Efficient Symbolic Analysis for Optimizing Compilers*

Robert A. van Engelen

Dept. of Computer Science, Florida State University, Tallahassee, FL 32306-4530
engelen@cs.fsu.edu

Abstract. Because most of the execution time of a program is typically spend in loops, loop optimization is the main target of optimizing and restructuring compilers. An accurate determination of induction variables and dependencies in loops is of paramount importance to many loop optimization and parallelization techniques, such as generalized loop strength reduction, loop parallelization by induction variable substitution, and loop-invariant expression elimination. In this paper we present a new method for induction variable recognition. Existing methods are either ad-hoc and not powerful enough to recognize some types of induction variables, or existing methods are powerful but not safe. The most powerful method known is the symbolic differencing method as demonstrated by the Parafrase-2 compiler on parallelizing the Perfect Benchmarks$^{(R)}$. However, symbolic differencing is inherently unsafe and a compiler that uses this method may produce incorrectly transformed programs without issuing a warning. In contrast, our method is safe, simpler to implement in a compiler, better adaptable for controlling loop transformations, and recognizes a larger class of induction variables.

1 Introduction

It is well known that the optimization and parallelization of scientific applications by restructuring compilers requires extensive analysis of induction variables and dependencies in loops in order for compilers to effectively transform and optimize loops. Because most of the execution time of an application is spend in loops, restructuring compilers attempt to aggressively optimize loops. To this end, many ad-hoc techniques have been developed for loop induction variable recognition [1, 2, 13, 20, 22] for loop restructuring transformations such as *generalized loop strength reduction*, *loop parallelization by induction variable substitution* (the reverse of strength reduction), and *loop-invariant expression elimination* (code motion). However, these ad-hoc techniques fall short of recognizing *generalized induction variables* (GIVs) with values that form polynomial and geometric progressions through loop iterations [3, 7, 8, 11, 19]. The importance of GIV recognition in the parallelization of the Perfect Benchmarks$^{(R)}$ and other codes was recognized in an empirical study by Singh and Hennessy [14].

* This work was supported in part by NSF grant CCR-9904943

The effectiveness of GIV recognition in the actual parallelization of the Perfect Benchmarks$^{(R)}$ was demonstrated by the Parafrase-2 compiler [10]. Parafrase-2 uses Haghighat's symbolic differencing method [10] to detect GIVs. Symbolic differencing is the most powerful induction variable recognition method known.

In this paper we show that symbolic differencing is an unsafe compiler method when not used wisely (i.e. without a user verifying the result). As a consequence, a compiler that adopts this method may produce incorrectly transformed programs without issuing a warning. We present a new induction variable recognition method that is safe and simpler to implement in a compiler and recognizes a larger class of induction variables.

This paper is organized as follows: Section 2 compares our approach to related work. Section 3 presents our generalized induction variable recognition method. Results are given in Section 4. Section 5 summarizes our conclusions.

2 Related Work

Many ad-hoc compiler analysis methods exist that are capable of recognizing *linear induction variables*, see e.g. [1, 2, 13, 20, 22]. Haghighat's symbolic differencing method [10] recognizes *generalized induction variables* [3, 7, 8, 11, 19] that form polynomial and geometric progressions through loop iterations. More formally, a GIV is characterized by its function χ defined by

$$\chi(n) = \varphi(n) + r\,a^n \tag{1}$$

where n is the loop iteration number, φ is a polynomial of order k, and a and r are loop-invariant expressions.

Parallelization of a loop containing GIVs with (multiple) update assignment statements requires the removal of the updates and the substitution of GIVs in expressions by their closed-form characteristic function χ. This is also known as *induction variable substitution*, which effectively removes all cross-iteration dependencies induced by GIV updates, enabling a loop to be parallelized.

The symbolic differencing method is illustrated in Fig. 1. A compiler symbolically evaluates a loop a fixed number of iterations using *abstract interpretation*. The sequence of symbolic values of each variable are tabulated in *difference tables* from which polynomial and geometric progressions can be recognized. To recognize polynomial GIVs of degree m, a loop is executed at most $m + 2$ iterations. For example, for $m = 3$, the compiler executes the loop shown in Fig. 1(a) five times and constructs a difference table for each variable. The difference table of variable `t` is depicted in Fig. 1(b) above the dashed line. According to [10], the compiler can determine that the polynomial degree of `t` is $m = 3$, because the final difference is zero (bottom-most zero above the dashed line). Application of induction variable substitution by Parafrase-2 results in the loop Fig. 1(c).

The problem with this approach is that the analysis is incorrect when m is set too low. Six or more iterations are necessary (this includes the diagonal below the dashed line) to find that the degree of `t` is actually 5. Hence, the loop shown in Fig. 1(c) is incorrect. Fig. 1(d) depicts the correctly transformed loop.

```
DO i=0,n
 t=a
 a=b
 b=c+2*b-t+2
 c=c+d
 d=d+i
ENDDO
```

(a)

a		b		2b+c-a+2		3b+3c+d-2a+6		4b+6c+4d-3a+12		5b+10c+10d-4a+21
	b-a		b+c-a+2		b+2c+d-a+4		b+3c+3d-a+6		b+4c+6d-a+9	
		c+2		c+d+2		c+2d+2		c+3d+3		
			d		d		d+1			
				0		1				
					1					

(b)

```
DO i=0,n
 t=a+(d*i**3
 +(3*c-3*d+6)*i**2
 +(6*b-9*c+2*d-6)*i)/6
ENDDO
```

(c)

```
DO i=0,n
 t=a+(i**5-10*i**4
 +(20*d+35)*i**3
 +(60*c-60*d+70)*i**2
 +(120*b-180*c+40*d-96)*i)/120
ENDDO
```

(d)

Fig. 1. Example Loop (a), Difference Table of `t` (b), Incorrectly Transformed Loop (c), and Correctly Transformed Loop (d)

The origin of the problem with symbolic differencing lies in the fact that difference tables form low-order approximations of higher-order polynomials and functions. Hence, symbolic differencing requires a user of the compiler to specify a maximum polynomial order m that is *guaranteed* to be the largest order among all of the known GIVs in a program. This means that the user must be knowledgeable about the type of algorithms in the program to make a wise decision. The optimization of the Perfect Benchmarks$^{(\mathrm{R})}$ by Parafrase-2 required $m = 3$ to produce correctly transformed programs. Many real-world applications exist that use higher degree polynomials such as in transfinite interpolation for grid generation [12] for CFD and CAD applications, and in hard-coded curve plotting and surface rendering applications, see e.g. [4].

According to [10], symbolic differencing requires extensive symbolic expression manipulation. A symbolic kernel is required with a concise set of algebraic/symbolic operations derived from mathematical tools such as number theory and mathematical induction for performing symbolic operations. The implementation is further hampered by the fact that difference tables can contain large and complicated symbolic entries. In contrast, our method relies on the use of a term rewriting system with 14 rules to derive normal forms for GIVs and 21 rules to derive the closed-form functions from these forms. Our approach requires symbolic manipulation equally powerful as classical constant-folding[1].

In comparing our method to symbolic differencing we find that both methods can handle *multiple assignments to induction variables*, *generalized induction variables in loops with symbolic bounds and strides*, *symbolic integer division*, *conditional induction expressions*, *cyclic induction dependencies*, *symbolic forward substitution*, *symbolic loop-invariant expressions*, and *wrap-around variables*. However, our method is not capable of detecting *cyclic recurrences*. We found that cyclic recurrences are very rare in the Perfect Benchmarks$^{(\mathrm{R})}$.

[1] Assuming that symbolic manipulation by constant-folding includes associativity, commutativity, and distributivity of product and addition.

3 Generalized Induction Variable Recognition

In this section we present our induction variable recognition method and associated compiler algorithms. First, we introduce the chains of recurrences formalism that forms the mathematical basis of our approach.

3.1 Chains of Recurrences

Chains of recurrences (CRs) are developed by Bachmann, Zima, and Wang [5] to expedite the evaluation of closed-form functions and expressions on regular grids. The CR algebra enables the construction and simplification of recurrence relations by a computer algebra system. An elementary (scalar) expression can be symbolically transformed into its mathematically equivalent CR [4]. The CR provides a representation that allows it to be translated into a loop construct which efficiently evaluates the expression on a regular grid, similar to loop strength reduction of the original single elementary expression.

Basic Formulation A closed-form function f evaluated in a loop with loop counter variable i can be rewritten into a mathematical equivalent *System of Recurrence Relations* (SSR) [21] $f_0(i), f_1(i), \ldots, f_k(i)$, where the functions $f_j(i)$ for $j = 0, \ldots, k-1$ are linear recurrences of the form

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

with $\odot_{j+1} \in \{+, *\}$, $j = 0, \ldots, k-1$, and coefficients ϕ_j are loop-invariant expressions (i.e. induction variables do not occur in ϕ_j). Expression f_k is loop invariant or a similar recurrence system. When the loop is normalized, i.e. $i = 0, \ldots, n$ for some $n \geq 0$, it can be shown that $f(i) = f_0(i)$ for all $i = 0, \ldots, n$.

A shorthand notation for Eq. (2) is a *Basic Recurrence* (BR) [5]:

$$f_j(i) = \{\phi_j, \odot_{j+1}, f_{j+1}\}_i \tag{3}$$

The BR notation allows the system (2) to be written as

$$\Phi_i = \{\phi_0, \odot_1, \{\phi_1, \odot_2, \cdots, \{\phi_{k-1}, \odot_k, f_k\}_i\}_i\}_i \tag{4}$$

When flattened to a single tuple

$$\Phi_i = \{\phi_0, \odot_1, \phi_1, \odot_2, \cdots, \odot_k, f_k\}_i \tag{5}$$

it is a *Chain of Recurrences* (CR) [4, 5] with $k = L(\Phi_i)$ the length of the CR.

A CR Φ_i is called *polynomial* if $\odot_j = +$, for all $j = 1, \ldots, k$. A polynomial CR has a closed-form function that is a k-order polynomial in variable i. The sequence $\phi_0, \phi_1, \ldots, \phi_{k-1}, f_k$ forms the lower-left diagonal of a difference table of the polynomial. CR Φ_i is called *exponential* if $\odot_j = *$, for all $j = 1, \ldots, k$. CR $\Phi_i = \{\phi_0, *, \phi_1, +, f_2\}_i$ is called *factorial* if $\phi_1 \geq 1$ and $f_2 = 1$, or $\phi_1 \leq -1$ and $f_2 = -1$.

CR Construction Fig. 2 depicts $\mathcal{CR}$ rewrite rules adapted from the CR algebra presented in [4, 5]. The proof of correctness of the algebra can be found in [4]. CR construction proceeds by replacing every occurrence of the loop counter variable i (i.e. the basic induction variable) in an expression by the CR $\{a, +, s\}_i$, where a is i's symbolic initial value and s is the stride. Then, $\mathcal{CR}$ rules are exhaustively applied to the expression[2]. In [16] we proved that $\mathcal{CR}$ is complete (i.e. confluent and terminating) and, hence, CRs are normal forms for polynomials, exponentials, and factorials.

The exhaustive application of $\mathcal{CR}$ results in so-called *CR-expressions*, which are expressions that *contain* CRs as subexpressions[3]. The normalized CR expression of a GIV with characteristic function Eq. (1) is the sum of a polynomial CR and the exponential CR $\{r, *, a\}_n$.

3.2 The Algorithms

We developed a compiler algorithm for induction variable recognition. The algorithm is capable of detecting multiple induction variables in loop hierarchies by exploiting the CR algebra in an entirely novel way. The induction variable recognition method forms the basis of our induction variable substitution algorithm. Induction variable substitution amounts to the removal of induction variable update operations and the replacement of the induction variables by their closed-forms. This requires the inverse rules $\mathcal{CR}^{-1}$ shown in Fig. 3 which we developed to translate CRs back to closed-form functions.

The principle of our algorithm is illustrated in the program fragment below, which demonstrates the key idea to our induction variable recognition method and induction variable substitution:

`DO i=0,n`
`...`
`j=j+h`
`k=k+2*i`
`ENDDO`
	$\Rightarrow$	`DO i=0,n`
		`...`
		`j=j+h`
		`k=k+`$\{0,+,2\}_i$
		`ENDDO`
			$\Rightarrow$	`DO i=0,n`
				`...`
				`j=`$\{\mathtt{j}_0,+,\mathtt{h}\}_i$
				`k=`$\{\mathtt{k}_0,+,0,+,2\}_i$
				`ENDDO`
					$\Rightarrow$	$\mathtt{j}_0$`=j; `$\mathtt{k}_0$`=k`
						`DOALL i=0,n`
						`j=`$\mathtt{j}_0$`+h*i`
						`k=`$\mathtt{k}_0$`+i*i-i`
						`...`
						`ENDDO`

First, loop counter variable `i` is replaced by its CR representation $\{0, +, 1\}_i$ upon which rule 2 of $\mathcal{CR}$ Fig. 2 translates `2*i` into the CR $\{0, +, 2\}_i$. Then, linear induction variable `j` and non-linear induction variable `k` are recognized by our algorithm from their update operations and replaced by CRs $\{\mathtt{j}_0, +, \mathtt{h}\}_i$ and $\{\mathtt{k}_0, +, 0, +, 2\}_i$, where $\mathtt{j}_0$ and $\mathtt{k}_0$ are initial values of `j` and `k` before the loop. Finally, the $\mathcal{CR}^{-1}$ rules are applied to substitute the CR of `j` and `k` with their closed-forms, enabling the loop to be parallelized. The advantage of our approach is that simple algebraic rules are exploited to modify the code from one form into another.

[2] Applied together with constant folding to simplify CR coefficients.

[3] When CRs are mentioned in this text we refer to pure CRs of the form Eq. (5). CR-expressions will be indicated explicitly.

	LHS	RHS	
1	$E + \{\phi_0, +, f_1\}_i$	$\Rightarrow \{E + \phi_0, +, f_1\}_i$	when E is loop invariant
2	$E * \{\phi_0, +, f_1\}_i$	$\Rightarrow \{E * \phi_0, +, E * f_1\}_i$	when E is loop invariant
3	$E * \{\phi_0, *, f_1\}_i$	$\Rightarrow \{E * \phi_0, *, f_1\}_i$	when E is loop invariant
4	$E^{\{\phi_0, +, f_1\}_i}$	$\Rightarrow \{E^{\phi_0}, *, E^{f_1}\}_i$	when E is loop invariant
5	$\{\phi_0, *, f_1\}_i^E$	$\Rightarrow \{\phi_0^E, *, f_1^E\}_i$	when E is loop invariant
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	$\log\{\phi_0, *, f_1\}_i$	$\Rightarrow \{\log \phi_0, +, \log f_1\}_i$	
12	$\{\phi_0, +, 0\}_i$	$\Rightarrow \phi_0$	
13	$\{\phi_0, *, 1\}_i$	$\Rightarrow \phi_0$	
14	$\{0, *, f_1\}_i$	$\Rightarrow 0$	

Fig. 2. $\mathcal{CR}$

	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 does not occur in f_1
6	$\{0, +, \log f_1\}_i$	$\Rightarrow \log\{1, *, f_1\}_i$	
7	$\{0, +, f_1^i\}_i$	$\Rightarrow \frac{f_1^{i+1} - 1}{f_1 - 1}$	when i does not occur in f_1 and $f_1 \neq 1$
8	$\{0, +, f_1^{g_1 + h_1}\}_i$	$\Rightarrow \{0, +, f_1^{g_1} * f_1^{h_1}\}_i$	
9	$\{0, +, f_1^{g_1 * h_1}\}_i$	$\Rightarrow \{0, +, (f_1^{g_1})^{h_1}\}_i$	when i does not occur in f_1 and g_1
10	$\{0, +, f_1\}_i$	$\Rightarrow i * f_1$	when i does not occur in f_1
11	$\{0, +, i\}_i$	$\Rightarrow \frac{i^2 - i}{2}$	
12	$\{0, +, i^n\}_i$	$\Rightarrow \sum_{k=0}^{n} \frac{\binom{n+1}{k}}{n+1} B_k\, i^{n-k+1}$	for $n \in \mathbb{N}$, B_k is k^{th} Bernoulli number
13	$\{1, *, -f_1\}_i$	$\Rightarrow (-1)^i \{1, *, f_1\}_i$	
14	$\{1, *, \frac{1}{f_1}\}_i$	$\Rightarrow \{1, *, f_1\}_i^{-1}$	
15	$\{1, *, f_1 * g_1\}_i$	$\Rightarrow \{1, *, f_1\}_i * \{1, *, g_1\}_i$	
16	$\{1, *, f_1^{g_1}\}_i$	$\Rightarrow f_1^{\{1, *, g_1\}_i}$	when i does not occur in f_1
17	$\{1, *, g_1^{f_1}\}_i$	$\Rightarrow {\{1, *, g_1\}_i}^{f_1}$	when i does not occur in f_1
18	$\{1, *, f_1\}_i$	$\Rightarrow f_1^i$	when i does not occur in f_1
19	$\{1, *, i\}_i$	$\Rightarrow 0^i$	
20	$\{1, *, i + f_1\}_i$	$\Rightarrow \frac{(i + f_1 - 1)!}{(f_1 - 1)!}$	when i does not occur in f_1 and $f_1 \geq 1$
21	$\{1, *, f_1 - i\}_i$	$\Rightarrow (-1)^i * \frac{(i - f_1 - 1)!}{(-f_1 - 1)!}$	when i does not occur in f_1 and $f_1 \leq -1$

Fig. 3. $\mathcal{CR}^{-1}$

$IVS(S)$
- input: statement list S
- output: *Induction Variable Substitution* applied to S
CALL $IVStrans(S)$
Use reaching flow information to propagate initial variable values to the CRs in S
Apply $\mathcal{CR}^{-1}$ to every CR in S
For every left-over CR Φ_i in S, create an index array ia and code K to initialize the array using algorithm $CRGEN(\Phi_i, b, ia, K)$, where b is the upper bound of the index array which is the maximum value of the induction variable i of the loop in which Φ_i occurs

---

$IVStrans(S)$
- input: statement list S
- output: *Induction Variable Substitution* applied to S, where CR expressions in S represent the induction expressions of loops in S

```
FOR each do-loop L in statement list S DO
      Let S(L) denote the body statement list of loop L, let I denote the basic induction variable
         of the loop with initial value expression a, bound expression b, and stride expression s
      CALL IVStrans(S(L))
      TRY
            CALL SSA*(S(L), A)
            CALL CR(I, a, s, S(L), A)
            CALL HOIST(I, a, b, s, S(L), A)
      CATCH FAIL:
            Continue with next do-loop L in S
```

Fig. 4. Algorithm IVS

Algorithm IVS shown in Fig. 4 applies induction variable substitution to a set of nested loops in a statement list. The programming model supported by IVS includes sequences, assignments, if-then-elses, do-loops, and while-loops. IVS analyzes a do-loop nest (not necessarily perfectly nested) from the innermost loops, which are the primary candidates for optimization, to the outermost loops. For every loop in the nest, the body is converted to a single-static assignment (SSA) form with assignments to scalar variables separated from the loop body and stored in set A. Next, algorithm CR converts the expressions in A to CR-expressions to detect induction variables. Algorithm $HOIST$ hoists the induction variable update assignments out of the loop and replaces induction expressions in the loop by closed-forms. If an induction variable has no closed-form, algorithm $CRGEN$ is used to create an index array that will serve as a closed-form. This enables IVS for a larger class of induction variables compared to GIVs defined by Eq. (1). FAIL indicates failure of the algorithm to apply IVS, which can only be due to failure of SSA^* (see SSA^* description later).

The worst-case computational complexity of IVS is $\mathcal{O}(k\,n\,\log(n)\,m^2)$, where k is the maximum loop nesting level, n is the length of the source code fragment, and m is the maximum polynomial order of the GIVs in the fragment. This bound is valid provided that Bachmann's CR construction algorithm [4] is used for the fast construction of polynomial CRs.

Note that the $\mathcal{CR}$ rules in Fig. 2 are applicable to both integer and floating-point typed expressions. Rule 11 handles floating point expressions only. It is guaranteed that integer-valued induction variables have CRs with integer-valued coefficients. However, some integer-valued expressions in a loop that *contain* induction variables may be converted into CRs with rational CR coefficients and a compiler implementation of the rules must handle rationals appropriately.

```
SSA*(S, A)
- input: loop body statement list S
- output: SSA-modified S and partially ordered set A, or FAIL
A:=∅
FOR each statement S_i ∈ S from the last (i = |S|) to the first statement (i = 1) DO
    CASE S_i
    OF assignment statement of expression X to V:
        IF V is a numeric scalar variable and (V,⊥) ∉ A THEN
            FOR each statement S_j ∈ S, j = i+1,...,|S| DO
                Substitute in S_j every occurrence of variable V by a pointer to X
            FOR each (U,Y) ∈ A DO
                Substitute in Y every occurrence of variable V by a pointer to X
            IF (V,_) ∉ A THEN /* note: _ is a wildcard */
                A:=A ∪ {(V,X)}
            Remove S_i from S
        ELSE /* assignment to non-numeric or non-scalar variable */
            Continue with next statement S_i
    OF if-then-else statement with condition C, then-clause S(T), and else-clause S(E):
        CALL SSA*(S(T), A_1)
        CALL SSA*(S(E), A_2)
        CALL MERGE(C, A_1, A_2, A_1,2)
        FOR each (V,X) ∈ A_1,2 DO
            FOR each statement S_j, j = i+1,...,|S| DO
                Substitute in S_j every occurrence of variable V by a pointer to X
            FOR each (U,Y) ∈ A DO
                Substitute in Y every occurrence of V by a pointer to X
            IF (V,_) ∉ A THEN /* note: _ is a wildcard */
                A:=A ∪ {(V,X)}
    OF do-loop OR while-loop:
        IF the loop body contains an assignment to a scalar numeric variable V THEN
            A:=A ∪ (V,⊥)
Topologically sort A with respect to ≺, FAIL if sort not possible (i.e. ≺ is not a partial order on A)
```

```
MERGE(C, A_1, A_2, A_1,2)
- input: Boolean expression C, variable-expression sets A_1 and A_2
- output: merged set A_1,2
A_1,2:=∅
FOR each (V,X) ∈ A_1 DO
    IF (V,Y) ∈ A_2 for some expression Y THEN
        A_1,2:=A_1,2 ∪ {(V, C?X:Y)}
    ELSE
        A_1,2:=A_1,2 ∪ {(V, C?X:V)}
FOR each (V,X) ∈ A_2 DO
    IF (V,_) ∉ A_1,2 THEN
        A_1,2:=A_1,2 ∪ {(V, C?V:X)}
```

Fig. 5. Algorithm SSA^*

Single Static Assignment Form The SSA^* algorithm shown in Fig. 5 uses the precedence relation on variable-expression pairs defined by

$$(U, Y) \prec (V, X) \qquad \text{if } U \neq V \text{ and } V \text{ occurs in } Y$$

to obtain an ordered set of variable-expression pairs A extracted from a loop body. The algorithm constructs a directed acyclic graph for the expressions in A and the expressions in the loop body such that common-subexpressions share the same node in the graph to save space and time. Thus, when a subexpression is transformed by rules $\mathcal{CR}$ or $\mathcal{CR}^{-1}$ the result is immediately visible to the expressions that refer to this subexpression. This works best when the rules $\mathcal{CR}$ and $\mathcal{CR}^{-1}$ are applied to an expression from the innermost to the outermost redexes (i.e. normal-order reduction).

```
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)

$\Rightarrow$

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

(2)

$\Rightarrow$

```
DO i=1,n
 IF cond({1,+,1}_i) THEN
  DO j=0,{0,+,1}_i
   a[{1,+,1}_i,{1,+,1}_j]
    =b[{k+1,+,2,+,1}_j]
  ENDDO
 ENDIF
 k=(cond(i)?{k,+,1,+,2,+,1}_i
          : {k,+,1,+,2,+,1}_i)
ENDDO
```

(3)

$\Rightarrow$

```
DO i=0,n-1
 IF cond(i+1) THEN
  DO j=0,i
   a[i+1,j+1]
   =b[k+(i**3+2*i+3*((i+j)**2-2*i*j)+9*j)/6+1]
  ENDDO
 ENDIF
ENDDO
```

(4)

Fig. 6. Example Analysis of Conditional Induction Variables

Each variable has only one expression in A. Set A contains the potential induction variables of the loop. Values of conditional induction variables are represented by conditional expressions of the form $C?X\!:Y$, where C is the condition and X and Y are expressions. The conditional expression $C?X\!:X$ is rewritten[4] into X. An example conditional induction variable analysis is shown in Fig. 6. For sake of simplicity, we assume that the values of conditions do not change when the conditions are replicated to different parts of the loop body. The algorithm can be easily modified to produce code in which a temporary variable is used to hold the value of a condition when conditions cannot be replicated.

Algorithm SSA^* fails when $\prec$ is not a partial order on A. The source of this problem is the presence of cyclic recurrences in a loop, see e.g. [10] (p. 29). Cyclic dependencies are not a problem.

Induction Variable Recognition and Substitution Algorithm CR shown in Fig. 7 converts the expressions in A into normalized CR-expressions to detect induction variables.

Fig. 8 illustrates the analysis of *wrap-around variables* by algorithm CR. The code example is from [10] (p. 52). A wrap-around 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. 8(1). For wrap-around variable analysis, each use of the variable V before its definition is replaced by the CR-expression

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

[4] Recall that CRs are normal forms, and when X is a CR this rewrite rule is trivial to implement.

$CR(I, a, s, S, A)$
- input: loop induction variable I with initial value a and stride s,
statement list S, and topologically ordered set S of variable-expression pairs
- output: expressions in A are converted to CR-expressions

FOR each $(V, X) \in A$ in topological order ($\prec$) DO
  Substitute in X every occurrence of I by $\{a, +, s\}_I$
  Apply $\mathcal{CR}$ rules to X
  IF X is of the form $V + C$, where C is a loop invariant expression or a CR THEN
    Replace (V, X) in A with $(V, \{V, +, C\}_I)$
    FOR each $(U, Y) \in A$, $(V, X) \prec (U, Y)$ DO
      Substitute every occurrence of V in Y by $\{V, +, C\}_I$
  ELSE IF X is of the form $V * C$, where C is a loop invariant expression or a CR THEN
    Replace (V, X) in A with $(V, \{V, *, C\}_I)$
    FOR each $(U, Y) \in A$, $(V, X) \prec (U, Y)$ DO
      Substitute every occurrence of V in Y by $\{V, *, C\}_I$
  ELSE IF V does not occur in X THEN /* **wrap-around variable** */
    Replace (V, X) in A with $(V, \{V - \mathcal{V}(\mathcal{B}(X)), *, 0\}_I + \mathcal{B}(X))$
    FOR each $(U, Y) \in A$, $(V, X) \prec (U, Y)$ DO
      Substitute every occurrence of V in Y by $\{V - \mathcal{V}(\mathcal{B}(X)), *, 0\}_I + \mathcal{B}(X)$
  ELSE /* **other type of assignment** */
    Continue with next (V, X) pair in A
FOR each $(V, X) \in A$ in topological order ($\prec$) DO
  FOR each statement $S_i \in S$ (and statements at deeper nesting levels) DO
    Substitute every occurrence of V in S_i by X
    Apply $\mathcal{CR}$ rules to every expression in S_i

Fig. 7. Algorithm CR

(1)
```
j=m
DO i=m,n
 a[j]=b[k]
 j=i+1
 k=j
ENDDO
```
$\Rightarrow$

(2)
j=m
DO i=m,n
 a[j]=b[k]
 j=$\{j - m, *, 0\}_i$ $+\{m, +, 1\}_i$
 k=$\{k - m, *, 0\}_i$ $+\{m, +, 1\}_i$
ENDDO

$\Rightarrow$

(3)
j=m
DO i=0,n-m
 a[$\{\{j - m, *, 0\}_i$ $+\{m, +, 1\}_i$]
 =b[$\{k - m, *, 0\}_i$ $+\{m, +, 1\}_i$]
ENDDO

$\Rightarrow$

(4)
```
DO i=0,n-m
 a[i+m]=b[0**i*(k-m)+i+m]
ENDDO
```

Fig. 8. Example Analysis of Wraparound Variables

where Φ_i is the CR-expression of the variable V. The symbolic functions $\mathcal{V}$ and $\mathcal{B}$ are described in [16] with their proof of correctness. The closed-form of the CR $\{\phi_0, *, 0\}_i$ is $\phi_0 * 0^i$, which evaluates in Fortran to ϕ_0 if $i = 0$ and 0 if $i \neq 0$.

Algorithm $HOIST$ shown in Fig. 9 applies the final stage by hoisting induction variable update assignments out of the loop. It is assumed that $a \leq b - 1$ for lower bound a and upper bound b of each loop counter variable with positive stride $s > 0$, and $a \geq b - 1$ for $s < 0$. When this constraint is not met for a particular loop, the same approach as in [10] (p. 82) is used in which the loop counter variable `i` is replaced with $\max(\lfloor(b - a + s)/s\rfloor, 0)$ in the closed-form of an induction variable to set the final value of the variable at the end of the loop.

Induction Variables and Loop Strength Reduction Polynomial, factorial, GIV, and exponential CR-expressions can always be converted to a closed-form. However, some types of CRs do not have equivalent closed-forms. To extend our approach beyond traditional GIV recognition, we use algorithm $CRGEN$

$HOIST(I, a, b, s, S, A)$
- input: loop induction variable I with initial value a, bound b, stride s, loop statement list S, and A the set of variable-expression pairs
- output: loop S with induction variable assignments hoisted out of S
$T := \emptyset$
FOR each $(V, X) \in A$ in reversed topological order ($\prec$) DO
    Apply $\mathcal{CR}^{-1}$ to X resulting in Y
    IF V does not occur in Y THEN
        IF V is live at the end of the loop THEN
            Substitute every occurrence of I in Y by $\lfloor \frac{b-a+s}{s} \rfloor$
            Append $V := Y$ at the end of T
    ELSE
        Append $V := X$ at the end of S
Replace the statement list S with a loop with body S and followed by statements T:
$S :=$ (DO I=0, $\lfloor \frac{b-a}{s} \rfloor$ S ENDDO T)

Fig. 9. Algorithm *HOIST*

$CRGEN(\Phi_i, b, id, S)$
- input: CR $\Phi_i = \{\phi_0, \odot_1, \ldots, \odot_k, f_k\}_i$, bound expression $b \geq 0$, and identifier id
- output: statement list S to numerically evaluate Φ_i storing the values in $id[0..b]$
S is the statement list created from the template below, where cr_j, $j = 1, \ldots, k$, are temporary scalar variables of type integer if all ϕ_j are integer, float otherwise:
  $id[0] = \phi_0$
  $cr_1 = \phi_1$
    ⋮
  $cr_k = f_k$
  DO $i = 0, b-1$
    $id[i+1] = id[i] \odot_1 cr_1$
    $cr_1 = cr_1 \odot_2 cr_2$
      ⋮
    $cr_{k-1} = cr_{k-1} \odot_k cr_k$
  ENDDO

Fig. 10. Algorithm *CRGEN*

adopted from algorithm CREval [4] (see [4] for the proof of correctness of the algorithm). Algorithm *CRGEN* shown in Fig. 10 stores CR values in an array. *CRGEN* can be used to evaluate CRs that have or do not have closed-forms.

Algorithm *CRGEN* can be used for *generalized loop strength reduction* as well, to replace GIVs by recurrences that are formed by iterative updates to induction variables. For example, the CR of the closed-form expression `(i*i-i)/2` is $\{0, +, 0, +, 1\}_i$, assuming that loop index variable `i` starts with zero and has stride one. The program fragment S produced by $CRGEN(\{0, +, 0, +, 1\}_i, \mathtt{n}, \mathtt{k}, S)$ calculates `(i*i-i)/2` with a strength reduced loop (in a slightly different form):

```
k=0
cr1=0
DO i=0,n-1
   k=k+cr1
   cr1=cr1+1
ENDDO
```

which is equivalent to

```
k=0
DO i=0,n-1
   /* k≡(i*i-i)/2 */
   k=k+i
ENDDO
```

This approach has an important advantage compared to symbolic differencing for generalized loop strength reduction. It automatically handles cases in which a loop body contains non-differentiable operators with arguments that are induction expressions. For example, suppose that `MAX((i*i-i)/2,0)` occurs in a loop body and assume that `i` is some induction variable. Our method recognizes `(i*i-i)/2` automatically as an induction expression which, for example, can be loop strength reduced by replacing `(i*i-i)/2` with a new induction variable. The symbolic differencing method is expensive to use for generalized loop strength reduction to detect subexpressions that are optimizable, because symbolic difference tables have to be constructed for *each* subexpression.

Interprocedural Analysis Due to limitations in space, the presentation of the IVS algorithm in this paper does not include interprocedural analysis. Subroutine inlining can be used but this may result in an explosion in code size. Instead, interprocedural analysis can be performed with algorithm IVS by treating the subroutine call as a jump to the routine's code and back, and by doing some work at the subroutine boundaries to ensure a proper passing of induction variables.

4 Results

In this section we give the results of applying the IVS algorithm to code segments of MDG and TRFD. We created a prototype implementation of IVS in CTADEL [17, 18] to translate MDG, TRFD, and the other code fragments shown in this paper. The MDG code segment is part of a predictor-corrector method in an N-body molecular dynamics application. The TRFD program has been extensively analyzed in the literature, see e.g. [9, 10], because its main loop is hard to parallelize due to the presence of a number of coupled non-linear induction variables. Performance results on the parallelization of MDG and TRFD with IVS are presented in [10]. We will not present an empirical performance comparison in this paper. Instead, we demonstrate a step-by-step application of the GIV recognition method and IVS algorithm on MDG and TRFD.

MDG Fig. 11(2) depicts the recognition of the `ikl` and `ji` GIV updates represented by CRs. Fig. 11(3) shows them hoisted out of the inner loop. This is followed by an analysis of the middle loop which involves forward substitution of the assignments `ji=jiz` and `ikl=ik+m`. Again, `ikl` is recognized as an induction variable of the middle loop together with `ik`, while `ji` is loop invariant with respect to `k`. Hoisting of these variable assignments results in Fig. 11(5). After substitution of the `ik` and `jiz` induction variables in Fig. 11(5), the IVS translated code is shown in Fig. 11(6). Note that arrays `c` and `v` in the inner loop in Fig. 11(5) are indexed by *nested* CRs. This form is automatically obtained, in which the CR coefficients of an outer CR may consist of CRs that are guaranteed to be loop invariant with respect to the outer CR's index variable.

TRFD The application of IVS on TRFD is shown in Fig. 12. The inner loop is analyzed first from which GIVs are eliminated resulting in Fig. 12(2). Working further outwards, subsequent GIVs are eliminated resulting in Fig. 12(3) and (4). The fully IVS transformed TRFD fragment is shown in Fig. 12(5). The complete loop nest can be parallelized on e.g. a shared-memory multiprocessor machine.

```
ik=1
jiz=2
DO i=1,n
 DO k=1,m
  ji=jiz
  ikl=ik+m
  s=0.0
  DO l=i,n
   s=s+c[ji]*v[ikl]
   ikl=ikl+m
   ji=ji+1
  ENDDO
  v[ik]=v[ik]+s
  ik=ik+1
 ENDDO
 jiz=jiz+n+1
ENDDO
```

(1)

⇒

```
ik=1
jiz=2
DO i=1,n
 DO k=1,m
  ji=jiz
  ikl=ik+m
  s=0.0
  DO l=i,n
   s=s+c[{ji,+,1}_l]
     *v[{ikl,+,m}_l]
   ikl={ikl,+,m}_l
   ji={ji,+,1}_l
  ENDDO
  v[ik]=v[ik]+s
  ik=ik+1
 ENDDO
 jiz=jiz+n+1
ENDDO
```

(2)

⇒

```
ik=1
jiz=2
DO i=1,n
 DO k=1,m
  ji=jiz
  ikl=ik+m
  s=0.0
  DO l=0,n-i
   s=s+c[{ji,+,1}_l]
     *v[{ikl,+,m}_l]
  ENDDO
  ikl=ikl+(n-i+1)*m
  ji=ji+n-i+1
  v[ik]=v[ik]+s
  ik=ik+1
 ENDDO
 jiz=jiz+n+1
ENDDO
```

(3)

⇒

```
ik=1
jiz=2
DO i=1,n
 DO k=1,m
  s=0.0
  DO l=0,n-i
   s=s+c[{jiz,+,1}_l]
     *v[{{ik+m,+,1}_k
        ,+,m}_l]
  ENDDO
  v[{ik,+,1}_k]
    =v[{ik,+,1}_k]+s
  ikl={ik+m+m
     *(n-i+1),+,1}_k
  ji=jiz+n-i+1
  ik={ik,+,1}_k
 ENDDO
 jiz=jiz+n+1
ENDDO
```

(4)

⇒

```
ik=1
jiz=2
DO i=1,n
 DO k=0,m-1
  s=0.0
  DO l=0,n-{1,+,1}_i
   s=s+c[{{jiz,+,n+1}_i,+,1}_l]
     *v[{{{ik+m,+,m}_i,+,1}_k
        ,+,m}_l]
  ENDDO
  v[{{ik,+,m}_i,+,1}_k]
    =v[{{ik,+,m}_i,+,1}_k]+s
 ENDDO
 ik={ik,+,m}_i
 jiz={jiz,+,n+1}_i
ENDDO
```

(5)

⇒

```
DO i=0,n-1
 DO k=0,m-1
  s=0.0
  DO l=0,n-i-1
   s=s+c[l+i*(n+1)+2]
     *v[k+m*(i+l+1)+1]
  ENDDO
  v[k+i*m+1]=v[k+i*m+1]+s
 ENDDO
ENDDO
```

(6)

Fig. 11. Algorithm IVS Applied to Code Segment of MDG

5 Conclusions

In this paper we presented a novel approach to generalized induction variable recognition for optimizing compilers. We have shown that the method can be used for generalized induction variable substitution, generalized loop strength reduction, and loop-invariant expression elimination. In contrast to the symbolic differencing method, our method is safe and can be implemented with as little effort as adding a compiler phase that includes rewrite rules for chains of recurrences (CRs). In our approach, CR-normalized representations are obtained for a larger class of induction variables than GIVs (e.g. factorials and exponentials).

We are currently investigating the application of the CR method in the areas of program equivalence determination, program correctness proofs, and in optimizing compiler validation techniques [15] to deal with loop semantics. The induction variable recognition method described in this paper can also be used for non-linear dependence testing, see [16]. The basic idea is that dependence

```
DO i=1,m
 DO j=1,i
  ij=ij+1
  ijkl=ijkl+i-j+1
  DO k=i+1,m
   DO l=1,k
    ijkl=ijkl+1
    xijkl[ijkl]=xkl[l]
   ENDDO
  ENDDO
  ijkl=ijkl+ij+left
 ENDDO
ENDDO
```

(1)

$\Rightarrow$

```
DO i=1,m
 DO j=1,i
  ij=ij+1
  ijkl=ijkl+i-j+1
  DO k=i+1,m
   DO l=0,k-1
    xijkl[{ijkl+1,+,1}_l]
       =xkl[{1,+,1}_l]
   ENDDO
   ijkl=ijkl+k
  ENDDO
  ijkl=ijkl+ij+left
 ENDDO
ENDDO
```

(2)

$\Rightarrow$

```
DO i=1,m
 DO j=1,i
  DO k=0,m-i-1
   DO l=0,{i,+,1}_k
    xijkl[{{ijkl+i-j+2,+,i+1
       ,+,1}_k,+,1}_l]=xkl[{1,+,1}_l]
   ENDDO
  ENDDO
  ijkl=ijkl+i-j+ij+left+2
    +(i*m*(i+1)+(m+1)*(m-i*(i+1)))/2
  ij=ij+1
 ENDDO
ENDDO
```

(3)

$\Rightarrow$

```
DO i=1,m
 DO j=0,i-1
  DO k=0,m-i-1
   DO l=0,{i,+,1}_k
    xijkl[{{{i+ijkl+1,+,i+ij
          +left+(i*m*(i+1)
          +(m+1)*(m-i*(i+1)))/2}_j,
          +,i+1,+,1}_k,+,1}_l]
       =xkl[{1,+,1}_l]
   ENDDO
  ENDDO
 ENDDO
 ijkl=ijkl+i*(ij+left+i*(m/2+1)
     +(m*i**2-(m+1)*(i+i**2-m))/2+1)
 ij=ij+i
ENDDO
```

(4)

$\Rightarrow$

```
DO i=0,m-1
 DO j=0,i
  DO k=0,m-i-2
   DO l=0,i+k+1
    xijkl[ijkl+l+i*(k+(m+m**2
          +2*left+6)/4)+j*(left
          +(m+m**2)/2)+((i*m)**2
          +2*(k**2+3*k+i**2*(left
          +1))+m*i**2)/4+2]=xkl[l+1]
   ENDDO
  ENDDO
 ENDDO
ENDDO
```

(5)

Fig. 12. Algorithm *IVS* Applied to Code Segment of TRFD

distance vectors are obtained by subtracting two CR-normalized array index expressions, normalizing the result with the $\mathcal{CR}$ rules, and testing for the resulting sign of the CR. No other existing non-linear dependence testing method is as fast as this approach. Other existing compiler techniques for dependence testing are based on value range analysis, first introduced in [6]. Fahringer [9] improved these methods for non-linear dependence testing. The method is particularly suitable for analysis of dependencies across multiple loop levels. These methods are complementary and can be combined with our approach.

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.L. Harrison 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, P.S. Wang, and E.V. 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.

[6] 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, August 1995.

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

[8] R. Eigenmann, J. Hoeflinger, Z. Li, and D.A. 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.

[9] T. Fahringer. Efficient symbolic analysis for parallelizing compilers and performance estimators. *Supercomputing*, 12(3):227–252, May 1998.

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

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

[12] P. Knupp and S. Steinberg. *Fundamentals of Grid Generation.* CRC Press, 1994.

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

[14] J.P. Singh and J.L. Hennessy. An emperical investigation of the effectiviness and limitations of automatic parallelization. In N. Suzuki, editor, *Shared Memory Multiprocessing*, pages 203–207. MIT press, Cambridge MA, 1992.

[15] R. van Engelen, D. Whalley, and X. Yuan. Automatic validation of code-improving transformations. In *ACM SIGPLAN Workshop on Language, Compilers, and Tools for Embedded Systems*, 2000.

[16] R.A. van Engelen. Symbolic evaluation of chains of recurrences for loop optimization. Technical report, TR-000102, Computer Science Deptartment, Florida State University, 2000. Available from http://www.cs.fsu.edu/˜engelen/cr.ps.gz.

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

[18] R.A. 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.

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

[20] M.J. Wolfe. *High Performance Compilers for Parallel Computers.* Addison-Wesley, Redwood City, CA, 1996.

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

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