 0.0
b = 1.0
k = 1
DO
  a = a+b
  b = -h*x*b/k
  k = k+1
WHILE ABS(b)>1e-10
```
(1)

```
a = 0.0
b = 1.0
k = 1
DO
```
$$\mathtt{a}_{\mathrm{new}} = \{0,+,1,*,\frac{(\mathtt{-h*x})^i}{\{1,+,1\}_i}\}_i\}_i$$
$$\mathtt{b}_{\mathrm{new}} = \{1,*,\frac{(\mathtt{-h*x})^i}{\{1,+,1\}_i}\}_i$$
$$\mathtt{k}_{\mathrm{new}} = \{1,+,1\}_i$$
WHILE ABS($\mathtt{b}_i$)>1e-10

(2)

```
i = 0
DO
  b = (-h*x)**i/FAC(i)
  i = i+1
WHILE ABS(b)>1e-10
a = EXP(-h*x)
```
(3)

Figure 11: Recognition of the Maclaurin Series for e^{-h*x}

depicts the loop with CR-expressions obtained after application of Eq. (8) and CR simplification. Variable j is recognized as a simple linear recurrence by the derivation

$$\begin{aligned}
&\{\mathtt{m} - \mathcal{V}(\mathcal{B}(\{\mathtt{m}+1,+,1\}_i)),*,0\}_i + \mathcal{B}(\{\mathtt{m}+1,+,1\}_i) \\
&\quad = \{\mathtt{m} - \mathcal{V}(\{\mathtt{m},+,1\}_1),*,0\}_i + \{\mathtt{m},+,1\}_i \\
&\quad = \{0,*,0\}_i + \{\mathtt{m},+,1\}_i \\
&\quad = \{\mathtt{m},+,1\}_i
\end{aligned}$$

The wraparound variable analysis of the Parafrase-2 compiler [12] relies on the "abuse" of 0^i, which is assumed to be 1 for $i = 0$ and 0 for $i > 0$. Our approach is related to this trick, but avoids the expansion of the indeterminate CR into the problematic form 0^i. For wraparound variable analysis, the underlying closed-form function of the indeterminate CR $\{\phi_0,*,0\}_i$ can be viewed as $\phi_0\delta_i^0$. where δ is the Kronecker delta function.

After CR loop analysis with wraparound variable analysis, every scalar variable on the left-hand side of an assignment with has been renamed and every use of such a scalar variable is changed into a CR-expression.

After the analysis, the assignments are grouped and lexicographically reordered and the loop is normalized by modifying the lower and upper bounds such that the lower bound is set to zero. Hence, the analysis derives a normalized intermediate form of loop code containing CRs as normal forms for polynomials, exponentials, and factorials. The computational complexity of loop analysis is linear in the code size of the loop body. However, CR simplification in the loop analysis may require $o(n^2)$ time for simplifying n-degree polynomials using Bachmann's algorithm [6].

3.1.1 Power Series Loop Analysis

The CR loop analysis method provides an algebraic basis for the analysis of power series calculations in loops. For example, the method can be adapted to recognize Taylor and

Maclaurin expansions of closed-form functions. Consider for example the loop in Fig. 11 (1). Loop analysis and subsequent CR synthesis result in loops Fig. 11 (2) and (3). Note that if variables `b` and `k` are not live at the end of the loop in Fig. 11 (3), the loop can be completely eliminated.

For the recognition of the recurrences that approximate exponentials (e^E), where E is an arbitrary expression, we added a new rule

$$\{0, +, \frac{E^i}{i!}\}_i \Rightarrow e^E$$

to $\mathcal{CR}^{-1}$. However, in contrast to the rules in $\mathcal{CR}^{-1}$ that translate CRs into closed-form functions that are deemed to be mathematically equivalent (up to numerical roundoff), the new rule replaces an approximation by an exact solution. That is, the CR on the left-hand side of the rule is a series approximation of the function on the right-hand side.

An application of this technique is the automatic verification of the correctness of an iterative power series algorithm. The technique also enables a symbolic analysis of convergence tests on power series in existing programs. For example, the technique recognizes that recurrence variable `b` in Fig. 11 (1) forms the sequence of terms in the series. For loop optimization using this approach, care has to be taken that the number of loop iterations in the original loop is sufficiently large to justify the replacement.

This example demonstrates that the addition of single rule suffices to recognize a power series approximation in the CR loop synthesis phase. In this paper, we will not further investigate the use of CR analysis for power series recognition.

3.1.2 Conditional Induction Expressions

For the analysis of conditional induction expressions we adopt a similar approach as that of Haghighat [12]. Conditional induction expressions are analyzed by traversing all possible execution paths through a loop body. On each path, CR-expressions are constructed for the induction variables. If every possible path through the loop body leads to the construction of the same closed-form expression for an induction variable, the induction variable can be replaced by its recurrence or closed-form throughout the loop nest.

Consider example Fig. 12 (1) taken from [12] (p. 50). The innermost loop is analyzed first and the CR $\{k, +, 1, +, 1\}_j$ is constructed for induction variable `k`. At the end of the innermost loop, the value of `k` is updated to $\mathtt{k} + (\mathtt{i} * (\mathtt{i} + 1))/2$. The closed-form expression is obtained by application of $\mathcal{CR}^{-1}$. The update of variable `k` in the else-branch is the same. After analysis of the outermost loop Fig. 12 (3) resulting in the multivariate CR $\{\{0, +, 0, +, 1, +, 1\}_i + 1, +, 2, +, 1\}_j$ for the index expression of array `b`, the program can be rewritten as in Fig. 12 (4). Note that this is different from [12] (p. 50) which contains an error.

3.1.3 Cyclic Dependencies and Cyclic Recurrences

We distinguish cyclic dependencies of loop variables and variables with cyclic recurrence relations. The loop analysis algorithm recognizes induction variables with cyclic dependencies. An example is shown in Fig. 13. The analysis propagates `y+a` to the assignment statement

```
k = 0
DO i = 1,n
  IF (cond(i)) THEN
    DO j = 1,i
      k = k+j
      a(i,j) = b(k)
    ENDDO
  ELSE
    k = k+(i*(i+1))/2
  ENDIF
ENDDO
```
(1)

```
k = 0
DO i = 1,n
  IF (cond(i)) THEN
    DO j = 1,i
      a(i,j) = b({k_i + 1,+,2,+,1}_j)
    ENDDO
    k = k+(i*(i+1))/2
  ELSE
    k = k+(i*(i+1))/2
  ENDIF
ENDDO
```
(2)

```
DO i = 1,n
  IF (cond(i)) THEN
    DO j = 1,i
      a(i,j) = b({{0,+,0,+,1,+,1}_i + 1,+,2,+,1}_j)
    ENDDO
  ENDIF
  k_new = {0,+,0,+,1,+,1}_i
ENDDO
```
(3)

```
DOALL i = 1,n
  IF (cond(i)) THEN
    DOALL j = 1,i
      a(i,j) = b((i**3-i+3*j*j+9*j)/6+1)
    ENDDO
  ENDIF
ENDDO
```
(4)

Figure 12: Example Conditional Induction Variable Recognition

for variable y after which y is recognized as a primary induction variable. Substitution yields Fig. 13 (3), where variable x is recognized as an indirect induction variable.

The loop analysis algorithm does not recognize induction variables with cyclic recurrences. Fig. 14 depicts an example loop with cyclic recurrence relations. Variables x1 and x2 are cyclic. It is not obvious that variable x1 is a sixth-order polynomial, as shown in Fig. 14 (3), and the algorithm fails to form CRs for this example.

We believe that the differencing method [12] suffers from a similar problem. The maximum polynomial order of loop induction variables has to be guessed, which is hard in the presence of cyclic recurrences. When the maximum polynomial order m for analysis is set too low, for example $m \leq 4$, the differencing method [12] recognizes induction variable x1 with the wrong polynomial $(i^2-8i+16)$ when analyzing the first $m+2$ terms of the sequence

```
DO i = 0,n
  x = y+a
  y = x+b
ENDDO
```
(1)

```
DO i = 0,n
  x_new = {y + a,+,a + b}_i
  y_new = {y,+,a + b}_i
ENDDO
```
(2)

Figure 13: Example Cyclic Dependencies

```
x1 = 16
x2 = 9
x3 = 0
x4 = 0
DO i = 0,n
  t = x1
  x1 = x2
  x2 = x3+2*x2-t+2
  x3 = x3+x4
  x4 = x4+i
ENDDO
```

(1)

```
x1 = 16
x2 = 9
x3 = 0
x4 = 0
DO i = 0,n
  t_new = x1_i
  x1_new = x2_i
  x2_new = x3_i+2*x2_i-x1_i+2
  x3_new = {0,+,0,+,0,+,1}_i
  x4_new = {0,+,0,+,1}_i
ENDDO
```

(2)

x1=$(i^6+85i^4+994i^2-15(i^5+15i^3+392i))/720+16$
x2=$(i^6+85i^4+994i^2-15(i^5+15i^3+296i))/720+9$
x3=$(i^3+2i-3i^2)/6$
x4=$(i^2-i)/2$

(3)

Figure 14: Example Cyclic Recurrences

$25, 16, 9, 4, 1, 0, 2, 11, 9, \ldots$ of values of `x1`. The differencing method is not safe as the degree cannot be easily determined in the presence of cyclic recurrences.

3.2 Loop Synthesis and Optimization

After CR analysis of the source code, CR loop synthesis constructs closed-form functions for CRs. The synthesis forms a basis for loop parallelization. CR loop analysis alone can be used to derive efficient serial and vector loops. The induction variable recognition using CRs results in the discovery of more induction variables than the methods described in [14, 21, 12].

3.2.1 Loop Induction Variable Substitution

Application of $\mathcal{CR}^{-1}$ rules on the CRs in Fig. 8 (6) and subsequent symbolic simplification of the arithmetic expression results in the code shown in Fig. 15 (1). Notice that $\mathtt{p}_{\text{new}}$ is a third-degree polynomial and $\mathtt{c}_{\text{new}} = \mathtt{c} + \mathtt{b} * \sum_{i=0}^{m-n} 2^i = \mathtt{c} + \mathtt{b} * (2^{\mathtt{m}-\mathtt{n}+1} - 1)$ is a geometric series. Polynomials and powers-of-two expressions are common in scientific codes, see e.g. [12]. Powers of two are frequently used to form bit patterns for masking operations.

In Fig. 15 (2) an efficient parallel loop is shown that is generated from the intermediate representation Fig. 15 (1). The loop is parallelized (`DOALL` denotes a parallel do-all loop) Code motion moved all assignments out of the parallel loop. The assignments are ordered using the dependence graph constructed in the CR loop analysis phase. The dependence graph constructed at CR analysis is acyclic and defines a partial ordering on the variable assignments. The variables are ordered in reverse order of the topological order defined by the arcs of the graph. This allows the elimination of the variable renaming, because of the absence of flow dependencies in the resulting set of assignments. Assignments to variables that are not live at the loop end can be removed altogether.

```
DOALL i = 0,m-n
  a_new = c+b*(3*2**i-1)
  p_new = q+h*i*(i**2+3*n*(i+1)-1)/6
          +r*(i+1)
  S1(b*2**i, q+h*i*(i**2+3*n*(i+1)-1)/6
              +r*(i+1))
  b_new = b*2**i
  S2(b*2**(i+1), b-c)
  q_new = q+h*i*(i**2+3*n*(i+1)-1)/6
          +r*(i+1)
  r_new = r+h*i*(n-(1-i)/2)
  c_new = c+b*(2**(i+1)-1)
ENDDO
```

(1)

```
DOALL i = 0,m-n
  S1(b*2**i, q+h*i*(i**2+3*n*(i+1)-1)/6
             +r*(i+1))
  S2(b*2**(i+1), b-c)
ENDDO
IF m-n >= 0 THEN
  a = c+b*(3*2**(m-n)-1)
  p = q+h*(m-n)*((m-n)**2+3*n*(m-n+1)-1)/6
      +r*(m-n+1)
  q = q+h*(m-n)*((m-n)**2+3*n*(m-n+1)-1)/6
      +r*(m-n+1)
  c = c+b*(2**(m-n+1)-1)
  b = b*2**(m-n)
  r = r+h*(m-n)*(n-(1-m+n)/2)
ENDIF
```

(2)

Figure 15: Example Parallel Loop Synthesis

3.2.2 Loop-Invariant Expression Removal

CR analysis and synthesis of a loop discovers loop invariant expressions. The CRs that are simplified to loop-invariant expressions at loop analysis are the first candidates for loop-invariant expression removal. Additional loop-invariant expressions are detected during loop synthesis, A non-trivial example is the second argument `b-c` of statement `S2` in Fig. 15 (1). From the original source code Fig. 8 (1), it is not obvious that `b-c` is loop invariant. The CR method automatically recognizes that $\mathtt{b} - \mathtt{c} = \mathtt{b}_0 * 2^i - \mathtt{c}_0 + \mathtt{b}_0 * \sum_{i=0}^{\mathtt{m}-\mathtt{n}} 2^i = \mathtt{b}_0 - \mathtt{c}_0$, where $\mathtt{b}_0$ and $\mathtt{c}_0$ are the initial values of `b` and `c` at the start of the loop.

3.2.3 Strength Reduction

Strength reduction optimization is a technique that substitutes expensive operations by semantically equivalent faster to evaluate operations [1, 2]. The CR analysis method generalizes strength reduction of polynomials, exponentials, logarithms, trigonometric functions, and factorials. Strength reduction replaces the induction variables and expressions in a loop by recurrences. For each recurrence, the template Fig. 5 is used to expand the recurrence into a sequence of induction variable updates in the loop body.

3.2.4 Common-Tail-Recurrence Elimination

Recurrences that have identical tail recurrences are said to have *common-tail recurrences.* For example, the CRs $\{0, +, \{1, +2\}_i\}_i$ and $\{1, +, \{1, +, 2\}_i\}_i$ have the same tail recurrences $\{1, +, 2\}_i$. Common-tail-recurrence elimination is essential to be applied on CRs before strength reduction can be applied. Otherwise, a new set of induction variables for the template Fig. 5 is introduced for every occurrence of the same recurrence expression in the loop. The nested recurrence representation enables the elimination of common-tail recur-

```
DO i = 0,n
  P(i*i,i*i+m,2*i+1)
ENDDO
```

(1)

```
DO i = 0,n
  P({0,+,1,+,2}_i,{m,+,1,+,2}_i,{1,+,2}_i)
ENDDO
```

(2)

```
cr0 = 0
cr1 = m
cr2 = 1
DO i = 0,n
  P(cr0,cr1,cr2)
  cr0 = cr0+cr2
  cr1 = cr1+cr2
  cr2 = cr2+2
ENDDO
```

(3)

Figure 16: Example Common-Tail-Recurrence Elimination and Strength Reduction

rences. Fig. 16 depicts common-tail-recurrence elimination and strength reduction for an example loop. Without common-tail-recurrence elimination, two extra induction variables are introduced in the example loop.

4 Loop Dependence Analysis

Haghighat [12] shows that monotonicity of polynomial index expressions forms a basis for non-linear dependence analysis. His method requires a symbolic tabulation of the derivatives of index expressions to determine monotonicity. The method lacks the analysis of the difference between two index expressions necessary for the formation of distance vectors. However, polynomial differences can be formed from the characteristic functions of index expressions. The determination of the signs of polynomial differences yields a direction vector. The determination requires a tabulation of a difference table, which is an expensive operation when performed for every pair of index expressions that is subject to dependence analysis.

In contrast, the CR loop analysis method constructs CR-expressions for arbitrary index expressions and the difference CRs can be calculated in linear time using the CR simplification rules. From the resulting difference CRs, non-linear distance vectors and direction vectors can be constructed, thereby providing a fast and effective method for non-linear array index dependence testing.

4.1 Sign and Monotonicity of CR-Expressions

Definition 4.1 *Let* $\Phi_i = \{\phi_0, \odot_1, \phi_1, \odot_2, \ldots, \odot_k, f_k\}_i$ *be a CR. The CR is* positive *(*non-negative*)* $\Phi_i \geq 0$ *if* $\phi_j \geq 0$ *for all* $0 \leq j < k$ *and* $f_k \geq 0$*. Similarly, a CR* $\Phi_i \leq 0$ *if* $-\Phi_i \geq 0$ *for all* $0 \leq j < k$ *and* $f_k \geq 0$*.*

Note: the system $\mathcal{CR}^+$ can be used to obtain $-\Phi_i$.

The sign of a CR-expression is determined by the signs of its constituent CRs. Fig. 17 lists a subset of rules for determining the sign of some common arithmetic operators that

$\lfloor(\geq)\rfloor$	$\Rightarrow$	$(\geq)$	$(\geq)/(\geq)$	$\Rightarrow$	$(\geq)$
$(\geq)+(\geq)$	$\Rightarrow$	$(\geq)$	$(\leq)/(\leq)$	$\Rightarrow$	$(\geq)$
$(\geq)-(\leq)$	$\Rightarrow$	$(\geq)$	$(\geq)^{(\geq)}$	$\Rightarrow$	$(\geq)$
$(\geq)*(\geq)$	$\Rightarrow$	$(\geq)$	$\max((\geq),(\geq))$	$\Rightarrow$	$(\geq)$
$(\leq)*(\leq)$	$\Rightarrow$	$(\geq)$	$\min((\geq),(\geq))$	$\Rightarrow$	$(\geq)$

Figure 17: Subset of Rules for Determining the Sign of Operators

may appear in CR-expressions. For example, the CR-expression $\{0,+,2\}_i + \{1,*,2\}_i \geq 0$ because $\{0,+,2\}_i \geq 0$ and $\{1,*,2\}_i \geq 0$.

If a CR-expression is positive (negative), then its closed-form function is positive (negative) on its interval. More formally stated

Theorem 4.2 *Let f be a closed-form function with CR-expression Φ_i for $f(x)$ on $x = x_0 + i\,h$, $i = 0,\ldots,n$, $(n > 0)$. If $\Phi_i \geq 0$ then $f(x) \geq 0$ for all $x = x_0 + i\,h$, $i = 0,\ldots,n$. Similarly, if $\Phi_i \leq 0$ then $f(x) \leq 0$ for all $x = x_0 + i\,h$, $i = 0,\ldots,n$.*

Proof. See Appendix D. □

For example, the CR-expression of $2i + i^2$ is $\{0,+,2\}_i + \{1,*,2\}_i$ which is positive. Clearly, the function is positive for $i \geq 0$.

Note that Theorem 4.2 does not take the upper limit of an interval into account. For example, the CR $\{10,+,-1\}_i$ on interval $i = 0,\ldots,10$ cannot be determined to be positive, while the function it represents (i.e. $10 - i$) certainly is.

The determination of index expression monotonicity is important in loop optimizations [12]. A non-indeterminate CR of the form $\{\phi_0,+,\phi_1,\odot_2,\ldots,\odot_k,f_k\}_i$ is *monotonic* if $\{\phi_1,\odot_2,\ldots,\odot_k,f_k\}_i \geq 0$.

4.2 CR-Based Distance and Direction Vectors

The signs of CR-expressions constructed from array index expressions is exploited in the formation of dependence direction vectors for a loop. Construction of the direction vectors amounts to the calculation of the differences between the CR-expressions of index expressions. These CR-expressions are available after loop analysis. A CR-based distance vector is constructed for a given pair of array references. A CR-based distance vector is similar to the "traditional" distance vector (see e.g. [14, 21]), except that the vector coefficients are formed by CR-expressions. The direction vector is formed by the determination of the signs of the distance vector coefficients using Theorem 4.2. The computational complexity of computing a distance and direction vector is $O(n)$, where n is the maximum length of the CRs involved. For polynomial array index expressions, n is the maximum order of the polynomials.

Consider for example the code fragment shown in Fig. 18 (1). Existing restructuring compilers are able to parallelize the `k`-loop. However, they fail to optimize the overall loop nest, because of the presence of various non-linear array index expressions.

The CR-based dependence test calculates the difference of the CR-expressions of index expressions in the construction of distance vectors. The signs of the coefficients of the

```
j1 = c
DO k = 0,9,2
  DO j = 0,m
    DO i = 0,n
      a[i+j,j+c,s*k] = a[i*i+j+1,j1,s*k]
      b[j1+i] = i+j
    ENDDO
    j1 = j1+h
  ENDDO
ENDDO
```

(1)

```
DO i=0,n
  IF h=1 THEN
    DOALL j=0,m
      DOALL k=0,9,2
        a[i+j,j+c,s*k] = a[i*i+j+1,j+c,s*k]
      ENDDO
      b[i+j+c] = i+j
    ENDDO
  ELSE
    DO j=0,m
      DOALL k=0,9,2
        a[i+j,j+c,s*k] = a[i*i+j+1,h*j+c,s*k]
      ENDDO
    ENDDO
    IF h≠0 THEN
      DOALL j=0,m
        b[i+h*j+c] = i+j
      ENDDO
    ELSE
      b[i+c] = i+m
    ENDIF
  ENDIF
ENDDO
```

(2)

Figure 18: Example CR-Based Loop Parallelization

$$
\begin{array}{ccc}
[\{\{0,+,1\}_j,+,1,+,2\}_i,\{c,+,h\}_j,\{0,+,2s\}_k] & \langle\{-1,+,0,+,-2\}_i,\{0,+,1-h\}_j,0\rangle & [\{\{1,+,1\}_j,+,1\}_i,\{c,+,1\}_j,\{0,+,2s\}_k] \\
[\{\{0,+,1\}_j,+,1,+,2\}_i,\{c,+,h\}_j,\{0,+,2s\}_k] & \langle\leq,\pm,=\rangle & [\{\{1,+,1\}_j,+,1\}_i,\{c,+,1\}_j,\{0,+,2s\}_k]
\end{array}
$$

Figure 19: Example CR-Based Distance and Direction Vectors

distance vector forms the direction vector. For example, the distance and direction vectors formed by the assignment to array `a` are shown in Fig. 19. The analysis of the loop nest results in multivariate CR normal forms for index expressions, where the index ordering of the multivariate CRs corresponds to the loop nesting.

With the direction vectors and the determination of the monotonicity of the `b` array index in the loop, the loop nest can be restructured and inner loops can be parallelized as is shown in Fig. 18 (2). The CR of the index expression of `b` is strictly monotonic. Strict monotonicity of an integer-valued CR $\{\phi_0, +, \phi_1, \odot_2, \ldots, \odot_k, f_k\}_i$ can be determined by verifying that $\{\phi_1, \odot_2, \ldots, \odot_k, f_k\}_i - 1 \geq 0$. In the parallelized code Fig. 18 (2), a run-time check on the value of the CR coefficient `h` has been included. The check `h=1` ensures that the second component of the direction vector is zero, so the `j`-loop can be executed in parallel.

To complement the determination of the sign of a CR, techniques like value-range propagation can be used [7] can be used. The value-range propagation technique computes conservative value ranges of expressions and it can be used to determine the existence of a data-dependence [7]. However, the value-range propagation method does not take the dynamic behavior of (induction) variables into account, while CRs describe the dynamic behavior of any closed-form function of induction variables and recurrences.

5 Conclusions

The chains of recurrence (CR) representation is an effective normal form for recurrences and induction expressions in loops. The representation enables the optimization of loops by loop induction expression recognition, loop parallelization by induction variable substitution, loop strength reduction, and loop invariant expression elimination (code motion). We have shown that CR loop analysis is relatively simple to implement, compared to the abstract interpretation technique developed by Haghighat. A term rewriting system on CRs and a set of primitive arithmetic rules is sufficient, while symbolic differencing heavily relies on abstract interpretation for evaluating the first couple of iterations of a loop for the tabulation of a difference table. In addition, the CR-based loop analysis is not restricted to differentiable expressions in loops. The CR methods are more powerful as the representation is more general than Haghighat's characteristic function representation of generalized induction variables. Haghighat demonstrated the performance impact of the Parafrase-2 loop optimization on the Perfect Benchmarks$^{(R)}$ and we are confident that our approach will be just as effective. Finally, CR-based dependence testing provides a new dependence algorithm for loop restructuring. Other applications of the method are in program equivalence testing and program correctness verification.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley Publishing Company, Reading MA, 1985.

[2] F. Allen, J. Cocke, and K. Kennedy. Reduction of operator strength. In S. Munchnick and N. Jones, editors, *Program Flow Analysis*, pages 79–101, New-Jersey, 1981. Prentice-Hall.

[3] Z. Ammerguallat and W. H. III. Automatic recognition of induction variables and recurrence relations by abstract interpretation. In *ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 283–295, White Plains, NY, 1990.

[4] O. Bachmann. *Chains of Recurrences.* PhD thesis, Kent State University of Arts and Sciences, 1996.

[5] O. Bachmann. Chains of recurrences for functions of two variables and their application to surface plotting. In N. Kajler, editor, *Human Interaction for Symbolic Computation.* Springer-Verlag, 1996.

[6] O. Bachmann, P. Wang, and E. Zima. Chains of recurrences - a method to expedite the evaluation of closed-form functions. In *International Symposium on Symbolic and Algebraic Computing*, pages 242–249, Oxford, 1994. ACM.

[7] W. Blume and R. Eigenmann. Demand-driven, symbolic range propagation. In 8^{th} *International workshop on Languages and Compilers for Parallel Computing*, pages 141–160, Columbus, Ohio, USA, Aug. 1995.

[8] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13:451–490, Oct. 1991.

[9] R. Eigenmann, J. Hoeflinger, G. JAxon, Z. Li, and D. Padua. Restructuring fortran programs for cedar. In *ICPP*, volume 1, pages 57–66, St. Charles, Illinois, 1991.

[10] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. Experience in the automatic parallelization of four perfect-benchmark programs. In 4^{th} *Annual Workshop on Languages and Compilers for Parallel Computing, LNCS 589*, pages 65–83, Santa Clara, CA, 1991. Springer Verlag.

[11] M. Haghighat and C. Polychronopoulos. Symbolic program analysis and optimization for parallelizing compilers. In 5^{th} *Annual Wprkshop on Languages and Compilers for Parallel Computing, LNCS 757*, pages 538–562, New Haven, Connecticut, 1992. Springer Verlag.

[12] M. R. Haghighat. *Symbolic Analysis for Parallelizing Compilers.* Kluwer Academic Publishers, 1995.

[13] D. Knuth and P. Bandix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

[14] S. Munchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, San Fransisco, CA, 1997.

[15] C. Polychronopoulos, M. Girkar, M. Haghighat, C. Lee, B. Leung, and D. Schouten. Parafrase-2: An environment for parallelizing, partitioning, synchronuzing and scheduling programs on multiprocessors. In *ICPP 1989*, volume II, pages 39–48, St. Charles, Illinois, Aug. 1989.

[16] R. van Engelen, L. Wolters, and G. Cats. CTADEL: A generator of multi-platform high performance codes for pde-based scientific applications. In 10^{th} *ACM International Conference on Supercomputing*, pages 86–93, New York, 1996. ACM Press.

[17] R. van Engelen, L. Wolters, and G. Cats. Tomorrow's weather forecast: Automatic code generation for atmospheric modeling. *IEEE Computational Science & Engineering*, 4(3):22–31, July/September 1997.

[18] M. Wolfe. Beyond induction variables. In *ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 162–174, San Fransisco, CA, 1992.

[19] E. Zima. Recurrent relations and speed-up of computations using computer algebra systems. In *DISCO'92*, pages 152–161. LNCS 721, 1992.

[20] E. Zima. Simplification and optimization transformations of chains of recurrences. In *International Symposium on Symbolic and Algebraic Computing*, Montreal, Canada, 1995. ACM.

[21] H. Zima. *Supercompilers for Parallel and Vector Computers.* ACM Press, New York, 1990.

A Completeness of $\mathcal{CR}$

We prove that $\mathcal{CR}$ is complete and $\mathcal{CR}$ with associativity and commutativity of $+$ and $*$ is complete for univariate CRs. A TRS that is complete is guaranteed to reduce terms into unique normal forms in a finite number of reduction steps (rewrite rule applications).

Definition A.1 *Basic TRS notions:*

- *A TRS is* complete *if it is* confluent *and* terminating.
- *A TRS is* confluent *if* $\forall a, b, c : \exists d : (a \Rightarrow b \wedge a \Rightarrow c) \rightarrow (b \Rightarrow d \wedge c \Rightarrow d)$.
- *A TRS is* terminating *if every reduction sequence* $t_0 \Rightarrow t_1 \Rightarrow \cdots$ *must eventually terminate.*

- *A* critical pair $\langle s_1, s_2\rangle$ *is formed by unifying the left-hand side ℓ of one reduction rule with subterm(s) of the left-hand side $t = (\cdots\ell\cdots)$ of another reduction rule $t \Rightarrow s$ and applying the first rule on the subterm ℓ in t giving s_1 and applying the second rule on t giving s_2. For example, rules $s(0) \Rightarrow 1$ and $f(s(x)) \Rightarrow x$ on $f(s(0))$ form the critical pair $\langle f(1), 0\rangle$.*

- *Term t* has a normal form *if $t \stackrel{*}{\Rightarrow} s$ for some normal form s, where $\stackrel{*}{\Rightarrow}$ is the reflexive transitive closure of the reduction relation $\Rightarrow$.*

- *Term t* is a normal form *if there is no s such that $t \Rightarrow s$.*

Theorem A.2 *$\mathcal{CR}$ is terminating.*

Proof. It is easy to verify that $\mathcal{CR}$ is terminating, since every redex is reduced to a CR of the form $\{E, \odot, F\}$ for some term E, operator $\odot \in \{+, *\}$, and term F. Because E does not contain any redexes it is not further reduced. For rules 7, 9, and 10, term F contains new redexes.

In rule 7 the redexes in term F are $\{\phi_0, +, f_1\}_i * g_1$ and $\{\psi_0, +, g_1\}_i * f_1$ assuming that f_1 and g_1 are fully reduced. The redexes can be reduced using rules 2 and 7. Rule 2 will terminate the reduction sequence. Rule 7 is applied when f_1 or g_1 are CRs. Suppose that f_1 (or g_1) is a CR of the form $\{\phi, \odot, f\}$. then the reduction sequence will terminate when f is not a CR. Since there can only be a finite number of nested CRs in f_1 (or g_1), the redexes on the right-hand side of rule 7 are reduced in a finite sequence of reduction steps.

In rule 9 the redexes in term F are $\{\phi_0, +, f_1\}_i^{g_1}$ and $f_1^{\{\psi_0, +, g_1\}_i}$ assuming that f_1 and g_1 are fully reduced. The redexes can be reduced using rules 4, 5, and 9. Rules 4 and 5 will terminate the reduction sequence. By the same argument as for rule 7, the redexes on the right-hand side of rule 9 are reduced in a finite sequence of reduction steps.

In rule 10, F contains a product of CRs $\{\phi_0 + j, +, f_1\}$, $(j = 1, \ldots, |f_1|)$. This product of recurrences forms a redex for rule 7 only. Rule 7 reduces the product in a finite sequence of reduction steps. □

Theorem A.3 (Knuth and Bandix [13]) *Let TRS R be terminating. Then R is confluent iff each critical pair has a common reduct.*

Theorem A.4 *$\mathcal{CR}$ is complete.*

Proof. In $\mathcal{CR}$ there are no critical pairs. Hence, by Theorem A.3, $\mathcal{CR}$ is confluent. □

Theorem A.5 *$\mathcal{CR}$ with associativity and commutativity of $+$ and $*$ is complete for univariate CRs.*

Proof. The extension of $\mathcal{CR}$ with associativity and commutativity of $+$ and $*$ requires adding the rules

$$
\begin{array}{llll}
1' & \{\phi_0,+,f_1\}_i + E & \Rightarrow & \{E+\phi_0,+,f_1\}_i & \text{when } i \notin FV[E] \\
2' & \{\phi_0,+,f_1\}_i * E & \Rightarrow & \{E*\phi_0,+,E*f_1\}_i & \text{when } i \notin FV[E] \\
3' & \{\phi_0,*,f_1\}_i * E & \Rightarrow & \{E*\phi_0,*,f_1\}_i & \text{when } i \notin FV[E]
\end{array}
$$

The addition of these rules introduces critical pairs. Consider the redex $\{\phi_0,+,f_1\}_i + \{\psi_0,+,g_1\}_j$. This redex can be reduced by rules 1 and $1'$ resulting in $\{\{\phi_0,+,f_1\}_i+\psi_0,+,g_1\}_j$ or $\{\phi_0+\{\psi_0,+,g_1\}_j,+,f_1\}_i$, respectively.

For univariate CRs, the index of all CRs are the same and rules 1 (and $1'$), 2 (and $2'$), 3 (and $3'$) are not applicable and the TRS for univariate CRs has no critical pairs. Hence, $\mathcal{CR}$ with associativity and commutativity of $+$ and $*$ is confluent for univariate CRs. □

B Completeness of $\mathcal{CR}^+$

The proof of completeness of $\mathcal{CR}^+$ requires the common rules for addition, multiplication, and exponentiation to be added to $\mathcal{CR}^+$.

Theorem B.1 *$\mathcal{CR}^+$ with associativity and commutativity and arithmetic rules is terminating.*

Proof. Clearly, $\mathcal{CR}$ remains terminating with the addition of rules 15–24. □

Theorem B.2 *$\mathcal{CR}^+$ with associativity and commutativity and arithmetic rules is complete for univariate CRs.*

Proof. Since $\mathcal{CR}^+$ is terminating the proof follows from Theorem A.3. The critical pairs involving rules 15–17 are formed by unifying the redex of these rules with a subterm of the redexes of rules 1–11.

Consider the redex $\{\phi_0,+,0\}$ of rule 15.

- The redex of rule 1: $E+\{\phi_0,+,0\}$ forms the critical pair $\langle E+\phi_0, \{E+\phi_0,+,E+0\}\rangle$. The pair has a common reduct found by applying rule 15: $\{E+\phi_0,+,E+0\} = \{E+\phi_0,+,0\} \stackrel{15}{\Rightarrow} E+\phi_0$.

- The redex of rule 2: $E*\{\phi_0,+,0\}$ forms the critical pair $\langle E*\phi_0, \{E*\phi_0,+,E*0\}\rangle$. This pair has a common reduct: $\{E*\phi_0,+,E*0\} = \{E*\phi_0,+,0\} \stackrel{15}{\Rightarrow} E*\phi_0$.

- The redex of Rule 4: $E^{\{\phi_0,+,0\}}$ forms the critical pair $\langle E^{\phi_0}, \{E^{\phi_0},*,E^0\}\rangle$. This pair has a common reduct: $\{E^{\phi_0},*,E^0\} = \{E^{\phi_0},*,1\} \stackrel{16}{\Rightarrow} E^{\phi_0}$.

- The redex of rule 6: $\{\phi_0,+,0\}+\{\psi_0,+,g_1\}$ forms the critical pair $\langle \phi_0+\{\psi_0,+,g_1\}, \{\phi_0+\psi_0,+,0+g_1\}\rangle$. This pair has a common reduct: $\phi_0+\{\psi_0,+,g_1\} \stackrel{1}{\Rightarrow} \{\phi_0+\psi_0,+,g_1\}$ and $\{\phi_0+\psi_0,+,0+g_1\} = \{\phi_0+\psi_0,+,g_1\}$.

- The redex of rule 7: $\{\phi_0,+,0\}*\{\psi_0,+,g_1\}$ forms the critical pair $\langle\phi_0*\{\psi_0,+,g_1\},\{\phi_0*\psi_0,+,\{\phi_0,+,0\}*g_1+\{\psi_0,+,g_1\}*0+0*g_1\}\rangle$. This pair has a common reduct: $\phi_0*\{\psi_0,+,g_1\}\overset{2}{\Rightarrow}\{\phi_0*\psi_0,+,\phi_0*g_1\}$ and $\{\phi_0*\psi_0,+,\{\phi_0,+,0\}*g_1+\{\psi_0,+,g_1\}*0+0*g_1\}=\{\phi_0*\psi_0,+,\{\phi_0,+,0\}*g_1\}\overset{15}{\Rightarrow}\{\phi_0*\psi_0,+,\phi_0*g_1\}$.
- The redex of rule 9: $\{\phi_0,*,f_1\}^{\{\psi_0,+,0\}}$ forms the critical pair $\langle\{\phi_0,*,f_1\}^{\psi_0},\{\phi_0^{\psi_0},*,f_1^{\{\psi_0,+,0\}}*0^{\{\psi_0,+,g_1\}}*0^{g_1}\}\rangle$. This pair has a common reduct: $\{\phi_0,*,f_1\}^{\psi_0}\overset{5}{\Rightarrow}\{\phi_0^{\psi_0},*,f_1^{\psi_0}\}$ and $\{\phi_0^{\psi_0},*,f_1^{\{\psi_0,+,0\}}*0^{\{\psi_0,+,g_1\}}*0^{g_1}\}=\{\phi_0^{\psi_0},*,f_1^{\{\psi_0,+,0\}}\}\overset{15}{\Rightarrow}\{\phi_0^{\psi_0},*,f_1^{\psi_0}\}$.
- The redex of rule 10: $\{\phi_0,+,0\}!$ forms the critical pair $\langle\phi_0!,\{\phi_0!,*,\prod_{j=1}^{0}\{\phi_0+j,+,0\}\}\rangle$. This pair has a common reduct: $\{\phi_0!,*,\prod_{j=1}^{0}\{\phi_0+j,+,0\}\}=\{\phi_0!,*,1\}\overset{16}{\Rightarrow}\phi_0!$.

Consider the redex $\{\phi_0,*,1\}$ of rule 16.

- The redex of rule 3: $E*\{\phi_0,*,1\}$ forms the critical pair $\langle E*\phi_0,\{E*\phi_0,*,1\}\rangle$. This pair has a common reduct: $\{E*\phi_0,*,1\}\overset{16}{\Rightarrow}E*\phi_0$.
- The redex of rule 5: $\{\phi_0,*,1\}^E$ forms the critical pair $\langle\phi_0^E,\{\phi_0^E,*,1^E\}\rangle$. This pair has a common reduct: $\{\phi_0^E,*,1^E\}=\{\phi_0^E,*,1\}\overset{16}{\Rightarrow}\phi_0^E$.
- The redex of rule 8: $\{\phi_0,*,1\}*\{\psi_0,*,g_1\}$ forms the critical pair $\langle\phi_0*\{\psi_0,*,g_1\},\{\phi_0*\psi_0,*,1*g_1\}\rangle$. This pair has a common reduct: $\{\phi_0*\psi_0,*,1*g_1\}=\{\phi_0*\psi_0,*,g_1\}$ and $\phi_0*\{\psi_0,*,g_1\}\overset{3}{\Rightarrow}\{\phi_0*\psi_0,*,g_1\}$.
- The redex of rule 9: $\{\phi_0,*,1\}^{\{\psi_0,+,g_1\}}$ forms the critical pair $\langle\phi_0^{\{\psi_0,+,g_1\}},\{\phi_0^{\psi_0},*,\{\phi_0,*,1\}^{g_1}*1^{\{\psi_0,+,g_1\}}*1^{g_1}\}\rangle$. This pair has a common reduct: $\phi_0^{\{\psi_0,+,g_1\}}\overset{4}{\Rightarrow}\{\phi_0^{\psi_0},*,g_1\}$ and $\{\phi_0^{\psi_0},*,\{\phi_0,*,1\}^{g_1}*1^{\{\psi_0,+,g_1\}}*1^{g_1}\}=\{\phi_0^{\psi_0},*,\{\phi_0,*,1\}^{g_1}\}\overset{16}{\Rightarrow}\{\phi_0^{\psi_0},*,g_1\}$.
- The redex of rule 11: $\log\{\phi_0,*,1\}$ forms the critical pair $\langle\log\phi_0,\{\log\phi_0,+,\log 1\}\rangle$. This pair has a common reduct: $\{\log\phi_0,+,\log 1\}=\{\log\phi_0,+,0\}\overset{15}{\Rightarrow}\log\phi_0$.

Consider the redex $\{0,*,f_1\}$ of rule 17.

- The redex of rule 3: $E*\{0,*,f_1\}$ forms the critical pair $\langle E*0,\{E*0,*,f_1\}\rangle$. This pair has a common reduct: $\{E*0,*,f_1\}=\{0,*,f_1\}\overset{17}{\Rightarrow}0$.
- The redex of rule 5: $\{0,*,f_1\}^E$ forms the critical pair $\langle 0^E,\{0^E,*,f_1^E\}\rangle$. This pair has a common reduct: $\{0^E,*,f_1^E\}=\{0,*,f_1^E\}\overset{17}{\Rightarrow}0$.
- The redex of rule 8: $\{0,*,f_1\}*\{\psi_0,*,g_1\}$ forms the critical pair $\langle 0*\{\psi_0,*,g_1\},\{0*\psi_0,*,f_1*g_1\}\rangle$. This pair has a common reduct: $0*\{\psi_0,*,g_1\}=0$ and $\{0*\psi_0,*,f_1*g_1\}=\{0,*,f_1*g_1\}\overset{17}{\Rightarrow}0$.
- The redex of rule 9: $\{0,*,f_1\}^{\{\psi_0,+,g_1\}}$ forms the critical pair $\langle 0^{\{\psi_0,+,g_1\}},\{0^{\psi_0},*,\{0,*,f_1\}^{g_1}*f_1^{\{\psi_0,+,g_1\}}*f_1^{g_1}\}\rangle$. This pair has a common reduct: $0^{\{\psi_0,+,g_1\}}\overset{4}{\Rightarrow}\{0^{\psi_0},*,0^{g_1}\}=\{0,*,0\}$ and $\{0^{\psi_0},*,\{0,*,f_1\}^{g_1}*f_1^{\{\psi_0,+,g_1\}}*f_1^{g_1}\}=\{0^{\psi_0},*,\{0,*,f_1\}^{g_1}\}\overset{17}{\Rightarrow}\{0^{\psi_0},*,0^{g_1}*f_1^{\{\psi_0,+,g_1\}}*f_1^{g_1}\}=\{0,*,0\}$.

- The redex of rule 11: $\log\{0, *, f_1\}$ forms the critical pair $\langle \log 0, \{\log 0, +, \log f_1\}\rangle$. Both are undefined.

The critical pairs involving the arithmetic rules 18–24 are formed by unifying the redexes of rules 1–11 (and 15–17) with rules 18–24.

- Consider the redex $0 + \{\phi_0, +, f_1\}$ of rules 1 and 18 forming the critical pair $\langle \{0 + \phi_0, +, 0 + f_1\}, \{\phi_0, +, f_1\}\rangle$. This pair has a common reduct: $\{0 + \phi_0, +, 0 + f_1\} = \{\phi_0, +, f_1\}$.
- Consider the redex $0 * \{\phi_0, +, f_1\}$ of rules 2 and 19 forming the critical pair $\langle \{0 * \phi_0, +, 0 * f_1\}, 0\rangle$. This pair has a common reduct: $\{0 * \phi_0, +, 0 * f_1\} = \{0, +, 0\} \stackrel{15}{\Rightarrow} 0$.
- Consider the redex $1*\{\phi_0, +, f_1\}$ of rules 2 and 20 forming the critical pair $\langle \{1*\phi_0, +, 1*f_1\}, \{\phi_0, +, f_1\}\rangle$. This pair has a common reduct: $\{1 * \phi_0, +, 1 * f_1\} = \{\phi_0, +, f_1\}$.
- Consider the redex $0*\{\phi_0, +, f_1\}$ of rules 3 and 19 forming the critical pair $\langle \{0*\phi_0, *, 0*f_1\}, 0\rangle$. This pair has a common reduct: $\{0 * \phi_0, *, 0 * f_1\} = \{0, *, 0\} \stackrel{17}{\Rightarrow} 0$.
- Consider the redex $1*\{\phi_0, *, f_1\}$ of rules 3 and 20 forming the critical pair $\langle \{1*\phi_0, *, 1*f_1\}, \{\phi_0, *, f_1\}\rangle$. This pair has a common reduct: $\{1 * \phi_0, *, 1 * f_1\} = \{\phi_0, *, f_1\}$.
- Consider the redex $0^{\{\phi_0, +, f_1\}}$ of rules 4 and 21 forming the critical pair $\langle \{0^{\phi_0}, *, 0^{f_1}\}, 0\rangle$. This pair has a common reduct: $\{0^{\phi_0}, *, 0^{f_1}\} = \{0, *, 0\} \stackrel{17}{\Rightarrow} 0$.
- Consider the redex $1^{\{\phi_0, *, f_1\}}$ of rules 4 and 22 forming the critical pair $\langle \{1^{\phi_0}, *, 1^{f_1}\}, 1\rangle$. This pair has a common reduct: $\{1^{\phi_0}, *, 1^{f_1}\} = \{1, *, 1\} \stackrel{16}{\Rightarrow} 1$.
- Consider the redex $\{\phi_0, *, f_1\}^0$ of rules 5 and 23 forming the critical pair $\langle \{\phi_0^0, *, f_1^0\}, 0\rangle$. This pair has a common reduct: $\{\phi_0^0, *, f_1^0\} = \{0, *, 0\} \stackrel{17}{\Rightarrow} 0$.

□

C Proof of Correctness of $\mathcal{B}$

We prove the correctness of symbolic function $\mathcal{B}$ by verifying that $\mathcal{F}(\mathcal{B}(\Phi_i)) = \mathcal{B}(\mathcal{F}(\Phi_i)) = \Phi_i$ for any non-indeterminate Φ_i. The proof is with induction on the structure of CR-expression Φ_i. The induction base case is formed by $\Phi_i = v$ variable v or $\Phi_i = c$ constant c:

$$\begin{aligned} \mathcal{F}(\mathcal{B}(v)) &= \; v \; = \mathcal{B}(\mathcal{F}(v)) \\ \mathcal{F}(\mathcal{B}(c)) &= \; c \; = \mathcal{B}(\mathcal{F}(c)) \end{aligned}$$

For $\Phi_i = \{\phi_0, \odot_1, f_1\}_i$, we have

$$\begin{aligned} \mathcal{F}(\mathcal{B}(\{\phi_0, \odot_1, f_1\}_i)) &= \mathcal{F}(\{\phi_0 \odot_1^{-1} \mathcal{V}(\mathcal{B}(f_1)), \odot_1, \mathcal{B}(f_1)\}_i) \\ &= \{(\phi_0 \odot_1^{-1} \mathcal{V}(\mathcal{B}(f_1))) \odot_1 \mathcal{V}(\mathcal{B}(f_1)), \odot_1, \mathcal{F}(\mathcal{B}(f_1))\}_i \\ &= \{\phi_0, \odot_1, \mathcal{F}(\mathcal{B}(f_1))\}_i \\ &= \{\phi_0, \odot_1, f_1\}_i \end{aligned}$$

The last step exploits the induction hypothesis. Conversely, we also find

$$\begin{array}{rcl}\mathcal{B}(\mathcal{F}(\{\phi_0, \odot_1, f_1\}_i)) & = & \mathcal{B}(\{\phi_0 \odot_1 \mathcal{V}(f_1), \odot_1, \mathcal{F}(f_1)\}_i \\ & = & \{(\phi_0 \odot_1 \mathcal{V}(f_1)) \odot_1^{-1} \mathcal{V}(\mathcal{B}(\mathcal{F}(f_1))), \odot_1, \mathcal{B}(\mathcal{F}(f_1))\}_i \\ & = & \{(\phi_0 \odot_1 \mathcal{V}(f_1)) \odot_1^{-1} \mathcal{V}(f_1), \odot_1, f_1\}_i \\ & = & \{\phi_0, \odot_1, f_1\}_i\end{array}$$

where the one-but-last step in the derivation exploits the induction hypothesis. For $\Phi_i = f(\Phi_i^1, \ldots, \Phi_i^k)$ with k-ary function symbol f, we have

$$\begin{array}{rcl}\mathcal{F}(\mathcal{B}(f(\Phi_i^1, \ldots, \Phi_i^k)) & = & \mathcal{F}(f(\mathcal{B}(\Phi_i^1), \ldots, \mathcal{B}(\Phi_i^k)) \\ & = & f(\mathcal{F}(\mathcal{B}(\Phi_i^1)), \ldots, \mathcal{F}(\mathcal{B}(\Phi_i^k))) \\ & = & f(\Phi_i^1, \ldots, \Phi_i^k)\end{array}$$

where the last step in the derivation exploits the induction hypothesis. Conversely, we also find

$$\begin{array}{rcl}\mathcal{B}(\mathcal{F}(f(\Phi_i^1, \ldots, \Phi_i^k)) & = & \mathcal{B}(f(\mathcal{F}(\Phi_i^1), \ldots, \mathcal{F}(\Phi_i^k)) \\ & = & f(\mathcal{B}(\mathcal{F}(\Phi_i^1)), \ldots, \mathcal{B}(\mathcal{F}(\Phi_i^k))) \\ & = & f(\Phi_i^1, \ldots, \Phi_i^k)\end{array}$$

where the last step in the derivation exploits the induction hypothesis.

The axiomatic identity $\phi_0 \odot^{-1} \phi_1 \odot \phi_1 = \phi_0$ is valid in the derivations, because $\mathcal{B}$ is defined for non-indeterminate CRs only, guaranteeing that $\phi_1 \neq 0$ for $\odot = *$ and $\odot^{-1} = /$.

D Proof of Theorem 4.2

The proof is with induction on the structure of CR-expression Φ_i. The induction base case is formed by $\Phi_i = v$ variable v or $\Phi_i = c$ constant c and is trivial:

$$\begin{array}{lcl}\Phi_i = v \geq 0 & \rightarrow & f(x) = v \geq 0 \\ \Phi_i = c \geq 0 & \rightarrow & f(x) = c \geq 0\end{array}$$

where f is the closed-form function of Φ_i.

For $\Phi_i = f(\Phi_i^1, \ldots \Phi_i^k)$ with k-ary function symbol f, we have the trivial implication

$$\Phi_i = f(\Phi_i^1, \ldots \Phi_i^k) \geq 0 \rightarrow f(\Phi_i^1, \ldots \Phi_i^k) \geq 0$$

where the sign of f is determined by the properties of f (see e.g. Fig. 17). For $\Phi_i = \{\phi_0, \odot_1, \phi_1, \ldots, \phi_{k-1} \odot_k, f_k\}_i$ we write Φ_i as a nested tuple $\Phi_i = \{\phi_0, \odot_1, f_1\}_i$ (i.e. in the form of Eq. (3)). By exploiting the induction hypothesis we assume that $f_1 \geq 0$ (since f_1 is a CR-expression). We show that $\phi_0 \geq 0$ implies $f(x) \geq 0$, where f is the closed-form function of Φ_i.

Suppose that $\phi_0 \geq 0$.

- If $\odot_1 = +$ we have $\Phi_i = \phi_0 + \sum_{j=0}^{i-1}(f_1)_j$ by definition of CRs and BRs, such that $f(x) = f(x_0 + i\,h) = \Phi_i$ holds. Since $\phi_0 + \sum_{i=0}^{j}(f_1)_i \geq 0$ for all $j \geq 0$, we conclude that $f(x) \geq 0$.
- If $\odot_1 = *$ we have $\Phi_i = \phi_0 \prod_{j=0}^{i-1}(f_1)_j$ by definition of CRs and BRs, such that $f(x) = f(x_0 + i\,h) = \Phi_i$ holds. Since $\phi_0 \prod_{j=0}^{j-1}(f_1)_j \geq 0$ for all $i \geq 0$, we conclude that $f(x) \geq 0$.