Technical Report TR-041223, Department of Computer Science, Florida State University, December 2004.

The CR# Algebra and its Application in Loop Analysis and Optimization

Robert A. van Engelen*
Department of Computer Science and School of Computational Science
Florida State University
FL32306, USA

Abstract

This report presents a novel family of linear-time algorithms for loop analysis based on the CR# (CR-sharp) algebra, which is a new nontrivial extension of the Chains of Recurrences (CR) algebra. Conventional compiler methods apply induction variable substitution and array recovery translations to construct closed forms for induction variables and pointers prior to dependence testing and loop optimization. In this report we take a radically different approach to symbolic analysis by turning the problem up-side-down. We convert closed forms to recurrences and compute recurrence relations for (non)linear induction variables and conditionally updated variables and pointers. The recurrence forms are used to solve a larger class of loop analysis problems such as nonlinear array dependence testing without requiring a-priori code translations.

1. Introduction

This report presents a novel family of compiler algorithms to analyze the recurrences of nonlinear induction variables and pointers that are conditionally updated in the control-flow graph (CFG) of a loop nest. The new CR# (CR-sharp) algebra introduced in this report makes these algorithms substantially more powerful, yet simpler to implement and more efficient compared to related work. To our knowledge, our algorithms are the first to analyze and accurately bound the sequences of conditionally updated variables and pointers in a multidimensional loop nests with a time complexity that is linear in the number of blocks comprising the CFG of the loop nest.

The algorithms presented in this report are part of a new class of algorithms for induction variable and pointer analysis. In our previous work [33, 34] we presented an algorithm for induction variable analysis based on the CR algebra [3, 42]. We showed that the algorithm is more powerful and efficient compared to symbolic differencing [16] and other approaches [2, 14, 28, 39]. In [35] we extended the algorithm to convert common pointer arithmetic to array accesses (also known as array recovery [13]) to support array-based data dependence testing on pointer-based codes. In [38] we further improved the algorithms to analyze conditional updates and more complex forms of pointer arithmetic and we implemented new methods for nonlinear data dependence testing [36, 37] and value range analysis [8]. However, the disadvantage of the latter type of methods is the potential exponential time complexity required to analyze all possible flow paths in the body of a loop.

In this report we present extensions and efficiency improvements of the aforementioned methods. More specifically,

- We present a family of new loop analysis algorithms that operate on the reducible CFG of a (multi-dimensional) loop nest without any restrictions on the loop structure;
- The algorithms run in linear time in the number of blocks of the CFG regardless of the complexity of the CFG, i.e. independent of the number of possible paths in the loop nest;
- Full analysis of the recurrences of variables and pointers that are conditionally updated, which means that loops that could not be analyzed by conventional methods due to control flow can now be analyzed and optimized;
- We introduce a nontrivial extension CR# of the CR algebra with “delayed” recurrences to simplify the manipulation of recurrences of loops in the presence of direct and indirect wrap-around variables;
- We present an efficient method to compute a symbolic bound on the number of loop iterations of a pre- or post-test loop, where the loop exit condition may contain conditionally updated variables. Our work is the first to address this class of loop iteration problems;
- We provide proof of soundness and termination of the algorithms.

* Supported in part by NSF grants CCR-0105422, CCR-0208892 and DOE grant DEFG02-02ER25543.

2. Motivation

Accurate dependence testing is critical for the effectiveness of compilers to optimize loops for vectorization and parallelization, or to improve performance. Most loop optimizations rely on exact or inexact array data dependence testing [6, 9, 10, 11, 12, 14, 15, 16, 17, 18, 19, 21, 22, 23, 25, 27, 29, 30, 31, 32, 40, 44]. Current dependence analyzers are quite powerful and are able to solve complicated dependence problems, e.g. using the polyhedral model [7, 20]. However, more recently several authors [13, 24, 26, 35, 38, 41] point at the difficulty these dependence analyzers still have with nonlinear symbolic expressions, pointer arithmetic, and control flow in loops.

Part of the problem is the application of induction variable substitution (IVS) to construct closed forms for induction variables prior to array dependence testing and loop optimization [2, 14, 16, 34]. Because conditionally updated variables do not have closed forms many compiler optimizations cannot be applied [38, 41]. Another problem is the complexity of current IVS methods [2, 14, 16, 28, 39] that require extensive symbolic manipulation. In addition, these algorithms cannot be easily adapted to low-level CFG-based compiler optimizations, because of the semantic gap between information processed by high-level restructuring compilers and low-level code optimizing compilers.

To address these concerns we take a radically different approach by turning the problem up-side-down. We convert closed forms to recurrences and compute recurrence relations for (non)linear induction variables and conditionally updated variables and pointers. The recurrence forms are used to solve a larger class of loop analysis problems, such as array-based dependence testing, without requiring any a-priori code translations.

The remainder of this report is organized as follows. In Section 3 we present the CR# algebra with a new formalism for representing and manipulating "delayed" recurrences. Section 4 presents the loop analysis algorithms with a discussion of their efficiency and proof of soundness and termination, followed by the results of the improved algorithm on dependence testing in Section 5. We summarize the results and conclusions of the report in Section 6.

3. The CR# Algebra

In this section we present our nontrivial extension CR# (CR-sharp) of the CR algebra for the analysis and manipulation of irregular functions.

3.1. Chains of Recurrences

A function or closed-form expression evaluated over a unit-distant grid with index i can be rewritten into a mathematically equivalent CR expression [3, 34]. A CR form $\Phi_i = \{\phi_0, \odot_1, f_1\}_i$ describes a sequence of values starting with an initial value ϕ_0 updated in each iteration by adding ($\odot_1 = +$) or multiplying ($\odot_1 = *$) the current value by the "increment" or "step" function f_1. When f_1 is another CR form this produces a *chain of recurrences*

$$\Phi_i = \{\phi_0, \odot_1, \{\phi_1, \odot_2, \{\phi_2, \cdots, \odot_k, \{\phi_k\}_i\}_i\}_i\}_i$$

which is usually written in flattened form

$$\Phi_i = \{\phi_0, \odot_1, \phi_1, \odot_2, \cdots, \odot_k, \phi_k\}_i$$

The CR coefficients ϕ are (symbolic) constants or nested CR forms of a different grid variable representing multivariate CR (MCR) forms [4].

A CR form is essentially a short-hand notation for a set of recurrence relations evaluated over an iteration space $i = 0, \ldots, n-1$. The following loop template is perhaps the simplest way to express the meaning of a CR form:

$\mathsf{cr}_0 = \phi_0$
$\mathsf{cr}_1 = \phi_1$
$\vdots = \vdots$
$\mathsf{cr}_{k-1} = \phi_{k-1}$
for $i = 0$ to $n-1$
  $\mathsf{val}[i] = \mathsf{cr}_0$
  $\mathsf{cr}_0 = \mathsf{cr}_0 \odot_1 \mathsf{cr}_1$
  $\mathsf{cr}_1 = \mathsf{cr}_1 \odot_2 \mathsf{cr}_2$
  $\vdots = \vdots \quad \vdots \quad \vdots$
  $\mathsf{cr}_{k-1} = \mathsf{cr}_{k-1} \odot_k \phi_k$
endfor

The loop produces the sequence $\mathsf{val}[i]$ of the CR form[1].

The CR form provides a powerful notation to describe functions evaluated in an iteration space. Consider the type of functions commonly encountered in the symbolic analysis performed by a compiler.

Affine functions are uniquely represented by nested CR forms $\{a, +, s\}_i$, where a is the integer-valued initial value and s is the integer-valued stride in the direction of i. The coefficient a is a nested CR form in another dimension.

Multivariate Polynomials are uniquely represented by nested CR forms of length k, where k is the maximum order of the polynomial. All $\odot$ operations in the CR form are additions, i.e. $\odot = +$.

Geometric functions $a\,r^i$ are uniquely represented by the CR form $\{a, *, r\}_i$.

Characteristic functions of generalized induction variables (GIVs) are uniquely represented by CR forms [33].

The CR forms of these and other functions can be easily derived using the CR algebra simplification rules. The rules

1 This sequence is one-dimensional. A multidimensional loop nest is constructed for multivariate CR forms.

do not require extensive symbolic manipulation [34], because the manipulation is similar to constant folding [1]. To compute the CR form of an expression we replace the iteration counter(s) with their CR forms and then apply the CR rules to produce a (multivariate) CR form, e.g. as in

$$f(i) = \tfrac{1}{2}(i^2 - i) \Rightarrow \tfrac{1}{2}(\{0,+,1\}_i^2 - \{0,+,1\}_i) \Rightarrow \tfrac{1}{2}(\{0,+,1,+,2\}_i - \{0,+,1\}_i) \Rightarrow \tfrac{1}{2}\{0,+,0,+,2\}_i \Rightarrow \{0,+,0,+,1\}_i$$

The CR algebra is closed under the formation of a (multivariate) characteristic function of a GIV for induction variable analysis [34]. However, the original CR algebra rules are insufficient to compose irregular functions with exceptional values. Functions that start with a sequence of unrelated initial values or functions with irregular increments cannot be represented. For more details on the original CR algebra, we refer to [3, 5, 34, 35, 42, 43].

3.2. The Delay Operator of the CR# Algebra

We introduce a new operator # together with new algebraic simplification rules on CR forms containing the operator. The new CR# algebra rules are shown in Figure 1.

Definition 1 *The* delay operator *# is defined by*

$$(x \# y) = y$$

for any x and y.

A CR form containing the delay operator will be referred to as a *delayed CR form*. The reason for the terminology is explained as follows.

Consider $\Phi_i = \{\phi_0, \odot_1, \cdots, \#_j, \cdots, \odot_k, \phi_k\}_i$. Note that the loop template of Φ_i updates variable cr_{j-1} by

$$\mathsf{cr}_{j-1} = \mathsf{cr}_{j-1} \# \mathsf{cr}_j$$

which is identical to the assignment

$$\mathsf{cr}_{j-1} = \mathsf{cr}_j$$

Therefore, the # operator introduces a one-iteration delay in the sequence of values produced by subsequent updates in the loop.

The delay operator allows a set of initial values to take effect before the regular sequence kicks in. Thus, delayed CR forms define recurrences with out-of-sequence values. This serves two important purposes. First, delayed CR forms can be used to define any sequence of values $x_0, x_1, \ldots, x_k$, possibly followed by a polynomial, geometric, or another delayed form Ψ_i as in $\Phi_i = \{x_0, \#, x_1, \#, \cdots, \#, x_k, \#, \Psi_i\}$. Secondly, induction variables that are dependent on wrap-around variables, i.e. indirect wrap-around variables, can be accurately represented using delayed forms. Wrap-around variables are currently handled with ad-hoc techniques by existing IVS methods and indirect wrap-around variables cannot be handled.

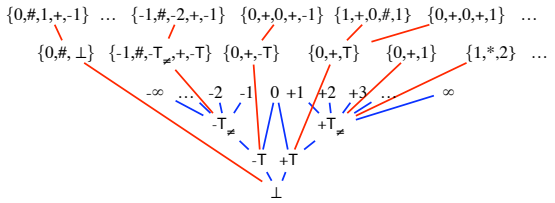


Figure 2. Snapshot of the CR# Lattice

3.3. The CR# Lattice

To define a lattice on the CR forms in the CR# algebra we introduce three special values $\bot$, $\top$, and $\top_{\neq}$.

Definition 2 *The $\bot$, $\top$, and $\top_{\neq}$ elements are defined by*

- *$\bot$ denotes an unknown quantity $-\infty \leq \bot \leq \infty$;*
- *$\top$ denotes a nonnegative unknown $0 \leq \top \leq \infty$;*
- *$\top_{\neq}$ denotes a positive unknown $0 < \top_{\neq} \leq \infty$.*

Definition 3 *The* reduction *operator $\mathcal{R}$ is defined by*

$$\mathcal{R}\{\phi_0, \odot_1, f_1\}_i = \{\mathcal{R}\phi_0, \odot_1, \mathcal{R}f_1\}_i$$

with $\mathcal{R}x$ of a non-CR (symbolic) coefficient x defined by

$$\mathcal{R}x = \begin{cases} -\top_{\neq} & \text{if } x < 0 \\ -\top & \text{if } x \leq 0 \\ 0 & \text{if } x = 0 \\ \top & \text{if } x \geq 0 \\ \top_{\neq} & \text{if } x > 0 \\ \bot & \text{if the sign of } x \text{ is unknown} \end{cases}$$

The following CR# rewrite rules are applied to reduce the CR form to a single $\bot$, $\pm\top$, $\pm\top_{\neq}$, or 0 value:

$$\begin{array}{rcl} \{\bot, \odot_1, f_1\}_i & \Rightarrow & \bot \\ \{\phi_0, \odot_1, \bot\}_i & \Rightarrow & \bot \\ \{\top, +, \top\}_i & \Rightarrow & \top \\ \{-\top, +, -\top\}_i & \Rightarrow & -\top \\ \{\top, +, -\top\}_i & \Rightarrow & \bot \\ \{-\top, +, \top\}_i & \Rightarrow & \bot \end{array} \qquad \begin{array}{rcl} \{\top, *, \top\}_i & \Rightarrow & \top \\ \{-\top, *, \top\}_i & \Rightarrow & -\top \\ \{\phi_0, *, -\top\}_i & \Rightarrow & \bot \\ \{\top, \#, \top\}_i & \Rightarrow & \top \\ \{-\top, \#, -\top\}_i & \Rightarrow & -\top \\ \{-\top, \#, \top\}_i & \Rightarrow & \bot \\ \{\top, \#, -\top\}_i & \Rightarrow & \bot \end{array}$$

The rules for 0 and $\top_{\neq}$ are similar.

Note that for determining the value of $\mathcal{R}x$ when x is symbolic we can use common rules such as

$$\top + E \Rightarrow \begin{cases} \top & \text{if } E \geq 0 \\ \bot & \text{otherwise} \end{cases}$$

$$E * \top \Rightarrow \begin{cases} \top & \text{if } E > 0 \\ -\top & \text{if } E < 0 \\ \bot & \text{otherwise} \end{cases}$$

The reduction operation traverses the lattice starting with a CR-form and stops at $\bot$, $\pm\top$, $\pm\top_{\neq}$, or 0. For example

$$\mathcal{R}\{0,+,0,+,1\}_i = \{\mathcal{R}0, +, \mathcal{R}\{0,+,1\}_i\}_i \Rightarrow \{0,+,\top\}_i \Rightarrow \top$$

Thus, the sequence of $\{0,+,0,+,1\}_i$ is nonnegative.

The CR# lattice shown in Figure 2 enables a graceful degradation of information on recurrences to determine their properties such as sign and monotonicity.

$CR\#$				
#	*LHS*		*RHS*	*Condition*
1	$\{\phi_0, +, 0\}_i$	$\Rightarrow$	ϕ_0	
2	$\{\phi_0, *, 1\}_i$	$\Rightarrow$	ϕ_0	
3	$\{0, *, f_1\}_i$	$\Rightarrow$	0	
4	$-\{\phi_0, +, f_1\}_i$	$\Rightarrow$	$\{-\phi_0, +, -f_1\}_i$	
5	$-\{\phi_0, *, f_1\}_i$	$\Rightarrow$	$\{-\phi_0, *, f_1\}_i$	
6	$\{\phi_0, +, f_1\}_i \pm E$	$\Rightarrow$	$\{\phi_0 \pm E, +, f_1\}_i$	when E is i-loop invariant
7	$\{\phi_0, *, f_1\}_i \pm E$	$\Rightarrow$	$\{\phi_0 \pm E, +, \phi_0 * (f_1 - 1), *, f_1\}_i$	when E and f_1 are i-loop invariant
8	$E * \{\phi_0, +, f_1\}_i$	$\Rightarrow$	$\{E * \phi_0, +, E * f_1\}_i$	when E is i-loop invariant
9	$E * \{\phi_0, *, f_1\}_i$	$\Rightarrow$	$\{E * \phi_0, *, f_1\}_i$	when E is i-loop invariant
10	$E/\{\phi_0, +, f_1\}_1$	$\Rightarrow$	$1/\{\phi_0/E, +, f_1/E\}_i$	when $E \neq 1$ is i-loop invariant
11	$E/\{\phi_0, *, f_1\}_1$	$\Rightarrow$	$\{E/\phi_0, *, 1/f_1\}_i$	when E is i-loop invariant
12	$\{\phi_0, +, f_1\}_i \pm \{\psi_0, +, g_1\}_i$	$\Rightarrow$	$\{\phi_0 \pm \psi_0, +, f_1 \pm g_1\}_i$	
13	$\{\phi_0, *, f_1\}_i \pm \{\psi_0, +, g_1\}_i$	$\Rightarrow$	$\{\phi_0 \pm \psi_0, +, \{\phi_0 * (f_1 - 1), *, f_1\}_i \pm g_1\}_i$	when f_1 is i-loop invariant
14	$\{\phi_0, +, f_1\}_i * \{\psi_0, +, g_1\}_i$	$\Rightarrow$	$\{\phi_0 * \psi_0, +, \{\phi_0, +, f_1\}_i * g_1 + \{\psi_0, +, g_1\}_i * f_1 + f_1 * g_1\}_i$	
15	$\{\phi_0, *, f_1\}_i * \{\psi_0, *, g_1\}_i$	$\Rightarrow$	$\{\phi_0 * \psi_0, *, f_1 * g_1\}_i$	
16	$\{\phi_0, *, f_1\}_i^E$	$\Rightarrow$	$\{\phi_0^E, *, f_1^E\}_i$	when E is i-loop invariant
17	$\{\phi_0, *, f_1\}_i^{\{\psi_0, +, g_1\}_i}$	$\Rightarrow$	$\{{\phi_0}^{\psi_0}, *, \{\phi_0, *, f_1\}_i^{g_1} * f_1^{\{\psi_0, +, g_1\}_i} * f_1^{g_1}\}_i$	
18	$E^{\{\phi_0, +, f_1\}_i}$	$\Rightarrow$	$\{E^{\phi_0}, *, E^{f_1}\}_i$	when E is i-loop invariant
19	$\{\phi_0, +, f_1\}_i^n$	$\Rightarrow$	$\begin{cases} \{\phi_0, +, f_1\}_i * \{\phi_0, +, f_1\}_i^{n-1} & \textbf{if } n \in \mathbb{Z},\ n > 1 \\ 1/\{\phi_0, +, f_1\}_i^{-n} & \textbf{if } n \in \mathbb{Z},\ n < 0 \end{cases}$	
20	$\{\phi_0, +, f_1\}_i!$	$\Rightarrow$	$\begin{cases} \{\phi_0!, *, \left(\prod_{j=1}^{f_1} \{\phi_0 + j, +, f_1\}_i\right)\}_i & \textbf{if } f_1 \geq 0 \\ \{\phi_0!, *, \left(\prod_{j=1}^{\lvert f_1 \rvert} \{\phi_0 + j, +, f_1\}_i\right)^{-1}\}_i & \textbf{if } f_1 < 0 \end{cases}$	
21	$\{\phi_0, +, \phi_1, *, f_2\}$	$\Rightarrow$	$\{\phi_0, *, f_2\}_i$	when $\frac{\phi_1}{\phi_0} = f_2 - 1$
22	$\{\phi_0, \#, f_1\}_i$	$\Rightarrow$	f_1	when $\phi_0 = \mathcal{V}\mathcal{B}f_1$ (see Appendix A.1 for $\mathcal{V}$ and $\mathcal{B}$)
23	$-\{\phi_0, \#, f_1\}_i$	$\Rightarrow$	$\{-\phi_0, \#, -f_1\}_i$	
24	$\{\phi_0, \#, f_1\}_i \pm E$	$\Rightarrow$	$\{\phi_0 \pm E, \#, f_1 \pm E\}_i$	when E is i-loop invariant
25	$E * \{\phi_0, \#, f_1\}_i$	$\Rightarrow$	$\{E * \phi_0, \#, E * f_1\}_i$	when E is i-loop invariant
26	$\{\phi_0, \#, f_1\}_i \pm \{\psi_0, \#, g_1\}_i$	$\Rightarrow$	$\{\phi_0 \pm \psi_0, \#, f_1 \pm g_1\}_i$	
27	$\{\phi_0, \#, f_1\}_i * \{\psi_0, \#, g_1\}_i$	$\Rightarrow$	$\{\phi_0 * \psi_0, \#, f_1 * g_1\}_i$	
28	$\{\phi_0, \#, f_1\}_i \pm \{\psi_0, +, g_1\}_i$	$\Rightarrow$	$\{\phi_0 \pm \psi_0, \#, f_1 \pm \mathcal{F}\{\psi_0, +, g_1\}_i\}_i$	(see Appendix A.1 for $\mathcal{F}$)
29	$\{\phi_0, \#, f_1\}_i * \{\psi_0, +, g_1\}_i$	$\Rightarrow$	$\{\phi_0 * \psi_0, \#, f_1 * \mathcal{F}\{\psi_0, +, g_1\}_i\}_i$	(see Appendix A.1 for $\mathcal{F}$)
30	$\{\phi_0, +, \phi_1, \#, f_2\}$	$\Rightarrow$	$\{\phi_0, \#, \phi_0 + \phi_1, +f_2\}_i$	
31	$\{\phi_0, *, \phi_1, \#, f_2\}$	$\Rightarrow$	$\{\phi_0, \#, \phi_0 * \phi_1, *f_2\}_i$	

$CR\#^{-1}$				
#	*LHS*		*RHS*	*Condition*
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	$\{0, +, f_1^i\}_i$	$\Rightarrow$	$\frac{f_1^i - 1}{f_1 - 1}$	when i does not occur in f_1 and $f_1 \neq 1$
7	$\{0, +, f_1^{g_1 + h_1}\}_i$	$\Rightarrow$	$\{0, +, f_1^{g_1} * f_1^{h_1}\}_i$	
8	$\{0, +, f_1^{g_1 * h_1}\}_i$	$\Rightarrow$	$\{0, +, (f_1^{g_1})^{h_i}\}_i$	when i does not occur in f_1 and g_1
9	$\{0, +, f_1\}_i$	$\Rightarrow$	$i * f_1$	when i does not occur in f_1
10	$\{0, +, i\}_i$	$\Rightarrow$	$\frac{i^2 - i}{2}$	
11	$\{0, +, i^n\}_i$	$\Rightarrow$	$\sum_{k=0}^{n} \frac{\binom{n+1}{k}}{n+1} B_k\, i^{n-k+1}$	for $n \in \mathbb{N}$, B_k is k^{th} Bernoulli number
12	$\{1, *, -f_1\}_i$	$\Rightarrow$	$(-1)^i \{1, *, f_1\}_i$	
13	$\{1, *, \frac{1}{f_1}\}_i$	$\Rightarrow$	$\{1, *, f_1\}_i^{-1}$	
14	$\{1, *, f_1 * g_1\}_i$	$\Rightarrow$	$\{1, *, f_1\}_i * \{1, *, g_1\}_i$	
15	$\{1, *, f_1^{g_1}\}_i$	$\Rightarrow$	$f_1^{\{1, *, g_1\}_i}$	when i does not occur in f_1
16	$\{1, *, g_1^{f_1}\}_i$	$\Rightarrow$	$\{1, *, g_1\}_i{}^{f_1}$	when i does not occur in f_1
17	$\{1, *, f_1\}_i$	$\Rightarrow$	f_1^i	when i does not occur in f_1
18	$\{1, *, i\}_i$	$\Rightarrow$	0^i	
19	$\{1, *, i + f_1\}_i$	$\Rightarrow$	$\frac{(i + f_1 - 1)!}{(f_1 - 1)!}$	when i does not occur in f_1 and $f_1 \geq 1$
20	$\{1, *, f_1 - i\}_i$	$\Rightarrow$	$(-1)^i * \frac{(i - f_1 - 1)!}{(-f_1 - 1)!}$	when i does not occur in f_1 and $f_1 \leq -1$
21	$\{\phi_0, \#, f_1\}_i$	$\Rightarrow$	$(i = 0)\,?\,\phi_0 : f_1[i \leftarrow i - 1]$	(assuming f_1 is in closed form, replace i with $i - 1$ in f_1)

Figure 1. The Complete $CR\#$ and $CR\#^{-1}$ Algebra Rewrite Rules

3.4. Monotonicity

To determining the monotonic properties of a CR form we extract directional information by applying the reduction operator on the increment function of a CR form.

Definition 4 *The* monotonic *operator $\mathcal{M}$ is defined by*

$$\begin{array}{rcl} \mathcal{M}\{\phi_0,+,f_1\}_i & = & \mathcal{R}f_1 \\ \mathcal{M}\{\phi_0,*,f_1\}_i & = & \begin{cases} \mathcal{R}\phi_0 & \text{if } \mathcal{R}(f_1-1)=\top_{\neq} \\ \bot & \text{otherwise} \end{cases} \\ \mathcal{M}\{\phi_0,\#,f_1\}_i & = & \mathcal{M}f_1 \bowtie \mathcal{R}(\mathcal{V}f_1-\phi_0) \end{array}$$

with $\mathcal{M}x=0$ when x is a (symbolic) constant. The lattice relation $\bowtie$ returns the maximum element $z=x\bowtie y$ in the lattice such that $z\preceq x$ and $z\preceq y$. See Appendix A.1 for the definition of $\mathcal{V}$.

The monotonic operator returns directional information on a CR form as a lattice element $\top$ (monotonically increasing), $\top_{\neq}$ (strictly monotonically increasing), $-\top$ (monotonically decreasing), $-\top_{\neq}$ (strictly monotonically decreasing), 0 (constant), or $\bot$ (unknown). Consider for example

$$\mathcal{M}\{0,\#,0,+,1\}_i = \mathcal{M}\{0,+,1\}_i \bowtie \mathcal{R}(\mathcal{V}\{0,+,1\}_i - 0) = \mathcal{R}1 \bowtie \mathcal{R}0 = \top_{\neq} \bowtie 0 = \top$$

Thus, the sequence generated by $\{0,\#,0,+,1\}_i$ is monotonically increasing. Determining the monotonicity of a CR form enables accurate value range analysis [8] and nonlinear dependence testing [36] further discussed in Section 5.

3.5. CR# Alignment

Two or more CR forms of different lengths or with different operations can be aligned for comparison. By comparing the coefficients ϕ_j of a CR form we can determine whether one of the CR forms bounds the other(s).

A delay operator can be inserted in a CR form according to the following lemma.

Lemma 1 *Let $\Phi_i=\{\phi_0,\odot_1,f_1\}_i$ be a (multivariate) CR form. Then,*

$$\Phi_i = \{\phi_0,\#,\mathcal{F}\Phi_i\}_i$$

See Appendix A.1 for the definition of $\mathcal{F}$.

As a consequence of the chain property of a nested CR form a # operator can be inserted anywhere in a CR form.

To align a (delayed) CR form of a mixed polynomial and geometric function to a longer (delayed) CR form, $+$ operators can be inserted for pairwise alignment of the $*$ operations between two or more CR forms.

Lemma 2 *Let $\Phi_i = \{\phi_0,\odot_1,\cdots,\odot_{k-1},\phi_{k-1},*,\phi_k\}_i$ such that ϕ_k is invariant of i. Then, any number $m>0$ of $+$ operators can be inserted at the $(k-1)^{\text{th}}$ coefficient*

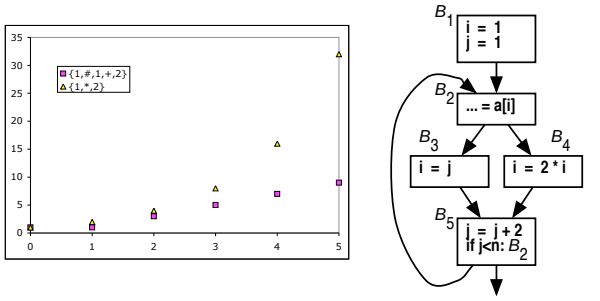


Figure 3. Plot of $\{1,\#,1,+,2\}_i$ and $\{1,*,2\}_i$

$$\Phi_i = \{\phi_0,\odot_1,\cdots,\odot_{k-1},\phi_{k-1},\underbrace{+,\phi_{k-1}(\phi_k-1),+,\phi_{k-1}(\phi_k-1)^2,+,\cdots,+,\phi_{k-1}(\phi_k-1)^m}_{\textit{inserted}},*,\phi_k\}_i$$

without changing the sequence of Φ_i.

To align two CR forms of unequal length, the shorter CR can be lengthened by adding dummy operations as follows.

Lemma 3 *Let $\Phi_i=\{\phi_0,\odot_1,\phi_1,\odot_2,\cdots,\odot_k,\phi_k\}_i$ be a (multivariate) CR form, where ϕ_k is invariant of i. Then, the following identities hold*

$$\begin{array}{rcl} \Phi_i & = & \{\phi_0,\odot_1,\phi_1,\odot_2,\cdots,\odot_k,\phi_k,+,0\}_i \\ \Phi_i & = & \{\phi_0,\odot_1,\phi_1,\odot_2,\cdots,\odot_k,\phi_k,*,1\}_i \\ \Phi_i & = & \{\phi_0,\odot_1,\phi_1,\odot_2,\cdots,\odot_k,\phi_k,\#,\phi_k\}_i \end{array}$$

Two or more CR forms can aligned using Lemmas 1, 2, and 3. Consider for example the alignment

$$\begin{array}{r} \Phi_i = \{1,\#,1,+,2\}_i = \{1,\#,1,+,2,*,1\}_i \\ \Psi_i = \{1,*,2\}_i = \{1,\#,2,*,2\}_i = \{1,\#,2,+,2,*,2\}_i \end{array}$$

3.6. CR# Bounds

Alignment allows us to compare the coefficients of CR forms to determine bounds. It is evident that the sequence of $\Psi_i=\{1,*,2\}_i$ dominates $\Phi_i=\{1,\#,1,+,2\}_i$, as is shown in Figure 3. A mix of the sequences is generated for variable i by the CFG of the loop shown in Figure 3. The Φ_i and Ψ_i sequences bound the values of i. Our induction variable analysis algorithm discussed in the next section computes the bounding CR forms for recurrences in loops using alignment and the min/max of two CR forms.

Definition 5 *The* minimum *of two CR forms is inductively defined by*

$$\min(\{\phi_0,\#,f_1\}_i,\{\psi_0,\#,g_1\}_i)=\{\min(\phi_0,\psi_0),\#,\min(f_1,g_1)\}_i$$
$$\min(\{\phi_0,+,f_1\}_i,\{\psi_0,+,g_1\}_i)=\{\min(\phi_0,\psi_0),+,\min(f_1,g_1)\}_i$$
$$\min(\{\phi_0,*,f_1\}_i,\{\psi_0,*,g_1\}_i)$$
$$= \begin{cases} \{\min(\phi_0,\psi_0),*,\min(f_1,g_1)\}_i & \text{if } \phi_0{>}0 \wedge \psi_0{>}0 \wedge f_1{>}0 \wedge g_1{>}0 \\ \{min(\phi_0,\psi_0),*,\max(f_1,g_1)\}_i & \text{if } \phi_0{<}0 \wedge \psi_0{<}0 \wedge f_1{>}0 \wedge g_1{>}0 \\ \{\phi_0,*,f_1\}_i & \text{if } \phi_0{<}0 \wedge \psi_0{>}0 \wedge f_1{>}0 \wedge g_1{>}0 \\ \{\psi_0,*,g_1\}_i & \text{if } \phi_0{>}0 \wedge \psi_0{<}0 \wedge f_1{>}0 \wedge g_1{>}0 \\ \{-\max(|\phi_0|,|\psi_0|),*,\max(|f_1|,|g_1|)\}_i & \text{if } f_1{<}0 \vee g_1{<}0 \\ \bot & \text{otherwise} \end{cases}$$

where the sign of the coefficients is determined using the $\mathcal{R}$ operator. The maximum *of two CR forms is inductively defined by*

$$\max(\{\phi_0,\#,f_1\}_i,\{\psi_0,\#,g_1\}_i)=\{\max(\phi_0,\psi_0),\#,\max(f_1,g_1)\}_i$$
$$\max(\{\phi_0,+,f_1\}_i,\{\psi_0,+,g_1\}_i)=\{\max(\phi_0,\psi_0),+,\max(f_1,g_1)\}_i$$
$$\max(\{\phi_0,*,f_1\}_i,\{\psi_0,*,g_1\}_i)$$
$$= \begin{cases} \{\max(\phi_0,\psi_0),*,\max(f_1,g_1)\}_i & \text{if } \phi_0{>}0 \wedge \psi_0{>}0 \wedge f_1{>}0 \wedge g_1{>}0 \\ \{\max(\phi_0,\psi_0),*,\min(f_1,g_1)\}_i & \text{if } \phi_0{<}0 \wedge \psi_0{<}0 \wedge f_1{>}0 \wedge g_1{>}0 \\ \{\phi_0,*,f_1\}_i & \text{if } \phi_0{>}0 \wedge \psi_0{<}0 \wedge f_1{>}0 \wedge g_1{>}0 \\ \{\psi_0,*,g_1\}_i & \text{if } \phi_0{<}0 \wedge \psi_0{>}0 \wedge f_1{>}0 \wedge g_1{>}0 \\ \{\max(|\phi_0|,|\psi_0|),*,\max(|f_1|,|g_1|)\}_i & \text{if } f_1{<}0 \vee g_1{<}0 \\ \bot & \text{otherwise} \end{cases}$$

Consider for example

$$\min(\{1,\#,1,+,2,*,1\}_i,\{1,\#,2,+,2,*,2\}_i)=\{1,\#,1,+,2,*,1\}_i$$
$$\max(\{1,\#,1,+,2,*,1\}_i,\{1,\#,2,+,2,*,2\}_i)=\{1,\#,2,+,2,*,2\}_i$$

After application of min and max the CR forms can be simplified using the CR# algebra rules. Figure 3 shows a plot of the simplified minimum and maximum CR forms.

4. Loop Analysis Algorithms

Loops are analyzed in a reducible CFG from the innermost nested loops to the outermost loops. For each loop, analysis proceeds in two phases. The first phase analyzes the updates of the live variables in the loop to collect the set of recurrences relations on the variables. In the second phase the recurrence relations are solved in CR form and stored in $recset[H]$ for loop header block H. CR-form bounds on the recurrences are stored in $rangeset[H]$. Analysis of outer loops require the *recset*'s and *rangeset*'s of the inner nested loops.

The *recset* and *rangeset* of a loop gives complete information necessary for further compiler analysis, such determining the number of loop iterations, for applying induction variable substitution and array recovery, for idiom recognition, and for data dependence testing.

4.1. Phase 1

The first phase is performed by FINDRECURRENCES shown in Figure 4. The routine computes the set of recurrence relations R of a loop with (pre)header H.

The first step in FINDRECURRENCES is to find all blocks with back edges to the header H, see for example the CFG

Algorithm FINDRECURRENCES(H, R)
Find recurrence relations R for loop (pre)header H
- input: CFG with $live[H]$ the set of live variables at H
- output: set of recurrence relations R in tuple form
$R := \emptyset$
$A := \{\langle v, v\rangle \mid v \in live[H]\}$
$S := \{B{:}A \mid$ block B has a back edge to $H\}$
while $S \neq \emptyset$ **do**
  Remove the next pair $B{:}A$ from the working set S
  if B has a back edge to a loop (pre)header H_{nested}
      and $H \neq H_{\text{nested}}$ **then**
    MERGELOOP(H_{nested}, A, S)
  else
    MERGE(H, B, A, S)
    if $B = H$ **then** $R := R \cup A$ **endif**
  endif
enddo

Algorithm MERGE(H, B, A, S)
Update the recurrence system A for loop (pre)header H
by merging the effects of block B with recurrence system A
- input: loop (pre)header number H, block B,
      recurrence system A, and working set S
- output: updated A and S
for each assignment $v = expr$ in backward order
    from the last instruction in B to the first **do**
  UPDATE($A, v, expr$)
enddo
if $B \neq H$ **then**
  for each predecessor $B_{\text{pred}} \in pred[B]$
      (excluding the back edges of B) **do**
    if there is a pair $B_{\text{pred}}{:}A' \in S$ **then**
      $S := S \setminus \{B_{\text{pred}}{:}A'\} \cup \{B_{\text{pred}}{:}(A \cup A')\}$
    else
      $S := S \cup \{B_{\text{pred}}{:}A\}$
    endif
  enddo
endif

Algorithm UPDATE($A, v, expr$)
Update the recurrence system A with $v = expr$
- input: recurrence system A, variable v, expression $expr$
- output: updated recurrence system A
Handle side-effects and potential aliases (Sec. 4.9) in *expr*
for each $\langle x, y\rangle \in A$ **do**
  Update $\langle x, y\rangle$ in A by replacing all v's in y with $expr$
enddo

Figure 4. Algorithm to Construct the Recurrence System of a Loop

shown in Figure 5. The CFG has two back edges at (1) and (2). The algorithm analyzes the paths in backward order through the CFG using a working set S of blocks B associated with partially completed recurrence systems A. The initial working set contains all blocks with back edges to the header H associated with the initial set of recurrence relations $A := \{\langle v, v\rangle \mid v \in live[H]\}$ where $live[H]$ is the set of live variables (and registers) at the loop header determined with data flow analysis [1]. For example, the first two items in the working set of the example loop shown in Figure 5 are

$$S = \{\quad B_3{:}\{\langle \mathsf{i}, \mathsf{i}\rangle, \langle \mathsf{j}, \mathsf{j}\rangle, \langle \mathsf{k}, \mathsf{k}\rangle\},\quad B_4{:}\{\langle \mathsf{i}, \mathsf{i}\rangle, \langle \mathsf{j}, \mathsf{j}\rangle, \langle \mathsf{k}, \mathsf{k}\rangle\}\quad\}$$

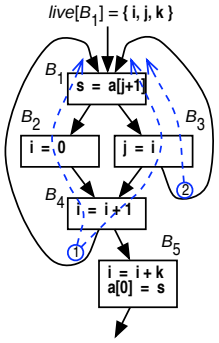


Figure 5. Initial Stage of Algorithm FIND-RECURRENCES **Applied to an Example Loop**

The working set is updated as the algorithm proceeds towards the loop header. The algorithm takes one $B{:}A$ pair from the working set, say $B_4{:}A$ with $A = \{\langle \mathsf{i}, \mathsf{i}\rangle, \langle \mathsf{j}, \mathsf{j}\rangle, \langle \mathsf{k}, \mathsf{k}\rangle\}$. The MERGE routine merges the effects of a block in the working set with its recurrence system (the merging of the effects of nested loops will be discussed later in Section 4.7). The routine updates the recurrence system for each instruction in the block in backward order starting with the last instruction. After merging, all predecessor blocks except headers are added to the working set with the updated system. For example, merging i=i+1 from block B_4 gives the updated system $A = \{\langle \mathsf{i}, \mathsf{i{+}1}\rangle, \langle \mathsf{j}, \mathsf{j}\rangle, \langle \mathsf{k}, \mathsf{k}\rangle\}$. The algorithm then proceeds with the predecessors of B_4, i.e. B_2 and B_3. Because B_3 is already in the working set, the new recurrence system is combined with the previous recurrence system for B_3 using set union:

$$\begin{aligned} S = \{ \quad & B_2{:}\{\langle \mathsf{i}, \mathsf{i{+}1}\rangle, \langle \mathsf{j}, \mathsf{j}\rangle, \langle \mathsf{k}, \mathsf{k}\rangle\}, \\ & B_3{:}\{\langle \mathsf{i}, \mathsf{i}\rangle, \langle \mathsf{j}, \mathsf{j}\rangle, \langle \mathsf{k}, \mathsf{k}\rangle\} \cup \{\langle \mathsf{i}, \mathsf{i{+}1}\rangle, \langle \mathsf{j}, \mathsf{j}\rangle, \langle \mathsf{k}, \mathsf{k}\rangle\} \quad \} \\ = \{ \quad & B_2{:}\{\langle \mathsf{i}, \mathsf{i{+}1}\rangle, \langle \mathsf{j}, \mathsf{j}\rangle, \langle \mathsf{k}, \mathsf{k}\rangle\}, B_3{:}\{\langle \mathsf{i}, \mathsf{i}\rangle, \langle \mathsf{i}, \mathsf{i{+}1}\rangle, \langle \mathsf{j}, \mathsf{j}\rangle, \langle \mathsf{k}, \mathsf{k}\rangle\} \quad \} \end{aligned}$$

This process continues until the header block is reached and merged. The resulting set of recurrence relations of the example loop is a set of variable-value pairs

$$R = \{ \quad \langle \mathsf{i}, 1\rangle, \langle \mathsf{i}, \mathsf{i}\rangle, \langle \mathsf{i}, \mathsf{i{+}1}\rangle, \langle \mathsf{j}, \mathsf{i}\rangle, \langle \mathsf{j}, \mathsf{j}\rangle, \langle \mathsf{k}, \mathsf{k}\rangle \quad \}$$

The MERGE routine plays a critical role for updating the partial recurrence systems by collecting the effects of a block. It applies a backward search through the instructions of the block to update the system. This is best illustrated with the example shown in Figure 6. In each step in MERGE the set of partial recurrence relations is updated.

4.2. Phase 2

The SOLVERECURRENCES algorithm shown in Figure 7 solves a set of recurrence relations in CR form. The algorithm requires that the initial set of recurrence relations R forms a partial order defined by the *inclusion relation* $\prec$

$$\langle v, x\rangle \prec \langle u, y\rangle \qquad \text{if } v \neq u \text{ and } v \text{ occurs in } y$$

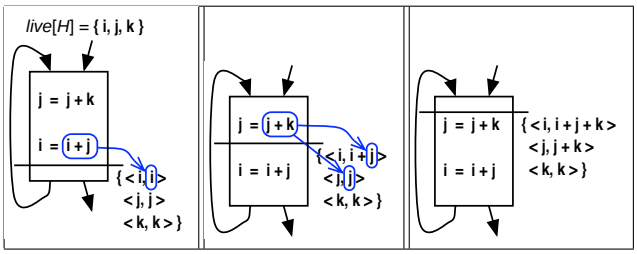


Figure 6. The MERGE **Operation**

The inclusion relation defines a digraph on the elements in R. The strongly connected components represent recurrences with periodic sequences or sequences that cannot be represented by sums and products of polynomials, exponentials, and factorials. Section 4.5 describes a technique to break the connected components.

SOLVERECURRENCES takes pairs from R in inclusion order $\prec$. The CR form solutions are computed one by one and substituted in the recurrence relations that depend on them. The CR# algebra rules are used in the main loop to simplify and normalize the CR forms after substitution to match the CR forms of the recurrence relations.

The recurrence relations R is a set of tuples computed in the first phase. For the example loop shown in Figure 5 the set R computed by FINDRECURRENCES is

$$R = \{ \quad \langle \mathsf{i}, 1\rangle, \langle \mathsf{i}, \mathsf{i}\rangle, \langle \mathsf{i}, \mathsf{i{+}1}\rangle, \langle \mathsf{j}, \mathsf{i}\rangle, \langle \mathsf{j}, \mathsf{j}\rangle, \langle \mathsf{k}, \mathsf{k}\rangle \quad \}$$

There are three inclusion relations on tuples in R

$$\langle \mathsf{i}, 1\rangle \prec \langle \mathsf{j}, \mathsf{i}\rangle, \quad \langle \mathsf{i}, \mathsf{i}\rangle \prec \langle \mathsf{j}, \mathsf{i}\rangle, \quad \langle \mathsf{i}, \mathsf{i{+}1}\rangle \prec \langle \mathsf{j}, \mathsf{i}\rangle$$

Thus, SOLVERECURRENCES selects any one of the tuples in R for variables i and k to replace them with CR forms. After computing the CR forms for these tuples and deleting the original non-CR form tuples we obtain[2]

$$\begin{aligned} R = \{ \quad & \langle \mathsf{i}, \{\mathsf{i}_0, \#, 1\}_{B_1}\rangle, \langle \mathsf{i}, \{\mathsf{i}_0\}_{B_1}\rangle, \langle \mathsf{i}, \{\mathsf{i}_0, +, 1\}_{B_1}\rangle, \langle \mathsf{j}, \mathsf{i}\rangle, \\ & \langle \mathsf{j}, \{\mathsf{i}_0, \#, 1\}_{B_1}\rangle, \langle \mathsf{j}, \{\mathsf{i}_0\}_{B_1}\rangle, \langle \mathsf{j}, \{\mathsf{i}_0, +, 1\}_{B_1}\rangle, \langle \mathsf{j}, \mathsf{j}\rangle, \langle \mathsf{k}, \{\mathsf{k}_0\}_{B_1}\rangle \quad \} \end{aligned}$$

At this point tuples for j are considered. The tuple in the deletion set $D = \{\langle \mathsf{j}, \mathsf{i}\rangle\}$ (updated by SUBSTITUTE) is discarded from R. After computing the CR forms for $\langle \mathsf{j}, \{\mathsf{i}_0, \#, 1\}_{B_1}\rangle, \langle \mathsf{j}, \{\mathsf{i}_0\}_{B_1}\rangle, \langle \mathsf{j}, \{\mathsf{i}_0, +, 1\}_{B_1}\rangle$ and $\langle \mathsf{j}, \mathsf{j}\rangle$ the resulting solution set R is stored in $recset[H]$, where

$$\begin{aligned} recset[H] = \{ & \langle \mathsf{i}, \{\mathsf{i}_0, \#, 1\}_{B_1}\rangle, \langle \mathsf{i}, \{\mathsf{i}_0\}_{B_1}\rangle, \langle \mathsf{i}, \{\mathsf{i}_0, +, 1\}_{B_1}\rangle, \\ & \langle \mathsf{j}, \{\mathsf{j}_0, \#, \mathsf{i}_0, \#, 1\}_{B_1}\rangle, \langle \mathsf{j}, \{\mathsf{j}_0, \#, \mathsf{i}_0\}_{B_1}\rangle, \\ & \langle \mathsf{j}, \{\mathsf{j}_0, \#, \mathsf{i}_0, +, 1\}_{B_1}\rangle, \langle \mathsf{j}, \{\mathsf{j}_0\}_{B_1}\rangle, \langle \mathsf{k}, \{\mathsf{k}_0\}_{B_1}\rangle\} \end{aligned}$$

2 For notational convenience, the i_0, j_0, and k_0 denote the initial values of i, j, and k at the start of the loop.

Algorithm SOLVERECURRENCES(R)
Compute the solution to the recurrence system R in $recset[H]$
- input: recurrence system R
- output: solution in $recset[H]$
$D := \emptyset$
Remove strongly connected components from R (Sec. 4.5)
for each $\langle v, x\rangle \in R$ in the order defined by $\prec$ **do**
  if $\langle v, x\rangle \in D$ **then**
    $R := R \setminus \{\langle v, x\rangle\}$
  else
    Apply $\mathcal{CR}\#$ rules to x (v is marked loop-variant in x)
    if x is of the form $v + \Psi$ (Ψ is CR or constant) **then**
      $\Phi := \{v_0, +, \Psi\}_H$
    else if x is of the form $v * \Psi$ (Ψ is CR or constant) **then**
      $\Phi := \{v_0, *, \Psi\}_H$
    else if x is of the form $c * v + \Psi$, where c is constant
        or a singleton CR form and Ψ is a constant
        or a polynomial CR form **then**
      $\Phi := \{\phi_0, +, \phi_1, +, \cdots, +, \phi_{k+1}, *, \phi_{k+2}\}_H$, where
        $\phi_0 = v_0; \quad \phi_j = (c-1)\phi_{j-1} + \psi_{j-1}; \quad \phi_{k+2} = c$
    else if x is variable v **then**
      $\Phi := \{v_0\}_H$
    else
      $\Phi := \{v_0, \#, x\}_H$
    endif
    SUBSTITUTE(v, Φ, R)
  endif
enddo
$recset[H] := R$

Algorithm SUBSTITUTE(v, Φ, R)
Substitute all occurrences of variable v with CR form Φ
in the recurrence system R
- input: variable v, CR form Φ, and recurrence system R
- output: updated recurrence system R
Replace $\langle v, x\rangle$ in R with $\langle v, \Phi\rangle$
for each $\langle u, y\rangle \in R$ such that $\langle v, x\rangle \prec \langle u, y\rangle$ **do**
  $D := D \cup \{\langle u, y\rangle\}$
  Create new y' by replacing all v's in y with Φ
  $R := R \cup \{\langle u, y'\rangle\}$
enddo

Figure 7. Algorithm to Compute $recset[H]$

Finally, the CR forms in $recset[H]$ are bounded using alignment and the application of min/max bounds (Sections 3.5 and 3.6). The CR-form bounds are stored in $rangeset[H]$ computed by BOUNDRECURRENCES shown in Figure 8. Applied to the $recset[H]$ of the example loop we obtain the $rangeset[H]$ of the loop

$$\begin{array}{ll}\{ & \langle \mathsf{i}, \{\mathsf{i}_0, \#, \min(\mathsf{i}_0, 1)\}_{B_1}, \{\mathsf{i}_0, \#, \max(\mathsf{i}_0{+}1, 1), +, 1\}_{B_1}\rangle \\ & \langle \mathsf{j}, \{\mathsf{j}_0, \#, \min(\mathsf{i}_0, \mathsf{j}_0)\}_{B_1}, \{\mathsf{j}_0, \#, \max(\mathsf{i}_0, \mathsf{j}_0), +, 1\}_{B_1}\rangle \\ & \langle \mathsf{k}, \{\mathsf{k}_0\}_{B_1}, \{\mathsf{k}_0\}_{B_1}\rangle \quad \}\end{array}$$

where each triple $\langle v, \Psi_{\min}, \Psi_{\max}\rangle$ in the *rangeset* defines CR-form bounds such that the sequence of values of v is bounded by the sequences of $\Psi_{\min}$ and $\Psi_{\max}$.

4.3. Performance and Complexity

The selection of a $B{:}A$ pair from the working set in FINDRECURRENCES affects the performance of the algorithm. A naive selection may lead to an exponential number

Algorithm BOUNDRECURRENCES(H)
Compute $rangeset[H]$ for the loop with (pre)header H
- input: $live[H]$ and $recset[H]$ for (pre)header block H
- output: $rangeset[H]$ with induction variable bounds
$rangeset[H] := \emptyset$
for each variable $v \in live[H]$ **do**
  Use alignment (3.5) and $\min/\max$ bounds (3.6) to compute
  $\Psi_{\min} := \min\{\Phi \mid \langle v, \Phi\rangle \in recset[H]\}$
  $\Psi_{\max} := \max\{\Phi \mid \langle v, \Phi\rangle \in recset[H]\}$
  $rangeset[H] := rangeset[H] \cup \{\langle v, \Psi_{\min}, \Psi_{\max}\rangle\}$
enddo

Figure 8. Algorithm to Compute $rangeset[H]$

of steps where all possible paths in the loop body are traversed. However, we adopt a selection order that ensures that the algorithm is linear in the number of blocks comprising the CFG of the loop body. To achieve this, we keep a counter $count[B]$ with each block B in the loop body. The counter is initialized to the number of successors of a block minus the back edges:

$$count[B] := |\{B_{\text{succ}} \in succ[B] \mid B_{\text{succ}} \text{ is not a loop header}\}|$$

The counter $count[B_{\text{pred}}]$ is decremented in MERGE each time a set of recurrences is saved or combined with another set of recurrences for block B_{pred} in the working set. Only when the counter is zero the block $B{:}A$ is selected from the working set for the next iteration in FINDRECURRENCES. Because this method ensures that each block is visited only once, FINDRECURRENCES is linear in the number of blocks in the CFG of the loop body.

4.4. Soundness and Termination

The FINDRECURRENCES and SOLVERECURRENCES algorithms are sound and terminating.

Theorem 1 *Given a reducible CFG of a loop (nest) with (pre)header block H,* FINDRECURRENCES *terminates.*

Proof. The header block H dominates all blocks with back edges to H by definition of a reducible CFG [1]. Because FINDRECURRENCES starts at blocks with back edges to H and traverses the CFG via predecessor blocks, the algorithm eventually terminates at the dominating block H. □

It is easy to verify that SOLVERECURRENCES terminates. Even though the set R may grow when considering tuples for variable v, a new finite set of tuples is added to R for variables u, $u \neq v$, in inclusion order defined by $\prec$. Thus, tuples in R are considered only once.

Theorem 2 *The general recurrence form of a variable v*

v = v_0
for i = 0 to n–1
  ...
  v = $\alpha * v$ + $p(i)$
  ...
endfor

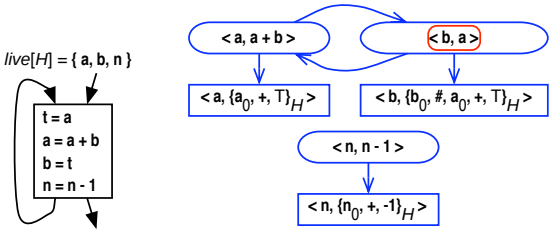


Figure 9. Loop with Cyclic Recurrences

where α is a numeric constant or an i-loop invariant symbolic expression and p is polynomial in i expressed as a CR form $\Psi_i = \{\psi_0, +, \psi_1, +, \cdots, +, \psi_k\}_i$, has the correct CR-form solution Φ_i for v computed by SOLVERECURRENCES

$$\Phi_i = \{\phi_0, +, \phi_1, +, \cdots, +, \phi_{k+1}, *, \phi_{k+2}\}_i$$

where

$$\phi_0 = v_0; \qquad \phi_j = (\alpha - 1)\phi_{j-1} + \psi_{j-1}; \qquad \phi_{k+2} = \alpha$$

Proof. See [36, 37]. □

4.5. Breaking Cyclic Recurrence Relations

SOLVERECURRENCES requires the use-def dependences between the recurrence relations in R to be acyclic. Strongly connected components must be eliminated. Figure 9 illustrates how a cycle is broken by converting a cyclic recurrence relation into a reduced CR form with the lattice element $\top$ (assuming that the initial value $\mathsf{b}_0 \geq 0$). In general, cycles can be broken by replacing a variable's update with an unknown. In many cases we do not need to determine the exact sequence of values as long as the sign and monotonic properties are preserved in the CR forms. In specific cases, pattern recognition can be applied to determine the functions of specific (periodic) sequences, such as the Fibonacci function whose iterative computation is shown in the example loop.

4.6. Bounding the Number of Loop Iterations

To accurately analyze a loop nest from inner loops to the outer loops, the number of loop iterations of the inner nested loops are determined. The LOOPBOUNDS algorithm shown in Figure 10 computes a symbolic lower bound $iter_{\min}[H]$ and upper bound $iter_{\max}[H]$ on the number of loop iterations, provided that the loop has a single pre- or post-test exit condition and a single back edge to the loop (pre)header H. The algorithm uses the CR# algebra rules to convert the loop exit condition to CR form and exploits interval arithmetic to determine the iteration range.

Consider for example the loop nest shown in Figure 11(a). After applying the loop analysis algorithms on the nested loop with header B_3 we have:

Algorithm LOOPBOUNDS(H)
Compute the bounds $iter_{\min}[H$ and $iter_{\max}[H]$ on the number of loop iterations of the loop with (pre)header H
- input: $rangeset[H]$ for (pre)header block H
- output: $0 \leq iter_{\min}[H] \leq iter_{\max}[H] \leq \top$
$iter_{\min}[H] := 0$
$iter_{\max}[H] := \top$
if loop H has a single pre- or post-test exit condition **then**
  Convert exit condition to $expr \leq 0$ by reordering terms
  $range := expr$
  $start := 0$
  for each $\langle v, \Psi_{min}, \Psi_{max}\rangle \in rangeset[H]$ **do**
    if the loop has a post-test exit condition **then**
      $\Psi_{\min} := \mathcal{F}\Psi_{\min}$
      $\Psi_{\max} := \mathcal{F}\Psi_{\max}$
      $start := 1$
    endif
    Replace all v's in $range$ with interval $[\Psi_{\min}, \Psi_{\max}]$
  enddo
  Apply interval arithmetic and $CR\#$ rules to simplify $range$ such that $range = [\Phi_{\min}, \Phi_{\max}]$
  Apply $CR\#_I^{-1}$ rules to convert $\Phi_{\min}$ to closed form $L(I)$
  Apply $CR\#_I^{-1}$ rules to convert $\Phi_{\max}$ to closed form $U(I)$
  Isolate I from $L(I)$ and $U(I)$ such that $L \leq I$ and $U \leq I$
    (if isolation is not possible, set $L := 0$ and $U := \top$)
  $iter_{\min}[H] := \max(start, U)$
  $iter_{\max}[H] := \max(start, L)$
endif

Figure 10. Algorithm to Compute Loop Iteration Bounds $iter_{\min}[H]$ and $iter_{\max}[H]$

$$recset[B_3] = \{\langle \mathsf{i}, \{\mathsf{i}_0, +, 1\}_{B_3}\rangle, \langle \mathsf{j}, \{\mathsf{j}_0\}_{B_3}\rangle, \langle \mathsf{j}, \{\mathsf{j}_0, +, -1\}_{B_3}\rangle\}$$
$$rangeset[B_3] = \{\langle \mathsf{i}, \{\mathsf{i}_0, +, 1\}_{B_3}, \{\mathsf{i}_0, +, 1\}_{B_3}\rangle, \langle \mathsf{j}, \{\mathsf{j}_0, +, -1\}_{B_3}, \{\mathsf{j}_0\}_{B_3}\rangle\}$$

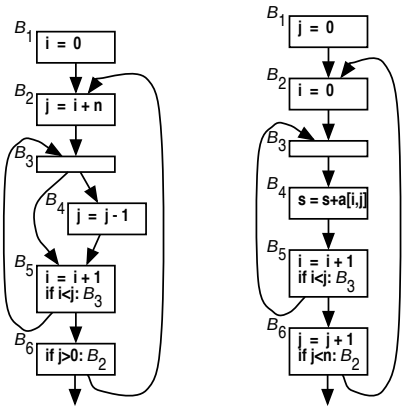
In the initial stage of the LOOPBOUNDS algorithm, the post-test loop exit condition $\mathsf{i} < \mathsf{j}$ is rewritten into $\mathsf{i} - \mathsf{j} + 1 \leq 0$ by reordering terms. Because the loop has a post-test condition, the CR forms of the variables i and j in the $rangeset[B_3]$ are shifted forward, that is

$$\begin{aligned} \mathcal{F}\{\mathsf{i}_0, +, 1\}_{B_3} &= \{\mathsf{i}_0{+}1, +, 1\}_{B_3} \\ \mathcal{F}\{\mathsf{j}_0\}_{B_3} &= \{\mathsf{j}_0\}_{B_3} \\ \mathcal{F}\{\mathsf{j}_0, +, -1\}_{B_3} &= \{\mathsf{j}_0{-}1, +, -1\}_{B_3} \end{aligned}$$

The i and j variables in the condition $\mathsf{i}{-}\mathsf{j}{+}1 \leq 0$ are replaced with their shifted value ranges and simplified using a combination of interval arithmetic and the CR# algebra rules, giving

$$\begin{aligned} &[\{\mathsf{i}_0{+}1, +, 1\}_{B_3}, \{\mathsf{i}_0{+}1, +, 1\}_{B_3}] - [\{\mathsf{j}_0{-}1, +, -1\}_{B_3}, \{\mathsf{j}_0\}_{B_3}] - 1 \\ &= \{\mathsf{i}_0{+}1, +, 1\}_{B_3} + [-\{\mathsf{j}_0\}_{B_3}, -\{\mathsf{j}_0{-}1, +, -1\}_{B_3}] - 1 \\ &= [\{\mathsf{i}_0{+}1, +, 1\}_{B_3} - \{\mathsf{j}_0\}_{B_3} - 1, \{\mathsf{i}_0{+}1, +, 1\}_{B_3} - \{\mathsf{j}_0{-}1, +, -1\}_{B_3} - 1] \\ &= [\{\mathsf{i}_0 - \mathsf{j}_0, +, 1\}_{B_3}, \{\mathsf{i}_0 - \mathsf{j}_0{+}1, +, 2\}_{B_3}] \end{aligned}$$

These CR forms describe the minimum $\Phi_{\min}$ and maximum $\Phi_{\max}$ progressions towards the exit condition, i.e. the loop terminates when the sequences reach zero. To compute the closed-form equivalent of this interval, we convert $\Phi_{\min}$ and $\Phi_{\max}$ to closed forms $L(I)$ and $U(I)$ and then isolate the variable I as follows



(a) *Loop Nest with Conditional Update* (b) *Triangular Loop*

Figure 11. Example Nested Loops

$$\begin{array}{lcll} L(I) \le 0 & \Rightarrow & \mathsf{i}_0 - \mathsf{j}_0 + I \le 0 & \Rightarrow \quad I \le \mathsf{j}_0 - \mathsf{i}_0 \\ U(I) \le 0 & \Rightarrow & \mathsf{i}_0 - \mathsf{j}_0 + 1 + 2I \le 0 & \Rightarrow \quad I \le \lfloor \frac{\mathsf{j}_0 - \mathsf{i}_0 - 1}{2} \rfloor \end{array}$$

Thus, we obtain $iter_{\min}[B_3] = \max(1, \lfloor \frac{\mathsf{j}_0 - \mathsf{i}_0 - 1}{2} \rfloor)$ and $iter_{\max}[B_3] = \max(1, \mathsf{j}_0 - \mathsf{i}_0)$. Indeed, a closer look reveals that the number of loop iterations ranges from $\lfloor \frac{\mathsf{n}-1}{2} \rfloor$ to n with a minimum of one iteration when $\mathsf{n} \le 1$.

4.7. Multidimensional Loops

The MERGELOOP algorithm shown in Figure 12 is used by FINDRECURRENCES to aggregate the updates on induction variables in a nested loop to update the current set of recurrence relations A of the outer loop. MERGLOOP uses the *recset* and iteration bounds of the nested loop to advance the traversal of the CFG of the outer loop immediately to the header H_{nested}. By omitting the blocks in the inner loop, which were already analyzed, the complexity of FINDRECURRENCES is linear in the total number of blocks in the CFG of the loop nest.

The MERGELOOP algorithm computes the minimum $A_{\min}$ and maximum $A_{\max}$ effects of the inner loop on the recurrence system of the outer loop based on the minimum $iter_{\min}[H_{\text{nested}}]$ and maximum $iter_{\max}[H_{\text{nested}}]$ iteration bounds of the nested loop. If the minimum and maximum bounds differ, then the induction variables in the nested loop must form monotonic sequences to ensure accuracy. This is tested with $\mathcal{M}\Phi \neq \perp$, where Φ is the CR form of an induction variable. To compute the aggregate value of the updates to induction variables in the nested loop, the CR forms of the variables need to be updated by the iteration count, which is accomplished by converting the CR forms to closed form with index I set to the loop bound(s).

For example, using the $recset[B_3]$, $iter_{\min}[B_3]$, and $iter_{\max}[B_3]$ of the nested loop shown in Figure 11(a) and

Algorithm MERGELOOP(H_{nested}, A, S)
Updated the recurrence system A with the recurrences of the nested loop and add to the working set S
- input: $recset[H_{\text{nested}}]$, $iter_{\min}[H_{\text{nested}}$, $iter_{\max}[H_{\text{nested}}$, recurrence system A, working set S
- output: updated working set S
$A_{\min} := A$
$A_{\max} := A$
for each $\langle v, \Phi \rangle \in recset[H_{\text{nested}}]$ **do**
  if $\mathcal{M}\Phi = \perp$ and $iter_{\min}[H_{\text{nested}}] \neq iter_{\max}[H_{\text{nested}}]$ **then**
    UPDATE($v, \perp, A_{\min}$)
    UPDATE($v, \perp, A_{\max}$)
  else
    Apply $CR\#_I^{-1}$ rules to convert Φ to closed form $f(I)$
    Replace I's in $f(I)$ with $iter_{\min}[H_{\text{nested}}]$ giving L
    UPDATE($v, L, A_{\min}$)
    Replace I's in $f(I)$ with $iter_{\max}[H_{\text{nested}}]$ giving U
    UPDATE($v, U, A_{\max}$)
  endif
enddo
$A := A_{\min} \cup A_{\max}$
for each predecessor block $B_{\text{pred}} \in pred[B]$
  (excluding the back edges of B) **do**
  if there is a pair $B_{\text{pred}}{:}A' \in S$ **then**
    $S := S \setminus \{B_{\text{pred}}{:}A'\} \cup \{B_{\text{pred}}{:}(A \cup A')\}$
  else
    $S := S \cup \{B_{\text{pred}}{:}A\}$
  endif
enddo

Figure 12. Merging a Nested Loop

computed in the previous section, MERGELOOP takes the recurrence relations A of the outer loop at block B_5, where

$$A := \{\langle v, v \rangle \mid v \in live[B_2]\} = \{\langle \mathsf{i}, \mathsf{i} \rangle, \langle \mathsf{n}, \mathsf{n} \rangle\}$$

Thus, MERGELOOP computes

$$\begin{array}{lcl} A_{\min} & = & \{\langle \mathsf{i}, \mathsf{i} \rangle, \langle \mathsf{n}, \mathsf{n} \rangle\} \\ & \Rightarrow & \{\langle \mathsf{i}, \mathsf{i} + I \rangle, \langle \mathsf{n}, \mathsf{n} \rangle\} \\ & \Rightarrow & \{\langle \mathsf{i}, \mathsf{i} + iter_{\min}[B_3] \rangle, \langle \mathsf{n}, \mathsf{n} \rangle\} \\ & \Rightarrow & \{\langle \mathsf{i}, \mathsf{i} + \max(1, \lfloor \frac{\mathsf{j}-\mathsf{i}-1}{2} \rfloor) \rangle, \langle \mathsf{n}, \mathsf{n} \rangle\} \\ A_{\max} & = & \{\langle \mathsf{i}, \mathsf{i} \rangle, \langle \mathsf{n}, \mathsf{n} \rangle\} \\ & \Rightarrow & \{\langle \mathsf{i}, \mathsf{i} + I \rangle, \langle \mathsf{n}, \mathsf{n} \rangle\} \\ & \Rightarrow & \{\langle \mathsf{i}, \mathsf{i} + iter_{\max}[B_3] \rangle, \langle \mathsf{n}, \mathsf{n} \rangle\} \\ & \Rightarrow & \{\langle \mathsf{i}, \mathsf{i} + \max(1, \mathsf{j}-\mathsf{i}-1) \rangle, \langle \mathsf{n}, \mathsf{n} \rangle\} \end{array}$$

Block B_2 is merged with the union of $A_{\min} \cup A_{\max}$ to compute the $recset[B_2]$ of the outer loop, which is

$$\{\langle \mathsf{i}, \{\mathsf{i}_0, +, \max(0, \lfloor \tfrac{\mathsf{n}_0 - 1}{2} \rfloor)\}_{B_2} \rangle, \langle \{\mathsf{i}, \mathsf{i}_0, +, \max(0, \mathsf{n}_0 - 1)\}_{B_2} \rangle, \langle \mathsf{n}, \{\mathsf{n}_0\}_{B_2} \rangle\}$$

It is evident that variable i's updates in the inner loop contribute to a positive update to the variable in the outer loop anywhere in the range $\max(1, \lfloor \frac{\mathsf{n}_0 - 1}{2} \rfloor)$ to $\max(1, \mathsf{n}_0 - 1)$ making i strictly monotonically increasing.

4.8. Multivariate Forms

Multivariate CR forms for pointers and array index expressions are required to apply dependence testing on pointer access and arrays [36, 38]. To compute multivariate CR forms, we carry along the *recset* of an inner

loop in the working set S of an outer loop and update it in MERGE for each assignment that is merged during the analysis of the outer loop. When the *recset* of the outer loop is computed, we substitute the CR forms (or their ranges) of the induction variables and pointers into the *recset*'s of the inner loops.

Consider for example the triangular loop in Figure 11(b). The $recset[B_3]$ is

$$\{\langle \mathsf{i}, \{\mathsf{i}_0, +, 1\}_{B_3}\rangle, \langle \mathsf{j}, \{\mathsf{j}_0\}_{B_3}\rangle, \langle \mathsf{s}, \{\mathsf{s}_0, +, \mathsf{a}[\{\mathsf{i}_0, +, 1\}_{B_3}, \{\mathsf{j}_0\}_{B_3}]\}_{B_3}\rangle\}$$

When traversing through the outer loop the assignment i=0 causes i_0 to be replaced with 0 in the $recset[B_3]$ of the inner loop. After computing the *recset* of the outer loop, j_0 is replaced with $\{0, +, 1\}_{B_2}$ in $recset[B_3]$ giving

$$\{\langle \mathsf{i}, \{0, +, 1\}_{B_3}\rangle, \langle \mathsf{j}, \{\{0, +, 1\}_{B_2}\}_{B_3}\rangle, \langle \mathsf{s}, \{\mathsf{s}_0, +, \mathsf{a}[\{0, +, 1\}_{B_3}, \{\{0, +, 1\}_{B_2}\}_{B_3}]\}_{B_3}\rangle\}$$

The multivariate forms in the *recset*'s of the inner and outer loop enable dependence testing using our nonlinear CR-based extreme value test [36, 38], such as on $\mathsf{a}[\{0, +, 1\}_{B_3}, \{\{0, +, 1\}_{B_2}\}_{B_3}]$.

4.9. Side-Effects and Aliases

Possible side-effects on variables and variable aliasing is handled by representing unknown variable updates with lattice elements. Thus, a variable updated with an expression that contains aliased variables is effectively updated with a positive ($\top$ or $\top_{\neq}$), a negative ($-\top$ or $-\top_{\neq}$), or an unknown ($\bot$) quantity. Alias analysis and points-to analysis help identify such cases.

5. Applications

In this section we briefly review a number of applications of the recurrence analysis algorithms for loop analysis and optimization problems.

5.1. Induction Variable Substitution

IVS replaces induction variables in a loop nest with closed-form expressions. When the *recset* of a loop contains one solution per variable, which is guaranteed when the updates to induction variables are unconditional, the CR forms of the induction variables in the set are converted to closed-form expressions using the inverse CR# algebra rules shown in Figure 1. For more details, see [34]. In fact, our algorithm is the first to recognize and represent the recurrences of indirect wrap-around variables. These variables are updated with the values of wrap-around variables or other indirect wrap-around variables. Consider for example Figure 13. The recurrence dependence graph shows the ordering of the recurrences in SOLVERECURRENCES to compute a solution. Variable i is a wrap-around

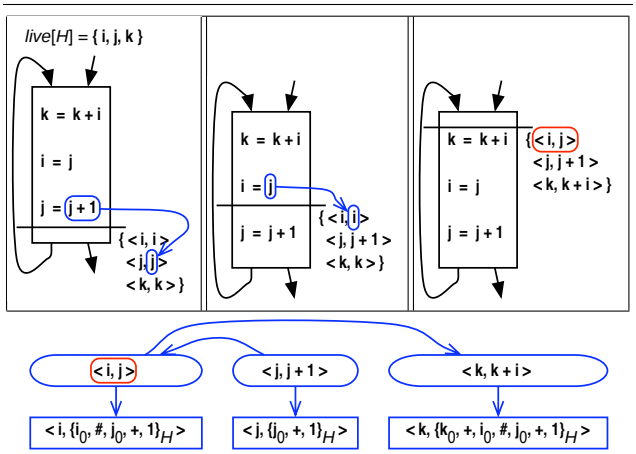


Figure 13. Nonlinear Recurrence Relations with an Indirect Wrap-Around Variable

variable (marked by the oval in recurrence dependence graph in the figure) with delayed CR form $\{\mathsf{i}_0, \#, \mathsf{j}_0, +, 1\}_H$ and k is an indirect wrap-around variable with delayed CR form $\{\mathsf{k}_0, +, \mathsf{i}_0, \#, \mathsf{j}_0, +, 1\}_H$ which is rewritten into $\{\mathsf{k}_0, \#, \mathsf{i}_0{+}\mathsf{k}_0, +, \mathsf{j}_0, +, 1\}_H$ by $CR\#$ rule 30 before applying the $CR\#_I^{-1}$ rules. Applying the $CR\#_I^{-1}$ rules gives the closed forms for loop iteration $I \geq 0$

$$\begin{aligned} \mathsf{i} &= (I = 0)\,?\,\mathsf{i}_0 : \mathsf{j}_0 + I - 1 \\ \mathsf{j} &= \mathsf{j}_0 + I \\ \mathsf{k} &= (I = 0)\,?\,\mathsf{k}_0 : \mathsf{k}_0 + \mathsf{i}_0 + (\mathsf{j}_0 - 1) * (I - 1) + \tfrac{I^2 - I}{2} \end{aligned}$$

where i_0, j_0, and k_0 denote the initial values of the variables at the start of the loop.

5.2. Array Recovery

Induction variables that are address-based pointers can be converted to explicit array accesses, also known as array recovery [13]. Array recovery is similar to IVS. That is, when the *recset* of a loop contains one solution per variable the CR forms of the pointer variables in the set are converted to closed-form expressions using the inverse CR# algebra rules and pointer dereferences are replaced by array accesses. For more details, see [35].

5.3. Idiom Recognition

Our CR-based algorithms facilitate the detection of global reductions such as sums. The CR form provides the pattern of the idiom. For example, the multivariate CR form $\{\mathsf{s}_0, +, \mathsf{a}[\{0, +, 1\}_{B_2}, \{\{0, +, 1\}_{B_2}\}_{B_3}]\}_{B_3}$ computed in Section 4.8 for the triangular loop nest shown in Figure 11(b) denotes $\sum_{0 \leq j < n, 0 \leq i < j} \mathsf{a}[i, j]$. Dependence testing on a is used to check for interference.

5.4. Data Dependence Testing

The details and results of our CR-based nonlinear dependence tests are available in [36, 38]. The new algorithms presented in this report enhance these dependence tests with improved efficiency and accuracy with respect to bounding the sequences of conditionally updated variables and pointers. The ability to determine loop bounds dependent on conditionally updated variables also improves the accuracy of the dependence test. Because our nonlinear dependence test directly utilizes the *recset* and *rangeset* of a loop without IVS or array recovery, the dependence test can be directly applied to compute a dependence hierarchy for array and pointer accesses in reducible CFGs with arbitrary loop structures.

6. Conclusions

In this report we presented a set of CR-based loop analysis algorithms to support compiler analysis and optimizations that depend on induction variable analysis, such as idiom recognition, array date dependence testing, value range analysis, and determining the number of iterations of a loop nest. An important property of the new algorithm is that its complexity is linear in the number of blocks of the CFG of a loop nest regardless of the nesting depth of the loop nest and independent of control flow structure of the loop. We use these algorithms at the basis of a family of compiler-related analysis algorithms for compiler optimizations requiring the analysis of iterative updates to the state of variables in loop-based or recursion-based codes, e.g. for dependence testing and loop optimization.

A. Appendix

A.1. The $\mathcal{V}$, $\mathcal{F}$, and $\mathcal{B}$ Operators

Definition 6 *Let $\Phi_i = \{\phi_0, \odot_1, \phi_1, \odot_2, \ldots, \odot_k, \phi_k\}_i$ be a CR form. The* value *operator is defined by*

$$\mathcal{V}\Phi_i = \phi_0$$

The forward shift operator *is defined by*

$$\mathcal{F}\Phi_i = \{\psi_0, \odot_1, \psi_1, \odot_2, \ldots, \odot_k, \psi_k\}_i$$

with $\psi_j = \phi_j \odot_{j+1} \phi_{j+1}$ for $j = 0, \ldots, k-1$ and $\psi_k = \phi_k$. The backward shift operator *is defined by*

$$\mathcal{B}\Phi_i = \{\psi_0, \odot_1, \psi_1, \odot_2, \ldots, \odot_k, \psi_k\}_i$$

with $\psi_j = \phi_j \odot_{j+1}^{-1} \psi_{j+1}$ for $j = 0, \ldots, k-1$ and $\psi_k = \phi_k$. The inverse operators $\odot^{-1}$ are defined by $+^{-1} = -$ (subtraction) and $*^{-1} = /$ (division).

The forward shift operator was introduced by Bachmann et al. [5]. The backward shift operator was introduced by Van Engelen [33, 34]. The $\mathcal{B}$ operator is not defined for CR forms with the $\#$ delay operator, because $\#$ has no inverse.

References

[1] AHO, A., SETHI, R., AND ULLMAN, J. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Publishing Company, Reading MA, 1985.

[2] AMMERGUALLAT, Z., AND HARRISON III, W. Automatic recognition of induction variables and recurrence relations by abstract interpretation. In *proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (White Plains, NY, 1990), pp. 283–295.

[3] BACHMANN, O. *Chains of Recurrences*. PhD thesis, Kent State University, College of Arts and Sciences, 1996.

[4] BACHMANN, O. Chains of recurrences for functions of two variables and their application to surface plotting. In *Human Interaction for Symbolic Computation* (1996), N. Kajler, Ed., Springer-Verlag.

[5] BACHMANN, O., WANG, P., AND ZIMA, E. Chains of recurrences - a method to expedite the evaluation of closed-form functions. In *proceedings of the International Symposium on Symbolic and Algebraic Computing (ISSAC)* (Oxford, 1994), ACM, pp. 242–249.

[6] BANERJEE, U. *Dependence Analysis for Supercomputing*. Kluwer, Boston, 1988.

[7] BASTOUL, C. Code generation in the polyhedral model is easier than you think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques* (2004). to appear.

[8] BIRCH, J., VAN ENGELEN, R. A., AND GALLIVAN, K. A. Value range analysis of conditionally updated variables and pointers. In *proceedings of Compilers for Parallel Computing (CPC)* (2004), pp. 265–276.

[9] BLUME, W., AND EIGENMANN, R. Demand-driven, symbolic range propagation. In *proceedings of the* 8^{th} *International workshop on Languages and Compilers for Parallel Computing* (Columbus, Ohio, USA, Aug. 1995), pp. 141–160.

[10] BURKE, M., AND CYTRON, R. Interprocedural dependence analysis and parallelization. In *proceedings of the Symposium on Compiler Construction* (1986), pp. 162–175.

[11] COLLARD, J.-F., BARTHOU, D., AND FEAUTRIER, P. Fuzzy array dataflow analysis. In *proceedings of the fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (1995), pp. 92–101.

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

[13] FRANKE, B., AND O'BOYLE, M. Compiler transformation of pointers to explicit array accesses in DSP applications. In *proceedings of the ETAPS Conference on Compiler Construction 2001, LNCS 2027* (2001), pp. 69–85.

[14] GERLEK, M., STOLZ, E., AND WOLFE, M. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form. *ACM Transactions on Programming Languages and Systems (TOPLAS) 17*, 1 (Jan. 1995), 85–122.

[15] GOFF, G., KENNEDY, K., AND TSENG, C.-W. Practical dependence testing. In *proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation (PLDI)* (Toronto, Ontario, Canada, June 1991), vol. 26, pp. 15–29.

[16] HAGHIGHAT, M. R., AND POLYCHRONOPOULOS, C. D. Symbolic analysis for parallelizing compilers. *ACM Transactions on Programming Languages and Systems 18*, 4 (July 1996), 477–518.

[17] HAVLAK, P. *Interprocedural Symbolic Analysis*. PhD thesis, Dept. of Computer Science, Rice University, 1994.

[18] HAVLAK, P., AND KENNEDY, K. Experience with interprocedural analysis of array side effects. pp. 952–961.

[19] KUCK, D. *The Structure of Computers and Computations*, vol. 1. John Wiley and Sons, New York, 1987.

[20] LI, W., AND PINGALI, K. A singular loop transformation framework based on non-singular matrices. *Parallel Programming 22*, 2 (1994), 183–205.

[21] MAYDAN, D. E., HENNESSY, J. L., AND LAM, M. S. Efficient and exact data dependence analysis. In *proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (1991), ACM Press, pp. 1–14.

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

[23] POLYCHRONOPOULOS, C. *Parallel Programming and Compilers*. Kluwer, Boston, 1988.

[24] PSARRIS, K. Program analysis techniques for transforming programs for parallel systems. *Parallel Computing 28*, 3 (2003), 455–469.

[25] PSARRIS, K., AND KYRIAKOPOULOS, K. Measuring the accuracy and efficiency of the data dependence tests. In *proceedings of the International Conference on Parallel and Distributed Computing Systems* (2001).

[26] PSARRIS, K., AND KYRIAKOPOULOS, K. The impact of data dependence analysis on compilation and program parallelization. In *proceedings of the ACM International Conference on Supercomputing (ICS)* (2003).

[27] PUGH, W. Counting solutions to Presburger formulas: How and why. In *proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Orlando, FL, June 1994), pp. 121–134.

[28] REDON, X., AND FEAUTRIER, P. Detection of recurrences in sequential programs with loops. In 5^{th} *International Parallel Architectures and Languages Europe* (1993), pp. 132–145.

[29] RUGINA, R., AND RINARD, M. Symbolic bounds analysis of array indices, and accessed memory regions. In *proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Vancouver, British Columbia, Canada, June 2000), pp. 182–195.

[30] SHEN, Z., LI, Z., AND YEW, P.-C. An empirical study on array subscripts and data dependencies. In *proceedings of the International Conference on Parallel Processing* (1989), vol. 2, pp. 145–152.

[31] SU, E., LAIN, A., RAMASWAMY, S., PALERMO, D., HODGES, E., AND BANERJEE, P. Advanced compilation techniques in the PARADIGM compiler for distributed-memory multicomputers. In *proceedings of the* 9^{th} *ACM International Conference on Supercomputing (ICS)* (Barcelona, Spain, July 1995), ACM Press, pp. 424–433.

[32] TU, P., AND PADUA, D. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In *proceedings of the* 9^{th} *ACM International Conference on Supercomputing (ICS)* (New York, July 1995), ACM Press, pp. 414–423.

[33] VAN ENGELEN, R. Symbolic evaluation of chains of recurrences for loop optimization. Tech. rep., TR-000102, Computer Science Dept., Florida State University, 2000.

[34] VAN ENGELEN, R. Efficient symbolic analysis for optimizing compilers. In *proceedings of the ETAPS Conference on Compiler Construction 2001, LNCS 2027* (2001), pp. 118–132.

[35] VAN ENGELEN, R., AND GALLIVAN, K. An efficient algorithm for pointer-to-array access conversion for compiling and optimizing DSP applications. In *proceedings of the International Workshop on Innovative Architectures for Future Generation High-Performance Processors and Systems (IWIA) 2001* (Maui, Hawaii, 2001), pp. 80–89.

[36] VAN ENGELEN, R. A., BIRCH, J., AND GALLIVAN, K. A. Array data dependence testing with the chains of recurrences algebra. In *proceedings of the IEEE International Workshop on Innovative Architectures for Future Generation High-Performance Processors and Systems (IWIA)* (January 2004), pp. 70–81.

[37] VAN ENGELEN, R. A., BIRCH, J., SHOU, Y., AND GALLIVAN, K. A. Array data dependence testing with the chains of recurrences algebra. Tech. rep., TR-041201, Computer Science Dept., Florida State University, 2004.

[38] VAN ENGELEN, R. A., BIRCH, J., SHOU, Y., WALSH, B., AND GALLIVAN, K. A. A unified framework for nonlinear dependence testing and symbolic analysis. In *proceedings of the ACM International Conference on Supercomputing (ICS)* (2004), pp. 106–115.

[39] WOLFE, M. Beyond induction variables. In *ACM SIGPLAN'92 Conf. on Programming Language Design and Implementation* (San Fransisco, CA, 1992), pp. 162–174.

[40] WOLFE, M. *High Performance Compilers for Parallel Computers*. Addison-Wesley, Redwood City, CA, 1996.

[41] WU, P., COHEN, A., HOEFLINGER, J., AND PADUA, D. Monotonic evolution: An alternative to induction variable substitution for dependence analysis. In *proceedings of the ACM International Conference on Supercomputing (ICS)* (2001), pp. 78–91.

[42] ZIMA, E. Recurrent relations and speed-up of computations using computer algebra systems. In *proceedings of DISCO'92* (1992), LNCS 721, pp. 152–161.

[43] ZIMA, E. Simplification and optimization transformations of chains of recurrences. In *proceedings of the International Symposium on Symbolic and Algebraic Computing* (Montreal, Canada, 1995), ACM.

[44] ZIMA, H., AND CHAPMAN, B. *Supercompilers for Parallel and Vector Computers*. ACM Press, New York, 1990.