On the Equivalence of Two Systems of Affine Recurrence Equations

Denis Barthou[1], Paul Feautrier[2], and Xavier Redon[3]

[1] Université de Versailles Saint-Quentin, Laboratoire PRiSM,
F-78035 Versailles, France,
Denis.Barthou@prism.uvsq.fr,
[2] INRIA, F-78153 Le Chesnay, France,
Paul.Feautrier@inria.fr,
[3] Université de Lille I, École Polytech. Univ. de Lille & Laboratoire LIFL,
F-59655 Villeneuve d'Ascq, France,
Xavier.Redon@eudil.fr,

Abstract. This paper deals with the problem of deciding whether two Systems of Affine Recurrence Equations are equivalent or not. A solution to this problem would be a step toward algorithm recognition, an important tool in program analysis, optimization and parallelization. We first prove that in the general case, the problem is undecidable. We then show that there nevertheless exists a semi-decision procedure, in which the key ingredient is the computation of transitive closures of affine relations. This is a non-effective process which has been extensively studied. Many partial solutions are known. We then report on a pilot implementation of the algorithm, describe its limitations, and point to unsolved problems.

1 Introduction

1.1 Motivation

Algorithm recognition is an old problem in computer science. Basically, one would like to submit a piece of code to an analyzer, and get answers like "Lines 10 to 23 are an implementation of Gaussian elimination". Such a facility would enable many important techniques: program comprehension and reverse engineering, program verification, program optimization and parallelization, hardware-software codesign among others.

Simple cases of algorithm recognition have already been solved, mostly using pattern matching as the basic technique. An example is reduction recognition, which is included in many parallelizing compilers. A reduction is the application of an associative commutative operator to a data set. See [9] and its references. This approach has been recently extended to more complicated patterns by several researchers (see the recent book by Metzger [8] and its references).

In this paper, we wish to explore another approach. We are given a library of algorithms. Let us try to devise a method for testing whether a part of the source

program is equivalent to one of the algorithms in the library. The stumbling block is that in the general case, the equivalence of two programs is undecidable. Our aim is therefore to find sub-cases for which the equivalence problem is solvable, and to insure that these cases cover as much ground as possible.

The first step is to normalize the given program as much as possible. One candidate for such a normalization is conversion to a System of Affine Recurrence Equations (SARE)[3]. It has been shown that *static control programs* [4] can be automatically converted to SAREs. The next step is to design an equivalence test for SAREs. This is the main theme of this paper.

1.2 Equivalence of two SAREs

Suppose we are given two SAREs with their input and output variables. Suppose furthermore that we are given a bijection between the input variables of the two SAREs, and also a bijection between the output variables. In what follows, two corresponding input or output variables are usually denoted by the same letter, one of them being accented.

The two SAREs are equivalent with respect to a pair of output variables, iff the outputs evaluate to the same values provided that the input variables are equal. In order to avoid difficulties with non-terminating computations, we will assume that both SAREs have a schedule.

The equivalence of two SAREs depends clearly on the domain of values used in the computation. In this preliminary work, we will suppose that values belong to the Herbrand universe (or the initial algebra) of the operators occurring in the computation. The Herbrand universe is characterized by the following property:

$$\omega(t_1, \ldots, t_n) = \omega'(t'_1, \ldots, t'_{n'}) \Leftrightarrow \omega = \omega', n = n' \text{ and } t_i = t'_i, i = 1 \ldots n. \quad (1)$$

where ω and ω' are operators and $t_1, \ldots, t_n$, $t'_1, \ldots, t'_{n'}$ are arbitrary terms. The general case is left for future work.

It can be proved that, even in the Herbrand universe, the equivalence of two SAREs is undecidable. The proof is rather technical and can be found in [1]. In Sect. 2 we define and prove a semi-decision procedure which may prove or disprove the equivalence of two SAREs, or fails. In Sect. 3 we report on a pilot implementation of the semi-decision procedure. We then conclude and discuss future work.

2 A Semi-decision Procedure

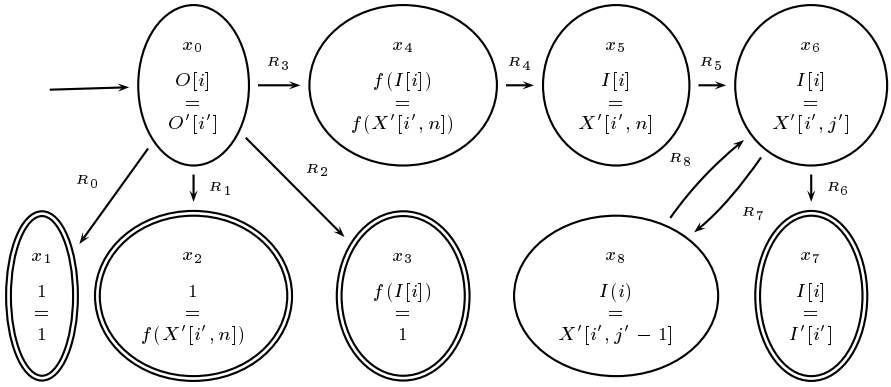
From the above result, we know that any algorithm for testing the equivalence of two SAREs is bound to be incomplete. It may give a positive or negative answer, or fail without reaching a decision. Such a procedure may nevertheless be useful, provided the third case does not occur too often. We are now going to design such a semi-decision procedure. To each pair of SAREs we will associate a memory state automaton (MSA) [2] in such a way that the equivalence of our SAREs can be expressed as problems of reachability in the corresponding MSA. Let us consider the two parametric SAREs (with parameter `n`):

$$\begin{aligned} O[i] &= 1, && i = 0, \\ &= f(I[i]), && 1 \le i \le n, \end{aligned} \quad (2)$$

$$\begin{aligned} O'[i'] &= 1, && i' = 0, \\ &= f(X'[i', n]), && 1 \le i' \le n, \\ X'[i', j'] &= I'[i'], && 0 \le i' \le n, j' = 0, \\ &= X'[i', j' - 1], && 0 \le i' \le n, 1 \le j' \le n. \end{aligned} \quad (3)$$

The reader familiar with systolic array design may have recognized a much simplified version of a transformation known as pipelining or uniformization, whose aim is to simplify the interconnection pattern of the array.

The equivalence MSA is represented by the following drawing. Basically, MSA are finite state automata, where each state is augmented by an *index vector*. Each edge is labelled by a *firing relation*, which must be satisfied by the index vector for the edge to be traversed.



The automaton is constructed on demand from the initial state $O[i] = O'[i']$, expressing the fact that the two SAREs have the same output. Other states are equations between subexpressions of the left and right SARE. The transitions are built according to the following rules: If the lhs of a state is $X[u(i_x)]$, it can be replaced in its successors by $X[i_y]$, provided the firing relation includes the predicate $i_y = u(i_x)$ (R_8). If the lhs is $X[i_x]$ where X is defined by n clauses $X[i] = \omega_k(\ldots Y[u_Y(i)]\ldots), i \in D_k$ then it can be replaced in its n successors by $\omega_k(\ldots Y[u_Y(i_y)]\ldots)$ provided the firing relation includes $\{i_x \in D_k, i_y = i_x\}$ ($R_0, \ldots, R_3$ and R_6, R_7). There are similar rules for the rhs. Note that equations of the successor states are obtained by simultaneous application of rules for lhs and rhs. Moreover, the successors of a state with equation $\omega(...) = \omega(...)$ are states with equations between the parameters of the function ω. The firing relation is in this case the identity relation (R_4). For instance, R_3 and R_8 are:

$$R_3 = \left\{ \begin{bmatrix} i_{x_0} \\ i'_{x_0} \end{bmatrix} \to \begin{bmatrix} i_{x_4} \\ i'_{x_4} \end{bmatrix}, \left\{ \begin{array}{c} i_{x_4} = i_{x_0} \\ i'_{x_4} = i'_{x_0} \\ 1 \le i_{x_0} \le n \\ 1 \le i'_{x_0} \le n \end{array} \right\} \right\}, R_8 = \left\{ \begin{bmatrix} i_{x_8} \\ i'_{x_8} \\ j'_{x_8} \end{bmatrix} \to \begin{bmatrix} i_{x_6} \\ i'_{x_6} \\ j'_{x_6} \end{bmatrix}, \left\{ \begin{array}{c} i_{x_6} = i_{x_8} \\ i'_{x_6} = i'_{x_8} \\ j'_{x_6} = j'_{x_8} - 1 \end{array} \right\} \right\}.$$

States with no successors are final states. If the equation of a final state is always true, then this is a success (x_1, x_7), otherwise this is a failure state (x_2, x_3). The access path from the initial state x_0 to the failure state x_2 is $R_{x_2} = R_1$ and to x_7 is $R_{x_7} = R_3.R_4.R_5.(R_7.R_8)^*.R_6$. When actual relations are substituted to

letters, the reachability relations of these states are:

$$R_{x_2} = \left\{ \begin{bmatrix} i_{x_0} \\ i'_{x_0} \end{bmatrix} \to \begin{bmatrix} i_{x_2} \\ i'_{x_2} \end{bmatrix}, \left\{ \begin{array}{c} i'_{x_2} = i'_{x_0} \\ i_{x_0} = 0 \\ i_{x_2} = 0 \\ 1 \leq i'_{x_2} \leq n \end{array} \right\} \right\}, R_{x_7} \left\{ \begin{bmatrix} i_{x_0} \\ i'_{x_0} \end{bmatrix} \to \begin{bmatrix} i_{x_7} \\ i'_{x_7} \end{bmatrix}, \left\{ \begin{array}{c} i_{x_7} = i_{x_0} \\ i'_{x_7} = i'_{x_0} \\ 1 \leq i_{x_0} \leq n \\ 1 \leq i'_{x_0} \leq n \end{array} \right\} \right\}.$$

Theorem 1. *Two SAREs are equivalent for outputs O and O' iff the equivalence MSA with initial state $O[i] = O'[i']$ is such that all failure states are unreachable and the reachability relation of each success state is included in the identity relation.*

In our example, reachability relations of success states are actually included in the main diagonal (obviously true for R_{x_7} since $i_{x_0} = i'_{x_0}$ implies $i_{x_7} = i'_{x_7}$) and it can be shown that the relations for the failure states are empty (verified for R_{x_2} since $i_{x_0} = i'_{x_0}$ implies $1 \leq 0$). Hence, the two SAREs are equivalent.

It may seem at first glance that building the equivalence MSA and then computing the reachability relations may give us an algorithm for solving the equivalence problem. This is not so, because the construction of the transitive closure of a relation is not an effective procedure [6].

3 Prototype

Our prototype SARE comparator, SAReQ, uses existing high-level libraries. More precisely SAReQ is built on top of SPPoC, an Objective Caml toolbox which provides, among other facilities, an interface to the PolyLib and to the Omega Library. Manipulations of SAREs involve a number of operations on polyhedral domains (handled by the PolyLib). Computing reachability relations of final states boils down to operations such as composition, union and transitive closure on relations (handled by the Omega Library).

The SAREs are parsed using the camlp4 preprocessor for OCaml, the syntax used is patterned after the language Alpha [7]. We give below the text of the two SAREs of section 2 as expected by SAReQ:

```
pipe [n] {
  O[i] = { { i=0 } : 1 ;
           { 1<=i<=n } : f(I[i]) }
  }
```

```
pipe' [n] {
  X'[i',j'] = { { 0<=i'<=n, j'=0 } : I[i'] ;
                { 0<=i'<=n, 1<=j'<=n } : X'[i', j'-1]
                } ;
  O'[i'] = { { i'=0 } : 1 ;
             { 1<=i'<=n } : f(X'[i',n]) }
  }
```

To make the program more friendly a WEB interface is available at the URL `http://sareq.eudil.fr`. This interface gives access to a library of examples, allows the testing of new problems and presents the results in a readable way.

4 Conclusions and Future Work

We believe that our SARE comparator has about the same analytic power as most automatic parallelization tools. It can handle only affine array subscripts

and affine loop bounds. Comparison with the work of Metzger et. al. [8] is difficult, since we do not have access to an implementation. We believe our normal form is more powerful than theirs, since we can upgrade an array to arbitrary dimension, while they are limited to scalar expansion. Also, it does not seem that they can deal with most loop modifications (interchange, skewing, index set splitting) and are limited to loop distibution. On the other hand, provided the program give them the necessary clues, they can handle some forms of associativity and commutativity.

We believe that the most important problem with the present tool is the fact that it cannot use semantical information on the underlying operators. We would like to specify a semantics by a set of simplification rules or algorithms. In the present prototype, the possibility of applying simplifications is limited, since computation rules are never combined. One suggestion is to add "forward" substitution rules. However, we still have to find a heuristics for driving the substitution process.

The present tool is just a building block in a complete program comparator. In the first place, we have to connect it to an array dataflow analyzer ([4], [5]). Secondly, we must build a library of reference algorithms, and this will depends on the application domain. Lastly, many source programs are built by composition from several reference algorithms. Our tool can only be applied if we have delineated the several components, and if we have identified inputs and outputs. At the time of writing, we believe that this has to be handled by heuristics, but this can only be verified by experiments.

References

1. D. Barthou, P. Feautrier, and X. Redon. On the equivalence of two systems of affine recurrence equations. Technical Report RR-4285, INRIA, Oct. 2001.
2. B. Boigelot and P. Wolper. Symbolic verification with periodic sets. In *Proceedings of the 6th International Conference on Computer-Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 55–67. Springer-Verlag, 1994.
3. A. Darte, Y. Robert, and F. Vivien. *Scheduling and automatic Parallelization.* Birkhäuser, 2000.
4. P. Feautrier. Dataflow analysis of scalar and array references. *Int. J. of Parallel Programming*, 20(1):23–53, Feb. 1991.
5. M. Griebl and C. Lengauer. The loop parallelizer LooPo – Announcement. In *9th Languages and Compilers for Parallel Computing Workshop*. Springer, LNCS 1239, 1996. `http://www.fmi.uni-passau.de/cl/loopo`.
6. W. Kelly, W. Pugh, E. Rosser, and T. Shpeisman. Transitive closure of infinite graphs and its applications. *Int. J. of Parallel Programming*, 24(6):579–598, 1996.
7. H. Leverge, C. Mauras, and P. Quinton. The ALPHA language and its use for the design of systolic arrays. *Journal of VLSI Signal Processing*, 3:173–182, 1991.
8. R. Metzger and Z. Wen. *Automatic Algorithm Recognition: A New Approach to Program Optimization.* MIT Press, 2000.
9. X. Redon and P. Feautrier. Detection of scans in the polytope model. *Parallel Algorithms and Applications*, 15:229–263, 2000.