INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

On the Equivalence of Two Systems of Affine Recurrence Equations

Denis Barthou — Paul Feautrier — Xavier Redon

N° 4285

Octobre 2001

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--4285--FR+ENG

INRIA
ROCQUENCOURT

On the Equivalence of Two Systems of Affine Recurrence Equations

Denis Barthou* , Paul Feautrier† , Xavier Redon‡

Thème 1 — Réseaux et systèmes
Projet A3

Rapport de recherche n° 4285 — Octobre 2001 — 23 pages

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 first 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. The proof is by reducing any instance of Hilbert's tenth problem (the solution of Diophantine equations) to the equivalence of two SAREs. We then show that there nevertheless exists a semi-algorithm, in which the key ingredient is the computation of transitive closures of affine relations. This is again an undecidable problem 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.

Key-words: System of Affine Recurrence Equations, program equivalence, program analysis, program optimization

* PRiSM, Université de Versailles Saint-Quentin Denis.Barthou@prism.uvsq.fr
† INRIA, on leave from Université de Versailles Saint-Quentin Paul.Feautrier@inria.fr
‡ LIFL, Université de Lille Flandre Artois Xavier.Redon@eudil.fr

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
Téléphone : +33 1 39 63 55 11 —Télécopie : +33 1 39 63 53 30

Sur l'équivalence de deux systèmes d'équations de récurrence affines

Résumé : Le sujet de cet article est le problème de la comparaison de deux systèmes d'équations de récurrence affines (SARE). Résoudre ce problème permettrait d'améliorer les techniques de reconnaissance d'algorithmes, qui sont un outil important pour l'analyse de programmes, l'optimisation et la parallélisation automatique. Nous prouvons d'abord que le problème est en général indécidable. La preuve utilise la transformation d'une instance du 10ème problème de Hilbert en une question d'équivalence de deux SAREs. Nous montrons ensuite qu'il est cependant possible de construire un semi-algorithme, qui utilise essentiellement le calcul de la fermeture transitive de relations affines. Ceci est également un problème indécidable, pour lequel de nombreuses solutions partielles sont connues. Nous présentons un prototype de comparateur, décrivons ses limitations, et indiquons quelques voies de recherches futures.

Mots-clés : Système d'équations de récurrence affines, équivalence de programmes, analyse de programmes, optimisation de programmes

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: if we know that the program specification asks for Gaussian elimination and the analyzer does not find it, we may suspect an error.
- Program optimization: if we have the necessary items in our library, we may replace lines 10 to 23 above by a hand optimized version, or by a sparse version, or a parallel version. If we are bold enough, we may even replace the relevant part of the code by a completely different implementation, as for instance an iterative solver.
- Hardware-Software Codesign: if we recognize in the source program a piece of code for which we have a hardware implementation (e.g. as a coprocessor or an Intellectual Property) we can remove the code and replace it by an activation of the hardware.

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. It can be detected by normalizing the input program, then matching it with a set of patterns which should include the most common associative operators (addition, multiplication, and, or, max, min ...). See [RF00] and its references. This approach has been recently extended to more complicated patterns by several researchers (see the recent book by Metzger [MW00] 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 part of the source program is equivalent to one of the algorithms in the library. Such a method could be associated to a heuristic system for extracting candidates for recognition from the given program. The stumbling block is that in the general case, the equivalence of two programs is undecidable. Our aim is therefore to find sub-cases where the equivalence problem is solvable, and to insure that such 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). It has been shown that *static control programs* [Fea91] can be automatically converted to SAREs, and such a conversion already was the first step in Redon's method. The next step is to design an equivalence test for SAREs. This is the main theme of this paper.

1.2 Theoretical Background

The basic reference on SAREs is [DRV00]. For background information on term systems, the reader may consult [BN98].

1.2.1 Algebras and Term Systems

A term system is defined by a set $\mathcal{S}$ of symbols, the *sorts*, and a set of operators. Each operator has a signature, which is an expression of the form $S_1 \times \ldots \times S_n \rightarrow S_0$, where the S_i are sorts. n is the arity of the operator. Operators of arity 0 are called constants. We are also given an unbounded supply of variables in each sort.

A term is recursively defined as follows:

1. A variable x of sort S is a term of sort S.
2. A constant of signature $\rightarrow S$ is a term of sort S.
3. If $t_1, \ldots, t_n$ are terms of sort $S_1, \ldots, S_n$ and if ω is an operator of signature $S_1 \times \ldots \times S_n \rightarrow S_0$, then $\omega(t_1, \ldots, t_n)$ is a term of sort S_0.

Case 2 is the special case $n = 0$ of 3 and is included for better understanding. A term without variable is *ground*. In case 3, ω is the *head* of $\omega(t_1, \ldots, t_n)$, and $t_1, \ldots, t_n$ are its immediate sub-terms. The sub-terms of a term are the term itself and the sub-terms of its immediate sub-terms. A term can be considered as a tree, hence we can import such notions as the height or size of a term.

A system of terms being given, a conforming algebra (or a concretization of the term system) is defined by:

- For each sort S, a set U_S, the *carrier* of S.
- For each operator ω of signature $S_1 \times \ldots \times S_n \rightarrow S_0$, a function f_ω from the Cartesian product $U_{S_1} \times \ldots U_{S_n}$ to U_{S_0}.

It is well known that the term system itself can be considered as an algebra conforming to its own signatures. This algebra is known as the Herbrand universe or the initial algebra of the term system.

In any algebra, we have the basic property:

$$a_1 = a'_1, \ldots a_n = a'_n \Rightarrow f(a_1, \ldots, a_n) = f(a'_1, \ldots, a'_n). \tag{1}$$

In the Herbrand universe, we have the converse:

$$\begin{aligned} \omega(t_1, \ldots, t_n) = \omega'(t'_1, \ldots, t'_{n'}) \quad &\Rightarrow \quad \omega = \omega', n = n' \\ &\qquad\quad t_i = t'_i, i = 1, n \end{aligned} \tag{2}$$

This last property may or may not be verified in a concretization of the term system.

1.2.2 Basic Concepts

Let us suppose now that we are given an algebra conforming to a fixed system of sorts and signatures. A System of Affine Recurrence Equations (SARE) is a set of equations of the form:

$$\forall i \in D_k : X[i] = \Phi_k(\ldots Y[u_{Yk}(i)] \ldots). \tag{3}$$

Each so-called variable, X, is a function from domains D_X to elements of one of the carriers of the underlying algebra. In the above formula, $D_k \subseteq D_X$ and $u_{Yk}(D_k) \subseteq D_Y$. Domains are assumed to be unions of bounded Z-polyhedra. Hence, variables can be considered as generalized arrays. Domains can be finite sets, in which case most questions about SAREs become trivial, or parametrically bounded (the domains are finite but their sizes depend on unbounded parameters), or infinite. In this paper, we will suppose that all domains are parametrically bounded.

u_{Yk} (a dependence function) is assumed to be affine. Φ_k is a term in the underlying algebra of the same sort as X. Our notation supposes that each variable has at most one occurrence in the right hand side (rhs) of each equation. This restriction is not important and can be lifted by more involved notations. A SARE must have the single assignment property: if equations k and l have the same variable on their left hand side (lhs) then $D_k \cap D_l = \emptyset$. The domain of a variable X is the union of all D_k such that equation k has X as its lhs. An input variable is a variable which never occurs on the lhs of an equation. Similarly, an output variable never occurs on the rhs of an equation.

A SARE is *uniform* (and is called a SURE) iff:

- All D_k have the same dimension.
- All dependence functions are translations:

  $$u_{Yk}(i) = i - d_{Yk},$$

  where d_{Yk} is a constant integral vector.

A SARE does not describe a computation by itself. Each equation represents a constraint that must be satisfied by the results of any evaluation of the SARE. One possibility is to devise an *evaluation on demand* scheme. Consider equation (3). Suppose we need to know the value of $X[i], i \in D_X$. Let us build a problem tree. The root is $X[i]$. Let k be such that $i \in D_k$. By the unique assignment property, there is exactly one such k. We add the rhs variables of equation k, $Y[u_{Yk}(i)], \ldots$ as sons of node $X[i]$. Since i and the dependence functions are given, $u_{Yk}(i)$ can be effectively computed. We go on in this way until either:

- a node of the tree is an input variable, whose value is known by definition.
- or, the rhs of a node is a constant.
- or, the current node is already present elsewhere in the tree. In this case, there is a dependence cycle in the computation and we decide it fails.

If the construction of the problem tree terminates successfully, then we can back up the values from the leaves, compute the Φ_k functions and find a value for the root. This method has several drawbacks: it generates a lot of overhead, and we are never sure whether a calculation will terminate or not.

Another solution is to build a *schedule*, i.e. a function giving the date $\theta(X, i)$ at which each variable $X[i]$ must be evaluated. A schedule must satisfy the following causality constraint:

$$\forall i \in D_k : \theta(X, i) \geq \theta(Y, u_{Yk}(i)) + 1$$

for all dependences in the SARE. If a schedule is known, we can use it for computing $X[i]$ by the following algorithm:

1. Set $t = 0$.
2. While $t \leq \theta(X, i)$ do:
   - Compute all $Y[j]$ such that $\theta(Y, j) = t$. By the causality constraint, all necessary values are known or have been computed in a preceding step.
   - Increase t by one.

If the domains are bounded, a schedule exists if and only if the given SARE has no dependence cycle. It follows that finding a schedule is equivalent to proving the termination of any demand driven execution of a SARE. The surprising fact is that the scheduling problem for parametrically bounded SAREs is undecidable [SQ93]. However, the scheduling problem for SUREs is decidable [DRV00] and the existence of affine schedules for SAREs is decidable [Fea92a, Fea92b].

1.2.3 Equivalence of two SAREs

The equivalence problem is the following. Suppose we are given two SAREs with their input and output variables. Suppose furthermore that we are given a bijection between the inputs variables of the two SAREs, and also a bijection between the output variables. These pairings must have the property that corresponding variables have the same sort and domain. 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, when started with equal values of the input variables, a demand driven computation stops with equal values of the output variables. Since it is clear that there is a deep relation between equivalence and termination, we will restrict the equivalence problem to the case of two SAREs whose termination is guaranteed (i.e. for which we know a schedule).

The equivalence of two SAREs clearly depends on the underlying algebra.

> Consider a system with one sort, S and two operators, ω with signature $S \times S \rightarrow S$ and the constant α of sort S. Let $\mathbb{N}$ be the set of integers, A be an alphabet with at least two letters, a be one of these letters, and . be concatenation. Then both $\langle \mathbb{N}, +, 1 \rangle$ and $\langle A^*, ., a \rangle$ are algebras conforming to the above system. However, the two SAREs:
>
> $$O = \omega(I, \alpha), \quad O' = \omega(\alpha, I')$$
>
> are equivalent in the first algebra but not in the second one.

It is clear, however, that equivalence in the Herbrand universe implies equivalence in all conforming algebras. From experience in similar situations (compare for instance term unification with associative-commutative unification), it is likely that deciding equivalence in an arbitrary algebra is more complex than deciding equivalence in the Herbrand universe. Hence, in this paper, we will consider the Herbrand case only. The general case is left for future work.

The balance of this paper is as follows. In Sect. 2 we prove that the equivalence of terminating SAREs is undecidable. In Sect. 3 we define and prove a semi-algorithm which may prove or disprove the equivalence of two SAREs, or fails. In Sect. 4 we show that this semi-algorithm is in fact complete for many important sub-cases of the equivalence problem. In Sect. 5 we report on a pilot implementation of the semi-algorithm. We then conclude, point to some simple extensions of the basic algorithm and discuss future work.

2 Undecidability in the Affine Framework

Saouter et. al. [SQ93] have shown that determining computability of a SARE is undecidable. We assume here that the SAREs we compare are computable since they have a schedule. We adapt this proof to the problem of showing that the equivalence of two SAREs is undecidable.

2.1 Principle

We show that the tenth problem of Hilbert can be reduced to the equivalence of two SAREs. This problem is to decide whether a Diophantine equation has an integer positive root or not. Given a Diophantine equation, we build two SAREs and show that deciding their equivalence is equivalent to decide the existence of an integer positive root for the Diophantine equation.

We first give an intuition of the way we connect the tenth problem of Hilbert with the equivalence of SAREs. Any polynomial $P(X_1, \ldots, X_n)$ can be written as the sum of its positive and negative terms:

$$P(X_1, \ldots, X_n) = \sum_k a_k X_1^{\alpha_{1k}} \ldots X_n^{\alpha_{nk}} - \sum_h b_k X_1^{\beta_{1k}} \ldots X_n^{\beta_{nk}}$$

where $a_k, b_k, \alpha_{ik}, \beta_{ik} \geq 0$. We denote $P_+(X_1, \ldots, X_n) = \sum_k a_k X_1^{\alpha_{1k}} \ldots X_n^{\alpha_{nk}}$ the positive part and $P_-(X_1, \ldots, X_n) = \sum_h b_k X_1^{\beta_{1k}} \ldots X_n^{\beta_{nk}}$ its negative one. On the one hand, finding an integer positive root for $P(X_1, \ldots, X_n) = 0$ boils down to find a value of $X_1, \ldots, X_n$ such that P_+ and P_- are equal. An equivalent problem is to decide whether for any positive value $x_1, \ldots, x_n$, $P_+(x_1, \ldots, x_n)$ is different from $P_-(x_1, \ldots, x_n)$. On the other hand, two SAREs are equivalent if and only if for any value of their parameters, their outputs have the same values when their inputs are equal. We link both problems by building two SAREs S and S' which are equivalent if and only if for any positive value of the parameters $x_1, \ldots, x_n$, $P_+(x_1, \ldots, x_n) \neq P_-(x_1, \ldots, x_n)$: both SAREs have only one output O. In S, O depends unconditionally of an input I_1, and in S', O depends on I_2 if only if $P_+(x_1, \ldots, x_n) = P_-(x_1, \ldots, x_n)$, otherwise on I_1. To make this relation appear through the dependences of S', we consider two sets of respectively $P_+(x_1, \ldots, x_n)$ and $P_-(x_1, \ldots, x_n)$ integer points. The dependences iterate alternatively through both sets and when all the integer points have been iterated, then O depends on I_2 otherwise it depends on I_1.

2.2 Example

Let us consider the following example. Let P be the polynomial:

$$P(X_1) = 5X_1 - X_1^2$$

Then $P_+(X_1) = 5X_1$ and $P_-(X_1) = X_1^2$. Let x_1 be a possible value for X_1. We build a SARE with a dependence between O and I_2 if and only if $5x_1 = x_1^2$. The sets considered are $[1, 5] \times [1, x_1]$ and $[1, x_1] \times [1, x_1]$. The variable X iterates through the first set, and Y through the second.

$$\begin{aligned}
O &= X(5, x_1, x_1, x_1) \\
X(i_1, i_2, i_3, i_4) &= \begin{cases} \textbf{if } i_1 > 1 & \textbf{then} \quad Y(i_1 - 1, i_2, i_3, i_4) \\ \textbf{else} & \textbf{if } i_2 > 1 \quad \textbf{then } Y(5, i_2 - 1, i_3, i_4) \\ & \textbf{else} \quad \textbf{if } i_3 = 1 \wedge i_4 = 1 \textbf{ then } I_2 \\ & \textbf{else } I_1 \end{cases} \\
Y(i_1, i_2, i_3, i_4) &= \begin{cases} \textbf{if } i_3 > 1 & \textbf{then } X(i_1, i_2, i_3 - 1, i_4) \\ \textbf{else} & \textbf{if } i_4 > 1 \textbf{ then } X(i_1, i_2, x_1, i_4 - 1) \\ & \textbf{else } I_1 \end{cases}
\end{aligned}$$

We define $g_X(i_1, i_2, i_3, i_4) = 5 - i_1 + 5(x_1 - i_2)$ and $g_Y(i_1, i_2, i_3, i_4) = x_1 - i_3 + x_1(x_1 - i_4)$. $g_X(i_1, i_2, i_3, i_4)$ counts the number of dependence edges from $X(5, x_1, x_1, x_1)$ to $X(i_1, i_2, i_3, i_4)$ in the first set. In particular, $g_X(5, x_1, x_1, x_1) = 0$ and $g_X(1, 1, 1, 1) = 5x_1 - 1$. g_X is incremented by one for each direct dependence between X and Y. Similarly, $g_Y(5, x_1, x_1, x_1) = 0$ and $g_Y(1, 1, 1, 1) = x_1^2 - 1$ and g_Y is incremented by one for each direct dependence between Y and X. Hence if O depends on I_2, $X(5, x_1, x_1, x_1)$ depends on $X(1, 1, 1, 1)$ then $g_X(1, 1, 1, 1) - g_Y(1, 1, 1, 1) = g_X(5, x_1, x_1, x_1) - g_Y(5, x_1, x_1, x_1)$ i.e. $5x_1 = x_1^2$. Moreover, if O does not depend on I_2, it can easily be shown that $g_X(1, 1, 1, 1) - g_Y(1, 1, 1, 1) \neq 0$, meaning that $5x_1 \neq x_1^2$. Proving that this SARE is equivalent to the SARE $O = I_1$ is therefore equivalent to prove that O does not depend on I_2 in the first SARE, i.e. $5x_1 \neq x_1^2$ for all values of x_1. In the general case, there are as many sets to iterate through as monomials in P_+ and P_-.

2.3 General case

Given a polynomial P and a value for its unknowns, its value can be written:

$$\sum_{i}^{p} \prod_{j}^{m_i} v_{ij} - \sum_{i}^{q} \prod_{j}^{n_i} w_{ij}$$

where all v_{ij} and w_{ij} are positive integer values. The products $\prod_j^{m_i} v_{ij}$ represent the different monomials of P_+ where v_{ij} is either the positive coefficient of the monomial or the value of one of the unknowns. For the sake of clarity, we use vectors of coordinates $(v_{ij})_j$ (resp. $(w_{ij})_j$), denoted by v_i (resp. w_i). Moreover, $1'$ denotes a vector where all coordinates are equal to 1 but the first which is 0, $dec(x, v)$ denotes the predecessor of vector x in the enumeration of the integer vectors in the lexicographic interval from 1 to v. More formally,

$$dec(x, v) = (v_1, \ldots, v_{(j-1)}, x_j - 1, x_{(j+1)}, \ldots, x_m)$$

where $j = \min\{k | x_k \neq 1\}$.

The first SARE is given by:

$$O = I_1$$

The second SARE is defined by:

$$\begin{aligned}
O &= X(v_1, \ldots, v_p, w_1, \ldots, w_q) \\
X(x_1, \ldots, x_p, y_1, \ldots, y_q) &= \begin{cases}
\textbf{if } (x_1, \ldots, x_p) \neq (1', \ldots, 1') \textbf{ then} \\
\quad Y(x_1, \ldots, x_{i-1}, x_i', x_{i+1}, \ldots, y_q) \\
\quad \textbf{with } i = \min\{k | x_h \neq 1'\}, \\
\quad \textbf{if } x_i = 1 \textbf{ then } x_i' = (0, 1, \ldots, 1) \\
\quad \textbf{else } x_i' = dec(x_i, v_i) \\
\textbf{else if } (y_1, \ldots, y_p) = (1, \ldots, 1) \textbf{ then } I_2 \\
\textbf{else } I_1
\end{cases} \\
Y(x_1, \ldots, x_p, y_1, \ldots, y_q) &= \begin{cases}
\textbf{if } (y_1, \ldots, y_p) \neq (1', \ldots, 1') \textbf{ then} \\
\quad X(x_1, \ldots, y_{i-1}, y_i', y_{i+1}, \ldots, y_q) \\
\quad \textbf{with } i = \min\{k | y_h \neq 1'\}, \\
\quad \textbf{if } y_i = 1 \textbf{ then } y_i' = (0, 1, \ldots, 1) \\
\quad \textbf{else } y_i' = dec(y_i, w_i) \\
\textbf{else } I_1
\end{cases}
\end{aligned}$$

Let g_{Xi} be the functions defined by:

$$g_{Xi}(x_i) = \sum_{j=1}^{m_i} \left(\prod_{k=1}^{j-1} v_{ik} \right) (v_{ij} - x_{ij})$$

and let

$$g_X(x_1, \ldots, x_p) = \sum_{i}^{p} g_{Xi}(x_i)$$

Likewise, let g_{Yi} be:

$$g_{Yi}(y_i) = \sum_{j=1}^{n_j} \left(\prod_{k=1}^{j-1} w_{ik} \right) (w_{ij} - y_{ij})$$

and

$$g_Y(y_1, \ldots, y_q) = \sum_{i}^{q} g_{Yi}(y_i)$$

Intuitively, g_X (resp. g_Y) counts the number of dependences that have been followed from the output.

Note that $g_X(v_1, \ldots, v_p) = 0$ and $g_X(1', \ldots, 1') = \sum_{i=1}^{p} \prod_{j=1}^{m_i} v_{ij}$. Indeed,

$$\begin{aligned} g_{Xi}(1') &= v_{i1} + \sum_{j=2}^{m_i} \left(\prod_{k=1}^{j-1} v_{ik} \right) (v_{ij} - 1) \\ &= \sum_{j=1}^{m_i} \prod_{k=1}^{j} v_{ik} - \sum_{j=2}^{m_i} \prod_{k=1}^{j-1} v_{ik} \\ &= \prod_{j=1}^{m_i} v_{ij} \end{aligned}$$

Likewise, $g_Y(w_1, \ldots, w_q) = 0$ and $g_Y(1', \ldots, 1') = \sum_{i=1}^{q} \prod_{j=1}^{n_i} w_{ij}$.

Lemma 1 *If $X(x_1, \ldots, x_p, y_1, \ldots, y_q)$ depends on $X(x'_1, \ldots, x'_p, y'_1, \ldots, y'_q)$ then*

$$g_X(x_1, \ldots, x_p) - g_Y(y_1, \ldots, y_q) = g_X(x'_1 \ldots, x'_p) - g_Y(y'_1, \ldots, y'_q)$$

Proof If $X(x_1, \ldots, x_p, y_1, \ldots, y_q)$ depends directly on $Y(x'_1, \ldots, x'_p, y'_1, \ldots, y'_q)$, then $(y_1, \ldots, y_q) = (y'_1, \ldots, y'_q)$. Therefore,

$$g_Y(y_1, \ldots, y_q) = g_Y(y'_1, \ldots, y'_q)$$

$(x_1, \ldots, x_p)$ can be written $(1', \ldots, 1', x_i, \ldots, x_p)$. There are two cases, depending on the value of x_i:

- $x_i = (1, \ldots, 1, x_{ij}, \ldots, x_{im_i})$ then by definition $(x'_1, \ldots, x'_p) = (1', \ldots, 1', x'_i, x_{i+1}, \ldots, x_p)$ with $x'_i = (v_{i1}, \ldots, v_{i(j-1)}, x_{ij} - 1, x_{i(j+1)}, \ldots, x_{im_i})$. Therefore, $g_{Xk}(x'_k) = g_{Xk}(x_k)$ for all $k \neq i$ and

$$g_{Xk}(x'_i) = \prod_{k=1}^{j-1} v_{ik}(v_{ij} - x_{ij} + 1) + \sum_{k=j+1}^{m_i} \left(\prod_{h=1}^{k-1} v_{ih} \right) (v_{ik} - x_{ik})$$

$$g_{Xk}(x_i) = \sum_{k=1}^{j-1} \left(\prod_{h=1}^{k-1} v_{ih} \right) (v_{ik} - 1) + \sum_{k=j}^{m_i} \left(\prod_{h=1}^{k-1} v_{ih} \right) (v_{ik} - x_{ik})$$

  Hence:

$$g_X(x_1, \ldots, x_p) + 1 = g_X(x'_1, \ldots, x'_p)$$

- $x_i = (1, \ldots, 1)$ then by definition, $x'_i = (0, 1, \ldots, 1)$. Hence

$$g_X(x_1, \ldots, x_p) + 1 = g_X(x'_1, \ldots, x'_p)$$

We conclude that

$$g_X(x_1, \ldots, x_p) - g_Y(y_1, \ldots, y_q) + 1 = g_X(x'_1 \ldots, x'_p) - g_Y(y'_1, \ldots, y'_q)$$

Similarly, if $Y(x_1, \ldots, x_p, y_1, \ldots, y_q)$ depends directly on $X(x'_1, \ldots, x'_p, y'_1, \ldots, y'_q)$, then

$$g_X(x_1, \ldots, x_p) - g_Y(y_1, \ldots, y_q) = 1 + g_X(x'_1 \ldots, x'_p) - g_Y(y'_1, \ldots, y'_q)$$

Both equalities imply the result by recurrence on the length of the dependence chain.

This entails the following result:

Lemma 2 *O depends on I_2 if and only if*

$$\sum_{i}^{p} \prod_{j}^{m_i} v_{ij} - \sum_{i}^{q} \prod_{j}^{n_i} w_{ij} = 0$$

Proof The "only if" part: suppose that O depends on I_2. This implies $X(v_1, \ldots, v_p, w_1, \ldots, w_q)$ depends on $X(1', \ldots, 1')$. According to the previous lemma, this entails

$$g_X(v_1, \ldots, v_p) - g_Y(w_1, \ldots, w_q) = g_X(1' \ldots, 1') - g_Y(1', \ldots, 1')$$

Thus:

$$\sum_i^p \prod_j^{m_i} v_{ij} - \sum_i^q \prod_j^{n_i} w_{ij} = 0$$

The "if" part: assume that O does not depend on I_2. According to the definition of the SARE, O depends on I_1. There are two cases:

- If O depends on $X(x_1, \ldots, y_q)$ which in turn depends directly on I_1, then $X(v_1, \ldots, v_p, w_1, \ldots, w_q)$ depends on $X(x_1, \ldots, y_q)$. As $(x_1, \ldots, x_p) = (1', \ldots, 1')$ and $(y_1, \ldots, y_p) \neq (1', \ldots, 1')$ then

  $$g_X(v_1, \ldots, v_p) - g_Y(w_1, \ldots, w_q) = g_X(1', \ldots, 1') - g_Y(y_1, \ldots, y_q)$$

  Moreover, $g_Y(y_1, \ldots, y_q) < \sum_i^q \prod_j^{n_i} w_{ij}$, thus:

  $$0 > \sum_i^p \prod_j^{m_i} v_{ij} - \sum_i^q \prod_j^{n_i} w_{ij}$$

- If O depends on $Y(x_1, \ldots, y_q)$ which in turn depends directly on I_1 then $(y_1, \ldots, y_q) = (1', \ldots, 1')$. Moreover, $X(v_1, \ldots, v_p, w_1, \ldots, w_q)$ depends on some $X(x'_1, \ldots, x'_p, y_1, \ldots, y_q)$ which depends on this $Y(x_1, \ldots, y_q)$. Since

  $$g_X(v_1, \ldots, v_p) - g_Y(w_1, \ldots, w_q) = g_X(x'_1, \ldots, x_p) - g_Y(y_1, \ldots, y_q)$$

  We have:

  $$0 = g_X(x'_1 \ldots, x'_p) - \sum_i^q \prod_j^{n_i} w_{ij}$$

  with $g_X(x'_1 \ldots, x'_p) < \sum_i^p \prod_j^{m_i} v_{ij}$. Therefore,

  $$0 < \sum_i^p \prod_j^{m_i} v_{ij} - \sum_i^q \prod_j^{n_i} w_{ij}$$

In both cases, this proves that when O does not depend on I_2, then the polynomial is not equal to 0.

Theorem 3 *The problem of the equivalence of two SAREs is undecidable.*

Proof Given a Diophantine equation, we can build two SAREs which are equivalent, according to the previous lemma, if and only if the equation has no integer root. This proves that the tenth problem of Hilbert can be reduced to the problem of equivalence of SAREs.

3 A Semi-algorithm for Testing the Equivalence of Two SAREs

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 semi-algorithm may nevertheless be useful, provided the third case does not occur too often. We are going now to design such a semi-algorithm. To each pair of SAREs we will associate a memory state automaton (MSA) in such a way that the equivalence of our SAREs can be expressed as problems of reachability in the corresponding MSA. MSA were introduced by Boigelot and Wolper in [BW94]. Basically, they are finite state automata, where each state is augmented by an integral vector. Now, reachability in an MSA is undecidable, as it should. However, several authors have found special cases in which reachability can be solved, and this gives as many cases in which equivalence can be decided. In fact, we will show that a crucial component of the accessibility computation is the transitive closure operation for affine relations. This operation is not effective, but there are many cases in which it can be computed, and even software packages implementing the operation [KMP+96].

3.1 Memory State Automata

The state of an MSA has two parts: an element of a finite set and a vector of integers. The vector associated to state p is noted v_p and the full state is $\langle p, v_p \rangle$. The dimension of v_p is determined by p and is noted n_p.

A transition in an MSA has three elements: a start state, p, an arrival state q, and a firing relation F_{pq} in $\mathbb{N}^{n_p} \times \mathbb{N}^{n_q}$. A transition from $\langle p, v_p \rangle$ to $\langle q, v_q \rangle$ can occurs only if $\langle v_p, v_q \rangle \in F_{pq}$. There is an edge from p to q in an MSA iff $F_{pq} \neq \emptyset$.

Let $\langle p_0, v_{p_0} \rangle$ be the initial state of the automaton. A state $\langle p, v_p \rangle$ is accessible iff there exists a finite sequence of transitions from the initial state to $\langle p, v_p \rangle$:

$$\begin{aligned} \exists p_1, \ldots, p_n, v_{p_1}, \ldots, v_{p_n} : p_n &= p, \\ \langle v_{p_{i-1}}, v_{p_i} \rangle &\in F_{p_{i-1}, p_i}. \end{aligned} \tag{4}$$

The accessible set of p, noted A_p, is the set of vectors v_p such that $\langle p, v_p \rangle$ is accessible from the initial state.

Computing the Accessibility Relation One method for computing the accessibility relation consists in first characterizing all possible paths in the MSA, then computing the relation associated to each path and "summing" the results. This can be done by associating a letter from a new alphabet to each edge of the MSA. This results in a finite state automaton on the given alphabet. Familiar algorithms [ABB97] allow one to associate to each state a regular expression representing all paths from the initial state to the current state. This expression is built from the letters of the new alphabet by concatenation, alternation and Kleene star.

To obtain the accessibility relation from such a regular expression, replace each letter by the corresponding firing relation, concatenation by relation composition, alternation by union and Kleene star by transitive closure. The accessible set is obtained by composing the result with the accessible set of the initial state.

Theorem 4 *The result of the above construction is the accessibility relation of the given MSA.*

Proof Let A be the alphabet used to label edges of the equivalence MSA. Let ϕ be the application of A into the set of firing relations. It is clear that the set of relations is a monoid, and that ϕ can be extended into a homomorphism from A^* to the monoid of relations with the added properties that:

$$\phi(a \mid b) = \phi(a) \cup \phi(b), a, b \in A^*,$$

$$\phi(a^*) = \phi(a)^*, a \in A^*.$$

In this last equation, the first * represents the Kleene star in A^*, and the second one represents transitive closure of a relation.

Let r be the regular expression representing all paths from the initial state to some state p. Let us first consider a path in the MSA from the initial state to p (in the sense of (4)). Let $F_1, \ldots, F_n$ be the firing relations encountered in this path. Let u_0 (resp. u_n) be the initial (resp. final) state vector. By definition of a path, we have:

$$\langle u_0, u_n \rangle \in F_1 \circ \ldots \circ F_n.$$

Let w be the word of A^* associated to $F_1, \ldots, F_n$. By construction, w is in r. It follows that

$$\phi(w) \subseteq \phi(r).$$

The proof is by induction on the structure of r. The property is trivial if r is the zero length word ϵ or one of the letters of A.

- If $r = s.t$, then w can be split: $w = w_1.w_2$ in such a way that $w_1 \in s$ and $w_2 \in t$. By the induction hypothesis,

  $$\phi(w_1) \in \phi(s),\ \phi(w_2) \in \phi(t),$$

  hence:

  $$\phi(w) = \phi(w_1).\phi(w_2) \subseteq \phi(s).\phi(t) = \phi(s.t) = \phi(r).$$

- If $r = s \mid t$, then w belongs either to s or to t. If for instance $w \in s$,
$$\phi(w) \subseteq \phi(s) \subseteq \phi(s) \cup \phi(t) = \phi(s \mid t).$$
- If $r = s^*$, then $w \in s^\alpha$ for some $\alpha \geq 0$. By the first case,
$$\phi(w) \subseteq \phi(s)^\alpha \subseteq \phi(s)^* = \phi(s^*).$$

Conversely, suppose that $\langle u_0, u_n \rangle \in \phi(r)$. By a process of decomposition similar to the one we used in the preceding proof, one can find a sequence of vectors $u_0, \ldots u_n$ and a sequence of letters $a_1, \ldots, a_n$ such that $a_1 \ldots a_n \in r$ and $\langle u_{i-1}, u_i \rangle \in \phi(a_i)$. We thus have found a path in the MSA. Furthermore, $a_1 \ldots a_n \in r$ implies that the last state of this path is p and hence that $u_n \in A(p)$, QED.

3.2 Construction of the Equivalence MSA

3.2.1 An Intuitive Explanation

Let us be given two SAREs whose equivalence is to be investigated, which will be called here the left SARE and the right SARE. Variables and subscripts in the right SARE will be distinguished by accents. Furthermore, for ease of explanation, let us suppose, without loss of generality, that subscripts in different equations are different.

The equivalence MSA has as state labels equations of the form:
$$e(i) = e'(i'),$$
where $e(i)$ (resp $e'(i')$) is a sub-expression from the left (resp. right) SARE and i and i' are distinct iteration vectors. $\langle i, i' \rangle$ is the state vector. Hence, the firing relation to state q with state vector $\langle j, j' \rangle$ is written $F_{pq}(i, i', j, j')$.

From one point of view, labels are just marks which are used for distinguishing states from each other. On the other hand, they are equations, which may be true or false when actual values $a, a'$are substituted for i and i'. The result of such a substitution is written $e(a) = e'(a')$. Our aim is to build the MSA in such a way that the label of a state becomes true for all vectors of its accessible set. Conversely, if a label is obviously false, its accessible set must be empty. Another special case is that of a label of the form $I(i) = I'(i')$ where I and I' are input variables. We assume here that input variables have arbitrary values, and that the underlying algebra has enough elements to give distinct values for distinct input positions. It follows that if I and I' are non corresponding inputs, then $I(i) = I'(i')$ is one of those obviously false labels, and its accessibility set must be empty. If I and I' correspond, then the label is true for state vectors $\langle a, a' \rangle$ iff $a = a'$. Equivalently, one may say that the accessibility set of this state must be included in the main diagonal, $\{\langle i, i' \rangle | i = i'\}$.

The initial state is $O[i] = O'[i']$ where O and O' are corresponding outputs whose equivalence we want to prove. Its relation is $i = i'$. With this choice, when the state vector of the initial state is in its accessible set, then the label of the state is true. We would like this property to be true for all states. We will build the transitions in such a way that if q_0 is a state and if $q_1, \ldots, q_n$ are its successors, and if v_0 is in $A(q_0)$, and there are transitions from $\langle q_0, v_0 \rangle$ to $\langle q_i, v_i \rangle$, then the label of q_0 is true for v_0 iff the labels of q_i are true for v_i, $i = 1, \ldots, n$.

In summary, we can decide whether the given SAREs are equivalent provided we can construct and test the accessible sets of their equivalence automaton.

Remark Here and everywhere in this section, an equation is true when its lhs and rhs become the same term when variables are replaced by the values computed from the given recurrence equations. More pedantically, we may say that "true" means "provable in the equational theory generated by the given recurrence equations".

3.2.2 States

The sub-expressions of a SARE are its left hand sides and all sub-terms of its right hand side. It is clear that a SARE has only a finite number of sub-expressions. The states of the equivalence automaton for two SAREs are labeled by pairs of sub-terms of the same sort: it follows that there are only a finite number of states.

Each sub-term of a SARE has a subscript vector, which is in fact the subscript vector of the left hand side of the equation it comes from. The vector of a state is obtained by concatenating the vectors of the left and right sub-term.

Names of state vector components are not relevant and can be changed at will provided the change is consistent: names which were distinct before must stay distinct after the change.

3.2.3 Transitions

It remains to explain how to construct the transitions of the equivalence MSA. There are four main cases and a "catch all" case. In the following, $\omega(\dots)$ represents any term which is not a variable, $X[u(i)]$ represents a non-local access to a variable (the function u is not the identity), and $X[i]$ represents a local access.

Simplification From a state with label:

$$\omega(t_1(i),\dots,t_n(i)) = \omega(t'_1(i'),\dots,t'_n(i'))$$

there is a transition to each state $t_k(j) = t'_k(j'), k = 1,n$. The firing relation is $i = j, i' = j'$ for all transitions.

Generalization If the lhs of a state is $X[u(i)]$, it can be replaced in its successors by $X[j]$, provided the firing relation includes the predicate $j = u(i)$. There is a similar rule for the rhs.

Computation If the lhs of a state is $X[i]$ where X is not an input variable, one may construct n left successors, where n is the number of clauses in the definition of X. Successor k has lhs $\Phi_k(j)$ and its firing rule includes $\{i \in \Delta_k, j = i\}$. There is a similar rule for the rhs.

- Rules "Generalization" and "Computation" allow one to build a set of successors for the lhs and a set of successors for the rhs. The actual successors are all labels which can be obtained by combining an expression from the left set and an expression for the right set, with associated firing rules.
- All states which cannot be expanded by one of the above rules are terminal states. There are three kinds of failures:
  - $\omega(\dots) = \omega'(\dots)$ where $\omega \neq \omega'$.
  - $I[i] = \omega(\dots)$ and $\omega'(\dots) = I'[i]$ where I and I' are input variables.
  - $I[i] = I'[i']$, where I and I' are non corresponding input variables.

  and two kinds of successes:
  - $I[i] = I'[i']$, where I and I' are corresponding input variables.
  - $\omega() = \omega()$ where ω is a constant.

The reader may want to check that the above rules cover all cases, and that there is never any ambiguity in the choice of the applicable rule.

3.3 An Example

Let us consider the two SAREs:

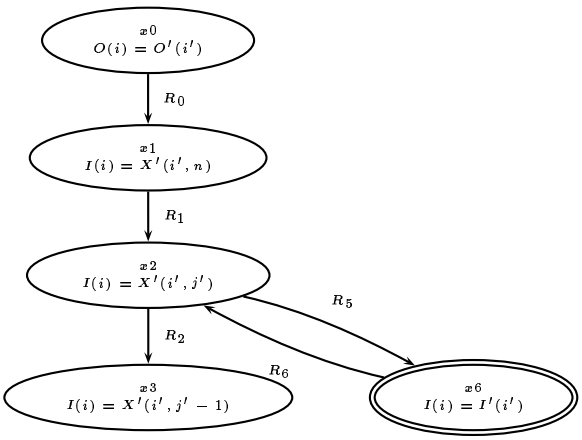
$$O[i] = I[i],\ 0 \leq i \leq n, \tag{5}$$

and

$$\begin{aligned} O'[i'] &= X'[i',n],\ 1 \leq i' \leq n, \\ X'[i',j'] &= I'[i'],\ 1 \leq i' \leq n, j' = 0, \\ &= X'[i',j'-1],\ 1 \leq i' \leq n, 1 \leq j' \leq n. \end{aligned} \tag{6}$$

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.



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. For instance, edge R0 is an application of "Computation" both on the left and right of the initial state. Edges R1 and R6 are generalizations. State $x3$ is a success state, and there are no failure states. Edge names also stand for the corresponding firing relation. For instance, $R6$ is:

$$R_6 = \left\{ \begin{bmatrix} i_1 \\ j_1 \\ j_2 \end{bmatrix} \rightarrow \begin{bmatrix} i'_1 \\ j'_1 \\ j'_2 \end{bmatrix}, \left\{ \begin{matrix} i_1 = i'_1 \\ j_1 = j'_1 \\ j_2 - j'_2 = 1 \end{matrix} \right\} \right\}.$$

The access path from the initial state $x0$ to the unique success state $x3$ is $R0.R1.(R5.R6)^*.R2$. When actual relations are substituted for letters, the result is:

$$x_5 = \left\{ \begin{bmatrix} i_1 \\ j_1 \end{bmatrix} \rightarrow \begin{bmatrix} i'_1 \\ j'_1 \end{bmatrix}, \left\{ \begin{matrix} j_1 = j'_1 \\ i_1 = i'_1 \\ j'_1 \leq n \\ j'_1 \geq 1 \\ i'_1 \leq n \\ i'_1 \geq 1 \end{matrix} \right\} \right\}.$$

3.4 The Equivalence Test and its Proof

Lemma 5 *Let $p : e(i) = e'(i')$ be a non terminal state and let $q : f(j) = f'(j')$ be one of its successors. Let $\langle a, a' \rangle$ be a vector in $A(p)$ and suppose a transition can occur from p with state vector $\langle a, a' \rangle$ to q with state vector $\langle b, b' \rangle$. If $e(a) = e'(a')$, then $f(b) = f'(b')$.*

Proof by cases.

- If the transition is of type "Simplification", for some k, $f(j) = t_k(j), f'(j) = t'_k(j'), a = b, a' = b'$ and the result follows by equation (2).
- If the lhs of p's label is of the form $X[u(i)]$, then $e(i) = X[u(i)], f(j) = X[j], j = u(i)$ which implies $b = u(a)$. The proof is similar for the rhs.
- if the left part of p's label is of the form $X[i]$, there is only one k such that $a \in \Delta_k$. When then have $e(a) = X[a] = \Phi_k(a) = f(b)$.
- If the transition is a combination of rules "Generalization" and rule "Computation", since in all cases we have $e(a) = f(b)$ and $e(a') = f(b')$, from $e(a) = e'(a')$ we deduce $f(b) = f'(b')$.
- There is nothing to prove for terminal nodes.

Lemma 6 *Let θ (resp. θ') be the schedule of the left (resp. right) SARE. The length of a path in the MSA starting from the initial state $O[i] = O'[i']$ with state vector $\langle a, b \rangle$ is bounded by $K\theta(O, a) + K'\theta'(O', a)$ where K and K' are fixed integers which depends only on the left and right SAREs respectively.*

Proof Let us consider an arbitrary path according to the definition in (4). A left move is a move which changes the lhs of the label. It is easy to see that each application of rule "Computation" on the left corresponds to a step in a demand driven calculation of $O[a]$. Hence there can be no more than $\theta(O, a)$ such steps. Between two consecutive applications of "Computation", there can be a number of applications of "Simplification" and one application of "Generalisation". Each application of "Simplication" reduces the height of the lhs by one. Since the lhs is a sub-term of the left SARE, the number of consecutive applications of "Simplification" is bounded by H, the height of the highest term in the left SARE. The number of left moves is thus bounded by $K\theta(O, a)$ with $K = H + 2$. A similar bound holds for the number of right moves, and the result follows from the observation that a path is the union of its left and right moves (notice that a move can be both a left and right move).

Theorem 7 *Two SAREs are equivalent for outputs O and O' iff the equivalence MSA with initial state $O[i] = O[i']$ has the following properties:*

- *all failure states are unaccessible.*
- *the accessibility relation of each success state is included in the identity relation.*

Proof

Only if part Suppose there is a path from the initial state to some terminal state. If the last state is a failure state, then the label of the last state is false when its variables are substituted by the coordinates of the last state vector. If the last state is a success but the state vector is $\langle a, a' \rangle$ with $a \neq a'$, then the label is also false since $I[a] = I'[a'] = I[a']$ would imply a spurious relation between input values.

Consider now the penultimate state in the path. Its label cannot be true, since this would contradict lemma 5. We can back up along the path until we reach the initial state, whose label is false, QED.

If part Let us construct a tree with nodes labeled by a state of the equivalence MSA and a state vector $\langle a, b \rangle$. Nodes can be marked or unmarked. Initially there is one node, $O[i] = O'[i'], \langle a, a \rangle$ which is unmarked.

Select an unmarked node, $p, \langle a, b \rangle$ and mark it. If p is terminal, do nothing. If p is not terminal, add its successors in the MSA as successors in the tree, with state vectors computed according to the firing rules. In the case of rule "Computation" a state vector can be constructed for at most one successor.

The construction of the tree stops when all nodes are marked. The branching factor of the tree (when applying rule "Simplification") is bounded by the maximum arity in the underlying algebra, and its height is bounded by Lemma 6. Hence the tree is finite and its construction terminates.

By hypothesis, all leaves are either of the form $\omega() = \omega()$, or of the form $I(i) = I'(i'), \langle a, a \rangle$, where I and I' are corresponding inputs. Hence, the label of the leaves are true. Let $p, \langle a, b \rangle$ be the deepest node whose label is false. All its sons have true labels. But this is impossible since:

- In case of "Simplification" the truth of the node label follows from (1).
- In case of "Generalization" and "Computation" when applied on the left, the value of the lhs does not change from father to son.
- In case of a combined application of the above rules, neither the lhs nor the rhs change value, and truth of the son label implies truth of the father label.

In all cases, the hypothesis that there is a node whose label is false entails a contradiction. Hence, all labels are true, including the label of the root, QED.

In the case of our example, we have no failure state, and the accessible set of the one success state is actually included in the main diagonal.

It may seem at first glance that construction of the equivalence MSA and then application of Theorem 4 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 process. While we may always write up a formula for the transitive closure, checking whether the corresponding relation is empty or not may entails an unbounded number of tests. Limiting ourselves to affine firing rules is not enough, since the transitive closure of an affine relation may not be affine [KPRS96]. Methods for computing the transitive closure do exists, but they are of an opportunistic nature. One is given a set of rewrite rules that one applies until a solution is found or progress is no longer possible. The interplay of these rules is so complex that it has not yet been possible to find restrictions that would guarantee success. The next section is devoted to a review of these methods.

4 Solvable cases of the Equivalence Problem

In the previous section, we have presented a procedure to prove the equivalence of two SAREs. We recall the main steps here:

1 Build the equivalence MSA associated to the SAREs, following the rules of section 3.2.3. This construction is polynomial in size and time with respect to the size of the SAREs.

2 Decide whether the SAREs are equivalent by testing the reachability of final states, according to Theorem 7.

As shown in section 3.1, reachability relations are rational expressions of firing relations. However, testing the conditions of Theorem 7 is undecidable, since the problem of the equivalence of two SAREs is undecidable (as described in Sect. 2).

In the following, we present algorithms deciding on the reachability of final states under some restrictions. Many other methods and computational tricks can be added to this list. However, outside of their scope of application, these algorithms must give approximate and conservative answers. It is safe to assume that the SAREs are not equivalent when reachability cannot be computed. Moreover, when the exact computation of a reachability relation fails, conservative results are obtained by taking a superset of the relation: the real relation R can be approximated by any relation R' such that $R \subseteq R'$. Indeed, if R' is empty then R is empty too, and if R' is included in the diagonal, so is R.

Our algorithm define a relation between SAREs: two SAREs are provably equivalent if our algorithm answers *yes* when asked to compare them. It would be nice if this relation were an equivalence. It is clear that provable equivalence is symmetric since the constructions rules and accessibility tests are symmetric between left and right. We will show in the next section that a slight optimization of the algorithm makes it reflexive. Transitivity is a more difficult problem and will not be discussed in this paper.

4.1 Comparing rational expressions

By assigning letters to firing relations (ϵ for the identity relation), the reachability relation of a state can be expressed as a rational expression of firing relations. More precisely, firing relations consist in a firing relation for the left SARE and one for the right, independent of each other. Therefore a letter can be assigned to the left relation and one to the right. Reachability relations are then rational expressions over an alphabet of couple of letters. For each state, we call left reachability relation the rational expression obtained by replacing each couple by its first letter. Likewise, we have a right reachability relation.

The following results can be used so as to decide the preconditions of theorem 7 without having to compute the actual relations.

Lemma 8 *If the left and right reachability expressions for a leaf describe the same language, then its reachability relation is included in the diagonal.*

Lemma 9 *If every word of the language described by the reachability relation of a state is a couple (waw_l, wbw_r) and a and b are disjoint relations, then this state is unreachable.*

The proof of these lemmas is straightforward. Using these results, the second step of the algorithm becomes:

- For each success state, check that the left and right reachability relations denote the same language.
- For each failure state, check that every word of the reachability relation is a couple (waw_l, wbw_r) where a and b are disjoint firing relations. Firing relations are disjoint by definition when they come from different clauses of the same variable definition. As a matter of fact, this rule can simplify the automaton if it is

applied during its construction: if the left and right reachability relations of a state are ea and eb with e the same expression and a and b disjoint, then this state is unreachable. There is no need to build its successors.

- If the above tests fail, compute the actual reachability relations and test them against the preconditions in theorem 7.

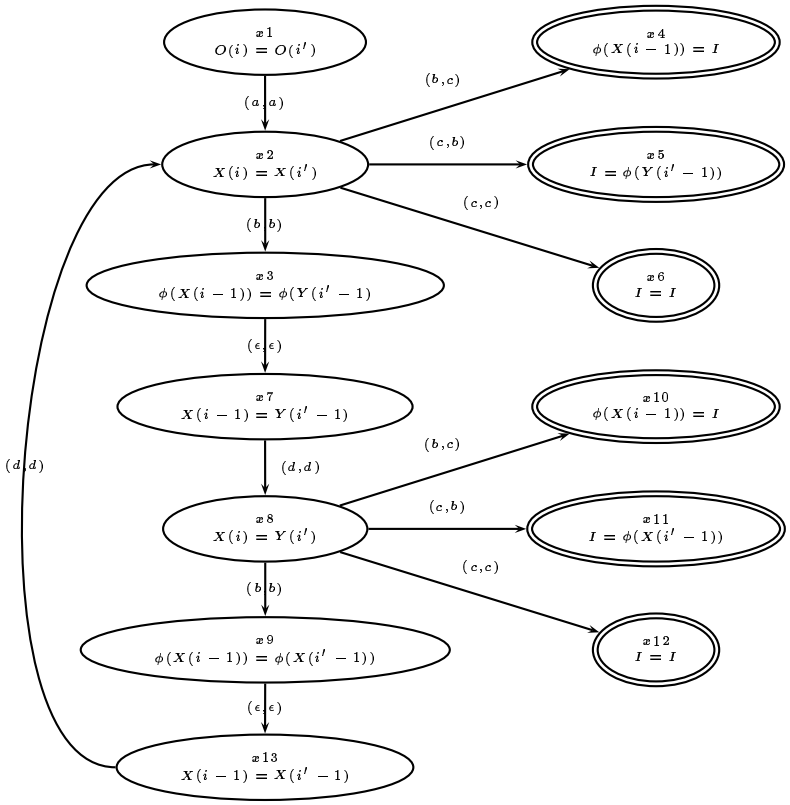
The following SAREs can be proved equivalent by using this method. Likewise, SAREs rewritten with intermediate variables, without changing the domains of the clauses, can be proved equivalent to the original one thanks to rational expressions.

Example: The two SAREs are:

$$\begin{array}{lcl} X(i) & = & I, i \in \Delta_1 \\ & = & \phi(X(f(i))), i \in \Delta_2 \\ O(i) & = & X(i), 1 \leq i \leq n \end{array}$$

$$\begin{array}{lcl} X(i) & = & I, i \in \Delta_1 \\ & = & \phi(Y(f(i))), i \in \Delta_2 \\ Y(i) & = & I, i \in \Delta_1 \\ & = & \phi(X(f(i))), i \in \Delta_2 \\ O(i) & = & X(i), 1 \leq i \leq n \end{array}$$

The equivalence MSA is:



where a stands for $\{(i,i)|1 \leq i \leq n\}$, b for $\{(i,i)|i \in \Delta_2\}$, c for $\{(i,i)|i \in \Delta_1\}$ and d for $\{(i,f(i))\}$. Moreover, according to the definition of the SAREs, we know that $b \cap c = \emptyset$. Next, we compute reachability relations for final states as rational expressions of firing relations;

$x4$	$\begin{pmatrix} a \\ a \end{pmatrix} \begin{pmatrix} bdbd \\ bdbd \end{pmatrix}^* \begin{pmatrix} b \\ c \end{pmatrix}$	$x10$	$\begin{pmatrix} a \\ a \end{pmatrix} \begin{pmatrix} bdbd \\ bdbd \end{pmatrix}^+ \begin{pmatrix} b \\ c \end{pmatrix}$
$x5$	$\begin{pmatrix} a \\ a \end{pmatrix} \begin{pmatrix} bdbd \\ bdbd \end{pmatrix}^* \begin{pmatrix} c \\ b \end{pmatrix}$	$x11$	$\begin{pmatrix} a \\ a \end{pmatrix} \begin{pmatrix} bdbd \\ bdbd \end{pmatrix}^+ \begin{pmatrix} c \\ b \end{pmatrix}$
$x6$	$\begin{pmatrix} a \\ a \end{pmatrix} \begin{pmatrix} bdbd \\ bdbd \end{pmatrix}^* \begin{pmatrix} c \\ c \end{pmatrix}$	$x12$	$\begin{pmatrix} a \\ a \end{pmatrix} \begin{pmatrix} bdbd \\ bdbd \end{pmatrix}^+ \begin{pmatrix} c \\ c \end{pmatrix}$

Failure states $x4, x5, x10$ and $x11$ fulfill conditions of lemma 9, they are unreachable. Moreover, success states $x6, x12$ fulfill conditions of lemma 8, their reachability relation is included in the diagonal. the conclusion is that the two SAREs are equivalent.

We give thereafter a sufficient condition for the approximate relation between SAREs to be reflexive. This condition is that the algorithm must be at least as precise as the method described above.

Theorem 10 *An algorithm for testing the equivalence of two SAREs implements a reflexive relation if it checks the properties of lemmas 8 and 9.*

Proof Suppose that the properties of the two lemmas are verified. We have to show the two conditions of theorem 7.

All failure states are unaccessible Given a failure state p, let w_l and w_r be the words representing respectively the left and right reachability relations.

Let w be the longest common prefix of w_l and w_r for which the label of the corresponding state is an equation between the same terms. This prefix exists since for the initial state (prefix ϵ), the equation $O(i) = O'(i')$ checks the condition. Let q be the state reached with w. Notice that $q \neq p$ since the label of p is not an identity. Let a (resp. b) be the letter for the left (resp. right) firing relation between p and its successor r. Since the left and right part of the label of p are equal and come from the same SARE, if $a \neq b$ then $a \cap b = \emptyset$. Indeed, a and b correspond to two different clauses of the same term. Likewise, if the left part of the label of r is different from the right part, then it implies that $a \neq b$. Thus, in all cases, every word representing the reachability relation on the path from the initial state to p can be written (waw', wbw'') where a and b represent disjoint relations. According to lemma 9, p is unreachable.

The accessibility relation for each success state is included in the identity relation Let p be a success state. Let w_l and w_r be words representing respectively the left and right reachability relation on the path from the initial state to p. If $w_l = w_r$, then Lemma 8 brings the conclusion. Otherwise, $w_l \neq w_r$. According to the previous paragraph, this implies that p is unreachable. In all cases, the property is shown.

4.2 Computing reachability relations

We describe in this part how to compute reachability relations and when it is possible to do it. We assume in the following that the result of this computation uses only Presburger logic.

Transitive closures only appear when there are cycles in the equivalence MSA. The difficulty of the computation of reachability relations depends therefore on the structure of the equivalence MSA. We examine three cases: there is no cycle in the MSA, there are only simple cycles, and the general case (any kind of cycles).

4.2.1 No cycle

According to the construction step of the equivalence MSA, a general expression for a firing relation is:

$$\{(i, i', j, j') | (j, j') = (Ai + a, A'i' + a'), Bi < b, B'i' < b'\} \tag{7}$$

where A, A', B, B' are matrices and a, a', b, b' are constant parametric vectors. This form is closed under a finite number of compositions. More generally, a finite number of unions and compositions of firing relations is a set of the form:

$$\bigcup_k \{(i, i', j, j') | (j, j') = (A_k i + a_k, A'_k i + a'_k), B_k i < b_k, B'_k i' < b'_k\} \tag{8}$$

which is a finite union of polyhedra.

In an MSA without cycles, reachability relations are expressed as a union and composition of firing relations. Therefore, the equivalence of SAREs is decidable when their MSA do not have cycles.

4.2.2 Simple cycles

A simple cycle is a path beginning and ending with the same node, visiting only once the other nodes of the path. Consider a simple cycle $p_1, \ldots, p_n, p_1$ in an equivalence MSA. The cycle relation R obtained by composition of the firing relations of this cycle has the form of a polyhedron given in Expression (7). The following lemma has been proved by Boigelot ([Boi98], theorem 8.53, p.236) and gives a sufficient condition for the computation of R^*:

Lemma 11 *Let R be a relation defined by Expression (7). If there exists p such that A^p and A'^p are diagonalizable and have their eigenvalues in $\{0,1\}$ then R^* is computable in Presburger logic.*

Besides, Boigelot provides an algorithm to check the condition on the matrices and gives the expression of R^*. This expression corresponds to Expression (8).

In an MSA with only simple cycles, it can easily be shown that reachability relations are expressed as a union and composition of firing relations and of transitive closure of cycle relations. Therefore, the equivalence of SAREs is decidable when their MSA has only simple cycles verifying the conditions of Lemma 11.

4.2.3 Complex cycles

In the general case, transitive closures appearing in reachability expressions may be applied to any kind of rational expression. We briefly review two kinds of approach to handle this:

Dynamic for each step of the computation, a list of recipes and computational tricks is used in order to compute a part of the final expression. The conditions for which the computation is exact are not decided beforehand. This approach is very flexible and is proposed by Pugh et al. [KPRS96, Won95]. Moreover, when an approximation is needed, Pugh introduces "unknown" variables corresponding to the non-linear parts of the relations. This leads to supersets of the exact relation and is therefore very useful in the computation of reachability relations.

Static conditions that must be satisfied in order to obtain exact results can be tested before the computation of the relations. This approach has been explored by Boigelot in [Boi98]. Basically, these conditions reflect the cases where a rational expression is equivalent to another one where transitive closure only applies to sets like (7), fulfilling conditions of lemma 11. For instance, in a strongly connected component of the MSA, if

- there is a node belonging to all cycles of the component
- all simple cycles of the SCC verify conditions of lemma 11
- the different transformations of these cycles commute between each other

then the transitive closure of the transformation of the SCC can be computed. Indeed, the computation boils down to the computation of transitive closure of simple cycles relations thanks to commutativity.

Many other methods could be applied to this problem and extensions are left to future work. We present more technical details on the methods we used in the description of our prototype.

5 Prototype

We have built a prototype SARE comparator, SAReQ, in order to validate the above theoretical results and to be able to handle significant examples. We believe the prototype to be reasonably bug free. However, the performances are still less than optimal and limit its use to small to medium kernels. Beside, the tool is not completely in agreement with some of our most recent results. For instance, some of the shortcuts from section 4.1 are not yet implemented. Conversely, some limited semantical extensions have been included, see later.

5.1 Tools

It would had been impossible to build this prototype without some already existing high-level libraries. Manipulation of SAREs involve a number of operations on polyhedral domains, and the computation of the relations for leaf nodes of the equivalence MSA leads to operations such as composition, union and transitive closure on relations. Two well known libraries are able to deal with those operations: the PolyLib [Wil93, CLW97] for the polyhedral domains and the Omega Library [Pug91, KMP+96] for Presburger relations. These are C or C++ libraries, and hence are not suitable for rapid prototyping. Our implementation language is Caml. We have used the SPPoC library [BR01] as an interface to Omega and the PolyLib, and the camlp4 preprocessor [dR] as a builder for the internal representations of SAREs.

5.2 Description of the Prototype

The SAReQ prototype is itself an Objective Caml program of about 2300 lines which is compiled and linked with SPPoC. The Objective Caml program has no true user interface; one has to initialize Objective Caml structures representing SAREs, call the comparison function and decipher the textual result. The SAREs are parsed using the camlp4 preprocessor; the syntax used is patterned after the language Alpha. We give below the text of the two SAREs of section 3.3 as expected by SAReQ:

```
pipe [n] {
  inputs = { } ;
  outputs = { } ;
  declare I[i];
  O[i] = { { 1<=i<=n } :
               I[i] }
  }
```

```
pipe' [n]
  inputs = { } ;
  outputs = { } ;
  declare I[i'];
  X'[i',j'] = { { 1<=i'<=n, j'=0  :
                          I[i'] ;
                { 1<=i'<=n, 1<=j'<=n } :
                          X'[i', j'-1]
                } ;
  O'[i'] = { { 1 <=i'<= n } :
                          X'[i',n] }
  }
```

The result of the execution of the comparison function is a clear text stating if the SAREs are equivalent or not but the more interesting output may be the log structure which give details such as the equivalence MSA, relations of leaf nodes and so on. To make the program more friendly a WEB interface is available at the URL `http:/sareq.eudil.fr`. This interface allows the loading of SARE examples and presents the results in a readable way. The equivalence MSA are drawn with the free package graphviz (`http://www.research.att.com/sw/tools/graphviz/`).

5.3 Details of the Implementation

The SAReQ prototype is an implementation of our algorithms but some modifications or improvements have been made in order to give more pertinent results. Moreover, the first versions of SAReQ were so slow that some optimization were needed.

5.3.1 Input SAREs Characterization

The prototype can analyze classical SAREs, that is a set of equations defined on polyhedral domains. In the definition of an equation several clauses can be used provided that the clause domains are a polyhedral partition of the equation domain. We could have used the modular version of the Alpha language [dD97] but we have chosen to use a simplified syntax (for example we do not type variables). To make a comparison take the example of the `pipe'` SARE given above in our notation and look at the Alpha way of coding it:

```
system pipe' :{n | n>=0}
             (I : {i' | 1<=i'<=n} of real)
       returns (O' : {i' | 1<=i'<=n} of real);
var
  X' : {i',j' | 1<=i'<=n; 1<=j'<=n} of real;
let
  X'[i',j'] =
        case
          {| 1<=i'<=n,j'=0} : I[i'];
          {| 1<=i'<=n,1<=j'<=n} : X'[i',j'-1];
        esac;
  O'[i'] = X'[i',n];
tel;
```

SAReQ is able to handle SAREs with multiple outputs. The correspondence between inputs is given either by their order in an `input` statement or by their order of occurrence in the SARE text. An equivalence MSA is built for each output couple. In fact it is possible to optimize this construction by node sharing, but building distinct graphs makes for more precise comparison results: SAReQ states which outputs are equivalent. Even more information is given: when two corresponding outputs are not equivalent their *score* is displayed. The

score is the fraction of leaf nodes which do not point out difference between outputs on the total number of leaf nodes. To illustrate this notion of score let us present the two SAREs below:

```
outs [n] {
  inputs = { } ;
  outputs = { O1, O2 } ;
  declare I1[i] ;
  declare I2[i] ;
  O1[i] = { { 1<=i<=n } :
        I1[i] + 1 } ;
  O2[i] = { { 1<=i<=n } :
        O1[i] + ( I2[i] + 1 ) }
  }
```

```
outs' [n] {
  inputs = { } ;
  outputs = { O1', O2' } ;
  declare I1[i] ;
  declare I2[i] ;
  O1'[i] = { { 1<=i<=n } :
        I2[i] + 1 } ;
  O2'[i] = { { 1<=i<=n } :
        ( I1[i] + 1 ) + O1'[i] }
  }
```

The result of our prototype is:

```
The SAREs are not equivalent:
     Outputs O1[ i ] and O1'[ i ] are not equal (score 1/2)
     Outputs O2[ i ] and O2'[ i ] are equal
```

The score for outputs `O1` and `O2` is $\frac{1}{2}$ because, in the addition defining `O2` and `O2'`, the order of operands has been changed and our present system knows nothing about commutativity.

5.3.2 Node Expressions Comparison

Another improvement of SAReQ upon the plain algorithm discussed in section 3 is that constant expressions in leaf nodes are not merely tested as character strings. Here a constant is an expression which do not include a reference to an variable. Such expressions are compared using SPPoC symbolic functions which give exact results for polynomial expressions and try to do something with others expressions. Note that an expression e can include equation subscripts i which lead to introduce dummy input array A initialized such that $\forall i,\ A[i] = e(i)$. Even better if the expression e is affine with respect to the equation subscripts and SARE parameters, the expression may be considered as the reference to an array Z initialized with the sequence of integers: $e(i) = Z[e(i)]$.

To get a feeling of the power of these normalizarion techniques, just take a look to the following SAREs which SAReQ find to be equivalent:

```
constant [n] {
  inputs = { } ;
  outputs = { t, u, b, z } ;
  t = n+2 ;
  u[i] = { { 1<=i<=n } :
       i+1 } ;
  a[i] = { { 1<=i<=n } :
       i } ;
  b[i] = { { 1<=i<=n } :
       a[i] } ;
  z[i,j] = { { 1<=i<=n, 1<=j<=n } :
         f(i,g(j)) }
  }
```

```
constant' [n] {
  inputs = { } ;
  outputs = { s, v, b, z' } ;
  s = 1+n+1 ;
  v[i] = { { 1<=i<=n } :
       -1+i+2 } ;
  a[i] = { { 1<=i<=n } :
       n-i+1 } ;
  b[i] = { { 1<=i<=n } :
       a[n-i+1] } ;
  z'[i',j'] = { { 1<=i'<=n, 1<=j'<=n } :
            f(i',g(j')) }
  }
```

5.3.3 Computation of Leaf Relations

The first step of our algorithm is to build the equivalence graphs and the second step consists in computing the relations which link output variables with input variables. In other words we have to compute relations associated with each equivalence MSA leaves by composing relations labeling the graph edges. When a loop occurs in a graph the computation of a transitive closure must be attempted. The sequencing of compositions, unions and transitive closures is given by the resolution of a system of equations whose left hand side are the nodes names x_i and the right hand side are unions of composition of names of incoming edges relations R_j with their source nodes names. For instance the system associated with the equivalence graph of SAREs `pipe` and

`pipe'` is (a dot . represents relation composition and a + represents union of relations):

$$\begin{cases} x_0 &= \epsilon \\ x_5 &= x_4.R_4 \\ x_4 &= x_3.R_3 \\ x_3 &= x_2.R_2 \\ x_2 &= x_1.R_1 + x_6.R_6 \\ x_6 &= x_2.R_5 \\ x_1 &= R_0 \end{cases}$$

Two problems arise with the resolution of this kind of system: it takes time and the size of the solutions may be important and so it may be expensive to compute the Presburger relations. Since finding the minimal solution of such systems is a NP-hard problem we can only come out with an heuristic. The one we use consists of applying, in parallel, all possible variable substitutions on each equation (except, of course, if the variable in the left hand side occurs in the right hand side). We stop as soon as the rhs of an equation has no reference to another equation (but may reference itself); this equation is called "suitable". Arden lemma is applied on all self-referencing equations, breaking the self-reference and introducing a transitive closure operator (unary operator $*$). The suitable equation with smallest RHS is selected and replace its original version in the initial system. The process is repeated until all equations are solved. To sum up, we use a Gaussian elimination algorithm with exhaustive testing of all possible pivots. Obviously this method can lead to an exponential explosion, so the number of explorations is limited by a constant. We can only expected a compact solution for small systems.

The heuristic described above is used to optimize solution size. For improving the performance, the classical "divide and conquer" method is used. It consists in splitting the system in sub-systems according its the decomposition in strongly connected components. Gaussian elimination can be applied to each sub-system, considering the variables not defined in a sub-system as constants. Another benefit of this method is that sub-expressions of the solution tends to be factored into sub-systems and do not need to be computed several times. In the example above the only sub-system with more than one equation is:

$$\begin{cases} x_2 &= x_1.R_1 + x_6.R_6 \\ x_6 &= x_2.R_5 \end{cases}$$

By substitution and application of Arden lemma the following solution is found:

$$\begin{cases} x_2 &= x_1.R_1.(R_5.R_6)* \\ x_6 &= x_2.R_5 \end{cases}$$

The relation for leaf node x_5 can then be computed, using the Omega Library part of SPPoC, by mere replacement of relation names R_i and node names x_j by their values and application of union, composition and transitive closure operators. The whole process is to be done according to the quotient order induced on the SCCs by the dependence graph.

6 Conclusions and Future Work

6.1 An Assessment

We believe that our SARE comparator has about the same analytic power as most automatic parallelization tools. It can handle only affine array subscript and affine loop bounds. It is difficult to decide, from the extent literature, if other algorithm recognizers suffer from the same restriction or not. When these restrictions are not met, both type of tools resort to conservative approximations, with the consequence that a parallelizer may find no parallelism, and that our tool may find no equivalence.

In cases where transitive closures cannot be computed, we may still find a solution, either by overestimating the relation or by solving the problem at the level of symbolic relations, as in section 4.1. The situation is simpler in the case of parallelization, since we usually can compute a schedule in situations where the transitive closure is not computable. In fact, Bill Pugh has coined the term "affine closure" for this kind of situation.

Another way of assessing our work is to consider the following scenario. Assume that we have been transforming a program by hand, and that we want to check the results of our handiwork. We can submit the source program and its transformed version to our tool and check equivalence. We have no formal results on this yet.

However, our experiments show that many familiar transformation can be verified in this way. Let us note first that our system is symmetrical. Hence, if we can verify a transformation, we can also verify its inverse. For instance, since we can verify transformations that insert temporary variables, we can also verify its inverse, forward substitution.

Anything that has to do with memory management, like scalar and array expansion, or with loop distribution, is directly handled by conversion to SAREs. Our comparator handle naturally all kinds of renaming – provided the inputs and outputs are clearly identified –, and all variants of temporary variables introduction and elimination, like redundant expression elimination or hoisting. Dead code elimination is dealt with directly since we build the equivalence MSA on demand. Lastly, all transformations which have to do with iteration space modifications, (loop reversal, loop interchange, loop skewing, scheduling, index set splitting), are within the scope of our comparator.

Non affine transformations, like strip mining or loop coalescing, are not allowed in the present version. Furthermore, since we have not been able to prove transitivity of provable equivalence, our method may fail if one of the SAREs is the result of a sequence of transformations applied to the other SARE.

6.2 Future Work

6.2.1 Introducing Semantics

We believe that the most important problem with the present tool is the fact that it cannot use semantical information on the underlying algebra. Compilers have a limited amount of semantical knowledge, and can use, for instance, associativity, commutativity and distributivity of common operators for optimizing arithmetics expressions. Parallelizers have almost no semantical knowledge, except that some of them are able to detect reductions (by pattern matching) and replace them by calls to optimized parallel routines. We would like to specify a semantics either by a set of axioms or by giving a normalization procedure for expressions, and to have the rules of construction of the MSA modified accordingly. We may for instance require that normalization be applied to both sides of the label before applying "Simplification" (see Sect. 3.2.3). The handling of "constants" in Sect. 5.3.2 is a particular case of this extension. However, we conjecture that it is not sufficient to handle the most general case.

6.2.2 Building a Complete Environment

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 ([Fea91], [Loo97]). 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 can only be handled by heuristics, but this can only be verified by experiments.

References

[ABB97] Jean-Michel Autebert, Jean Berstel, and Luc Boasson. Context-free languages and push-down automata. In *Handbook of Formal Languages*. Springer Verlag, 1997.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and all that*. Cambridge University Press, 1998.

[Boi98] B. Boigelot. *Symbolic Methods for Exploring Infinite State Spaces*. PhD thesis, Université de Liège, 1998.

[BR01] Pierre Boulet and Xavier Redon. Sppoc : manipulation automatique de polyèdres pour la compilation. *TSI*, 8:1–31, 2001.

[BW94] B. Boigelot and P. Wopler. 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.

[CLW97] Philippe Clauss, Vincent Loechner, and Doran K. Wilde. Deriving formulae to count solutions to parameterized linear systems using ehrhart polynomials: Applications to the analysis of nested-loop

programs. Technical Report RR 97-05, Laboratoire Image et Calcul Parallèle Scientifique, April 1997. URL: `http://icps.u-strasbg.fr/PolyLib`.

[dD97] F. de Dinechin. *Systèmes structurés d'equations récurrentes : mise en oeuvre dans le langage Alpha et applications.* PhD thesis, Rennes I, January 1997.

[dR] Daniel de Rauglaudre. Camlp4. URL: `http://caml.inria.fr/camlp4`.

[DRV00] Alain Darte, Yves Robert, and Frédéric Vivien. *Scheduling and automatic Parallelization.* Birkhäuser, 2000.

[Fea91] Paul Feautrier. Dataflow analysis of scalar and array references. *Int. J. of Parallel Programming*, 20(1):23–53, February 1991.

[Fea92a] Paul Feautrier. Some efficient solutions to the affine scheduling problem, I, one dimensional time. *Int. J. of Parallel Programming*, 21(5):313–348, October 1992.

[Fea92b] Paul Feautrier. Some efficient solutions to the affine scheduling problem, II, multidimensional time. *Int. J. of Parallel Programming*, 21(6):389–420, December 1992.

[KMP+96] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott. The Omega Library Interface Guide. Technical report, Dept. of Computer Science, Univ. of Maryland, College Park, 1996. URL: `http://www.cs.umd.edu/projects/omega/`.

[KPRS96] 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.

[Loo97] The polyhedral loop parallelizer: Loopo. `http://www.fmi.uni-passau.de/cl/loopo`, 1997.

[MW00] Robert Metzger and Zhaofang Wen. *Automatic Algorithm Recognition: A New Approach to Program Optimization.* MIT Press, 2000.

[Pug91] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In IEEE, editor, *Proceedings, Supercomputing '91: Albuquerque, New Mexico, November 18–22, 1991*, pages 4–13. IEEE Computer Society Press, 1991.

[RF00] Xavier Redon and Paul Feautrier. Detection of scans in the polytope model. *Parallel Algorithms and Applications*, 15:229–263, 2000.

[SQ93] Yannick Saouter and Patrice Quinton. Computability of recurrence equations. *TCS*, 114, 1993.

[Wil93] Doran K. Wilde. A library for doing polyhedral operations. Technical Report Internal Publication 785, IRISA, Rennes, France, Dec 1993. Also published as INRIA Research Report 2157.

[Won95] D. Wonnacott. *Constraint-Based Array Dependence Analysis.* PhD thesis, University of Maryland, 1995.

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399