Electronic Notes in Theoretical Computer Science 82 No. 2 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume82.html 15 pages

On the Recognition of Algorithm Templates

Christophe Alias [1]

PRiSM
U. of Versailles Saint-Quentin
Versailles, France

Denis Barthou [2]

PRiSM
U. of Versailles Saint-Quentin
Versailles, France

Abstract

This paper deals with the problem of deciding whether a System of Affine Recurrent Equations (SARE) is an instantiation of a SARE template. A solution to this problem would be a step toward algorithm template recognition and open new perspectives in program analysis, optimization and parallelization. The problem is known to be undecidable and we show that there 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. We then describe the limitations of our algorithm and point to unsolved problems.

Keywords: algorithm recognition, SARE, templates, unification, preliminary approach.

1 Introduction

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 optimization: if we have the necessary items in our library, we may replace lines 10 to 23 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

[1] Email: Christophe.Alias@prism.uvsq.fr Fax: +33/0 139 25 40 57
[2] Email: Denis.Barthou@prism.uvsq.fr

part of the code by a completely different implementation, as for instance an iterative solver.

- 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.
- 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 [13] and its references. This approach has been recently extended to more complicated patterns by several researchers (see the recent book by Metzger [12] and its references). In contrast, the starting point of the algorithm recognition procedure proposed by [3,4] and [15] are systems of affine recurrences. From this normal form the method described in [4] is able to find the equivalence of two programs, modulo transformations such as variable hoisting, data expansion/shrinking, affine transformations of the iteration domain, or common sub-expression optimizations.

All these methods recognize only algorithms that have exactly the same semantics as the code they match. Many algorithms however are better described in generic terms, abstracting away the details of implementation. For instance, Gaussian elimination is one instance of the well-known algebraic path problem (APP), as the Warshall's transitive closure algorithm and Floyd's shortest path algorithm are also instances of this same APP. The only difference is the underlying algebraic structure. The only way to handle them by the previous methods is to consider one different pattern for each instantiation. Such generic algorithms are called algorithm templates and many efficient implementations of templates have been proposed. See [16] for matrix manipulations, [11] for graph algorithms or [17] for the APP, to name a few. Compilation of an instantiated pattern consists in compiling the code tailored by the programmer with the optimized code of the template.

The aim of this paper is to propose a method in order to perform some algorithm template recognition and to find out how the instantiation is performed. This important issue have never been tackled before in the framework of algorithm recognition. This preliminary work is based on the framework presented in [3]. As in most algorithm recognition methods, 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* [6] can be automatically converted to SAREs, and such a conversion already was the first step in [13]. The next step is to design an equivalence test between SAREs and SARE templates. This is the main theme of this paper.

Section 2 introduces some essential definitions about SAREs and provides the necessary background on rewriting systems. Section 4 defines the rules in order to match a SARE template with a SARE and Section 5 build the semi-algorithm performing this unification and we conclude in Section 6.

2 Preliminaries

We assume the reader is familiar with term rewriting systems [2]. These preliminaries give the definitions of linearly indexed terms and of SAREs and templates used in the rest of this paper.

2.1 Terms

A signature is a set Σ of function symbols. The set $\mathcal{T}(\Sigma, \mathcal{V})$ of terms built from a signature Σ and a set of variables $\mathcal{V}$ is the smallest set containing $\mathcal{V}$ such that $f(t_1, \dots, t_n) \in \mathcal{T}(\Sigma, \mathcal{V})$ whenever $t_1, \dots, t_n$ are in $\mathcal{T}(\Sigma, \mathcal{V})$. A substitution is a map between $\mathcal{V}$ and $\mathcal{T}(\Sigma, \mathcal{V})$. If σ is a substitution and t a term then $t\sigma$ denotes the result of applying σ to t. We note $\mathcal{D}om(\sigma)$ the set of variables substituted by σ. If two substitutions σ, σ' provide two different values to the same variable then $\sigma \circ \sigma'$ is equal to the error substitution, denoted $\perp$. Composition with the error substitution gives the error substitution.

We consider a signature Σ consisting of functions $\mathcal{F}$, of arrays $\mathcal{A}$ and of the Presburger arithmetic signature $(\mathbb{Z}, +)$. Arrays are constants indexed by affine expressions. We assume that $\mathcal{F} \cap \mathcal{A} = \emptyset$. The set of affine expressions is a set of terms $\mathcal{T}((\mathbb{Z}, +), \mathcal{C})$ with $\mathcal{C}$ a set of index variables. We use an array notation $X[\vec{c}]$ to represent the array X indexed by the vector of expressions c and small letters will denote functions of $\mathcal{F}$. Terms with arrays have been introduced by [8] (called primal terms) in order to finitely represent an infinite number of terms, as it arises in divergent rewriting term systems. Although we are not concerned with divergent rewriting systems, we use them to define systems of affine recurrent equations.

In order to represent templates, we consider a set Φ of function variables, and a set Ψ of array variables, indexed by affine expressions. The set of all variables $\mathcal{V}$ is therefore $\Phi \times \Psi \times \mathcal{C}$. We make a difference between index variables and the other variables, and ground terms denote terms from $\mathcal{T}(\Sigma, \mathcal{C})$. Symbols from Ψ and Φ are chosen among Greek letters.

2.2 Systems of Affine Recurrence Equations

Systems of affine recurrence equations (SARE) are a convenient way to represent algorithms: they can be obtained from imperative programs by reaching definition analysis [6] and already eliminate some syntactic aspects of the programs since they represent the computations with explicit dataflow information. The basic reference on SAREs is [5]. We present here the SAREs as systems of equations between ground terms and for SARE templates, equations between terms.

Definition 2.1 A *System of Affine Recurrence Equations* is a set of ground equations of $\mathcal{T}(\Sigma, \mathcal{V})$, called clauses, of the form:

$$\forall i \in D_k : X[i] = f_k(\dots Y[u_{Yk}(i)] \dots). \tag{1}$$

where i is a vector of $\mathcal{C}$, D_k a domain of integer vectors, $X, Y \in \mathcal{A}$, $f_k \in \mathcal{F}$ and u_{Yk} is a function of $\mathcal{C}$. We introduce the following definitions: free index variables in the equations are called *parameters* of the SARE and all arrays appearing in the SARE are called *SARE variables*; *Domains* are assumed to be union of $\mathbb{Z}$-polyhedra. They can be finite sets, parametrically bounded (the domains are finite but their sizes depend on unbounded parameters), or infinite; D_X denotes the union of all the sets D_k, for all k, defining the clauses of X; Functions u_{Yk} are called *dependence functions* and are affine w.r.t. index variables. SARE variables that do not appear in the left-hand side (lhs) of any clause are called the *inputs* of the SARE. The *outputs* are special SARE variables defined in a lhs of some clauses. Note that there can be several output variables in a SARE.

Moreover a SARE must satisfy the single assignment property, i.e. each value of X is defined uniquely, and we assume that all values of arrays which are not inputs are defined in the SARE.

The example of Fig. 1 illustrates the transformation from a program to a SARE. The output, O is set to the last element of the recurrence in Fig. 1.(b), the input is the array A and the variable `s` has been expanded into a one dimensional array S.

```
s=0;
for (i=1; i<=n; i++)
  s = s+a[i]*a[i];
```

(a)

$$\begin{array}{rl} & O = S[n] \\ i = 0 : & S[i] = A[i] * A[i], \\ 1 \leq i \leq n : & S[i] = S[i-1] + A[i] * A[i], \end{array}$$

(b)

Fig. 1. (a). Sum of the squares (b). Corresponding SARE

A SARE does not describe a computation by itself. One possibility is to build a *schedule*, i.e. a function giving the date $\theta(X, i)$ at which each SARE variable $X[i]$ must be evaluated. A schedule must satisfy the following causality constraint, stating that $X[i]$ cannot be computed before the computation

$$\begin{aligned} & O = T[n], \\ i = 0 : \quad & T[i] = \psi[i], \\ 1 \leq i \leq n : & T[i] = \varphi(T[i-1], \psi[i]), \end{aligned}$$

Fig. 2. Template of a reduction

of the array variables appearing in the rhs:

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

for all dependences in the SARE. If the domains are bounded, a schedule exists iff the given SARE has no dependence cycle. The scheduling problem for parametrically bounded SAREs is undecidable [14]. However, the existence of affine schedules for SAREs is decidable [7]. Note that in general, these schedules have a parametric latency. We only consider in this paper SAREs with a schedule.

A *SARE template* has the same definition as a SARE, except that in the definition of the clauses, f_k is in $\mathcal{F} \cup \Phi$ and Y is in $\mathcal{A} \cup \Psi$.

We can assume, without loss of generality, that equations contains *at most* one functional variable.

3 Matching Problem

Consider two scheduled SAREs S and S', with S a template. Suppose that we are given a bijection between the output variables of the two SAREs and a mapping between input variables. These pairings must have the property that corresponding variables have the same domain.

We define the matching problem between S and S' as follows:

Definition 3.1 The template S matches the SARE S' w.r.t. a pair of output variables if there exists a substitution of the variables of S such that the outputs evaluate to the same values provided the inputs are equal.

For example, the template of Figure 2 matches the SARE of Figure 1 with the substitution: $[\varphi \mapsto \lambda xy.x + y,\ \psi_i \mapsto A[i] * A[i]\ (0 \leq i \leq n)]$, when $n \geq 1$.

This problem depends clearly on the underlying algebra associated to Σ. It is clear, however, that equivalence in the Herbrand universe implies equivalence in all conforming algebras. We only consider in this paper equivalence in the initial algebra. The word problem between two SAREs has been proved undecidable in [4], therefore the matching problem, which is at least as difficult as the equivalence problem, is also undecidable.

4 Matching procedure

This matching procedure provides the rules in order to match a template with a SARE. This boils down to a simultaneous computation of both SARE and template outputs, finding out the substitutions for the variables. We will show that this procedure is correct and complete. However, it may take a parametric number of steps to terminate. The next section proposes to address this termination problem by the construction of an automaton.

4.1 Matching clauses

We call *context* a boolean expression, conjunction of affine relations on index variables.

Consider a SARE on terms of $\mathcal{T}(\Sigma)$ and a template $\mathcal{T}(\Sigma, \mathcal{V})$.

Definition 4.1 A *matching clause* is either:

- a triplet $\sigma, R : t \overset{?}{=} t'$ where σ is a substitution, R a context and $t \overset{?}{=} t'$ an equation between terms of $\mathcal{T}(\Sigma, \mathcal{V})$ and $\mathcal{T}(\Sigma)$;
- $\bot$, the *failure* matching clause;
- or $\top_C$, the *success* matching clause, where C is a set of couples (σ, R) of substitutions σ with their context R.

$\bot$ and $\top_C$ are called *solved forms.* $\bot$ means that the SARE and template are not equivalent and have no unifier. $\top_C$ means that the SARE and template have the set of unifiers given in C, provided their context is true. A solved form is said *unreachable* when its context is false.

We define the following operations on solved forms:

$$\top_{\{(\sigma,R)\}} \vee \top_{\{(\sigma',R')\}} = \top_{\{(\sigma,R),(\sigma',R')\}} \quad \top_{\{(\sigma,R)\}} \vee \bot = \top_{\{(\sigma,R)\}}$$

$$\top_{\{(\sigma,R)\}} \sqcup \top_{\{(\sigma',R')\}} = \top_{\{(\sigma,R),(\sigma',R')\}} \qquad \top_{\{(\sigma,R)\}} \sqcup \bot = \bot$$

$$\top_{\{(\sigma,R)\}} \wedge \top_{\{(\sigma',R')\}} = \top_{\{(\sigma\circ\sigma',R\wedge R')\}} \qquad \top_{\{(\sigma,R)\}} \wedge \bot = \bot$$

Moreover, these rules extend element-wise to matching clauses with sets of unifiers and contexts. For any context R, $\top_{\{(\bot,R)\}} = \bot$, meaning that an error substitution leads to non unifiable terms.

Let S and S' be two SAREs with outputs respectively $O[\vec{\imath}]$ and $O'[\vec{\imath}]$, $\forall \vec{\imath} \in D$. The initial clause associated to the SARE matching problem $S \overset{?}{=} S'$ is:

$$Id, (\vec{\imath} \in D) : O[\vec{\imath}] \overset{?}{=} O'[\vec{\imath}]$$

We say that two clauses $\sigma_1, R_1 : t_1 \overset{?}{=} t'_1$ and $\sigma_2, R_2 : t_2 \overset{?}{=} t'_2$ are equivalent $(\sigma_1, R_1 : t_1 \overset{?}{=} t'_1) \equiv (\sigma_2, R_2 : t_2 \overset{?}{=} t'_2)$ if $\sigma_1 = \sigma_2$ and there exists a renaming of function variables $\sigma : \mathcal{V} \to \mathcal{V}$ such that $\sigma(t_1) =_{syntactic} t_2$ and $\sigma(t'_1) =_{syntactic} t'_2$.

We describe here a matching procedure that takes as input the initial clause and compute its value according to the solved forms.

4.2 Description of the rules

The matching procedure presented here is a combination of the Huet's algorithm to perform syntactic term unification [9] with the algorithm in [4] to test the equivalence of two SAREs. It solves a SARE matching problem $S \overset{?}{=} S'$ by beginning with its initial clause and by repeatedly applying the following rules.

If an expression E in matching clauses is obtained from $\sigma, R : t \overset{?}{=} t'$ by application of one of the transition rules, we denote it by $\sigma, R : t \overset{?}{=} t' \vdash E$. The subscript Q in $\vdash_Q$ indicates the explicit use of the transition rule Q in the deduction step. By $\vdash^*$ we denote the reflexive and transitive closure of $\vdash$.

Rules Decompose, Delete and Conflict are the usual rules of unification and cope with *rigid-rigid* pairs.

Rules Generalize, Compute, Input Failure and Input Success are specific to the computation of the arrays with the SAREs. Rule Generalize rewrites an index expression into a new index variable, which is necessary to apply rule Compute. Rule Compute unfolds the arrays according to their definition in the SAREs, into as many values as there are clauses defining the array. Due to the property of single assignment, note that at most one of the derived contexts is true; We consider that the mapping between inputs of the SARE and the template is the identity (inputs are equal if their indices are the same).

Decompose:
$$\frac{\sigma, R : f(\vec{t}) \overset{?}{=} f(\vec{t'})}{(\sigma, R : t_1 \overset{?}{=} t'_1) \wedge \ldots \wedge (\sigma, R : t_n \overset{?}{=} t'_n)}$$

Delete:
$$\frac{\sigma, R : t \overset{?}{=} t}{\top_{\{(\sigma,R)\}}}$$

Conflict:
$$\frac{\sigma, R : f(\vec{t}) \overset{?}{=} g(\vec{t'})}{\bot}$$
if $f \neq g$ and $R = true$

Generalize:
$$\frac{\sigma, R : S[u(\vec{\imath})] \overset{?}{=} t'}{\sigma, R \wedge (\vec{\imath'} = u(\vec{\imath})) : S[\vec{\imath'}] \overset{?}{=} t'}$$
where $\vec{\imath'}$ is a new vector of index variables

Compute:
$$\frac{\sigma, R : S[\vec{\imath}] \overset{?}{=} t'}{\bigsqcup_{k=1}^{n} \sigma, R \wedge (\vec{\imath} \in D_k) : t_k \overset{?}{=} t'}$$
if $\vec{\imath}$ is vector of index variables, and : $S[\vec{\imath}] = t_k \ (\vec{\imath} \in D_k)$

Empty:
$$\frac{\sigma, \mathbf{false} : t \overset{?}{=} t'}{\top_{\{(\sigma,\mathbf{false})\}}}$$

Substitute:
$$\frac{\sigma \circ [\varphi \mapsto u], R : t \overset{?}{=} t'}{\sigma \circ [\varphi \mapsto u], R : t[\varphi/u] \overset{?}{=} t'}$$
if φ occurs in t

Project/Imitate:

$$\frac{\sigma, R : \varphi(\vec{t}) \overset{?}{=} f(\vec{t'})}{\begin{array}{c} \bigvee_{k=1}^{n} \sigma \circ [\varphi \mapsto \lambda \vec{x}.x_k], R : t_k \overset{?}{=} f(\vec{t'}) \\ \bigvee \sigma \circ [\varphi \mapsto \lambda \vec{x}.f(H_1\vec{x}, ..., H_p\vec{x})], R : \\ f(H_1\vec{t}, ..., H_p\vec{t}) \overset{?}{=} f(\vec{t'}) \end{array}}$$

if $\varphi \notin \mathcal{D}om(\sigma)$ and no clause $e \equiv \sigma, R : \varphi(\vec{t}) \overset{?}{=} f(\vec{t'})$ has been previously computed

Project 1:

$$\frac{\sigma, R : \varphi(\vec{t}) \overset{?}{=} f(\vec{t'})}{\bigvee_{k=1}^{n} \sigma \circ [\varphi \mapsto \lambda \vec{x}.x_k], R : t_k \overset{?}{=} f(\vec{t'})}$$

if $\varphi \notin \mathcal{D}om(\sigma)$ and a clause $e \equiv \sigma, R : \varphi(\vec{t}) \overset{?}{=} f(\vec{t'})$ has been previously computed

Project 2:

$$\frac{\sigma, R : \varphi(\vec{t}) \overset{?}{=} I[\vec{\imath'}]}{\bigvee_{k=1}^{n} \sigma \circ [\varphi \mapsto \lambda \vec{x}.x_k], R : t_k \overset{?}{=} I[\vec{\imath'}]}$$

if $\varphi \notin \mathcal{D}om(\sigma)$

Input Variable:

$$\frac{\sigma, R : \psi_{\vec{\imath}} \overset{?}{=} t'}{\top_{\{(\sigma \circ [(\psi_{\vec{\imath}} \mapsto t', R)], R)\}}}$$

Input Success:

$$\frac{\sigma, R : I[\vec{\imath}] \overset{?}{=} I[\vec{\imath'}]}{\top_{\{(\sigma, R)\}}}$$

if $R \wedge (\vec{\imath} = \vec{\imath'}) = true$

Input Failure:

$$\frac{\sigma, R : I[\vec{\imath}] \overset{?}{=} I[\vec{\imath'}]}{\bot}$$

if $R \wedge (\vec{\imath} \neq \vec{\imath'}) = true$

Rules Project/Imitate, Project, and Substitute are the same as in Huet's algorithm and find unifiers. To prevent from infinite branches, *Imitation* is not applied if this would lead to a matching clause equivalent to a matching clause previously computed (in this case, we apply Project1). Rule Project 2 performs projection, since inputs cannot be computed nor imitated.

4.3 Soundness and Completeness

Consider a SARE S and a template S'. Then the following theorem proves the soundness and completeness of the procedure.

Theorem 4.2 *The instantiation of S' with substitution σ is equivalent to S iff $Id, i \in D : O[i] \overset{?}{=} O'[i] \vdash^* \top_C$ with $(\sigma, true) \in C$.*

Proof. *Only if part:* This part corresponds to the soundness of the procedure. We show by induction on the length of derivation n that $\sigma, R : t \overset{?}{=} t' \vdash^n \top_C$

implies that for any σ in C, t and t' have the same value in the context R with substitution σ. All rules producing directly a solved form are clearly correct. For Compute, suppose that $\sigma, R \cap D_k : t_k \overset{?}{=} t' \vdash^{n-1} \top_{C_k}$ and that for any $\sigma_k \in C_k$, t_k has the same value as t'_k in the context $R \cap D_k$ with the substitution σ_k; as $S[\vec{\imath}] = t_k$ when $i \in D_k$, then $S[\vec{\imath}]$ is equal to t' with the substitution σ_k and in the context $R \cap D_k$. According to the computation on solved forms, this implies that the hypothesis is true for $\sigma, R : S[\vec{\imath}] \overset{?}{=} t'$ and the rule is correct. Likewise, Decompose and Generalize are correct. Moreover, the correction of Project/Imitate, Project1, Project2 and Substitute have been shown by Huet [9]. Therefore, for any n, if $\sigma, R : t = t' \vdash^n \top_C$, then t and t' are equal with any substitution σ in C. The conclusion follows by applying this result to the initial clause.

If part: This part corresponds to the completeness of the procedure. Assume an instantiation of the template with the substitution σ is equivalent to the SARE. Rules Decompose, Delete, Conflict and Empty are complete for the same reasons as in a usual unification setting. Compute corresponds to a rewrite step for S and is complete and Generalize is just an index variable renaming (the value of the rhs does not change for these two rules). The completeness of the rules modifying the substitution are complete as well, due to Huet's algorithm, provided that rule Project1 is never applied. Indeed, this corresponds to the possible substitutions that are not found by Rule Imitate. If the same equation appears in a previous step of the rewriting, modulo a renaming of the variables, then one of the H'_k of a previous application of the rule is applied to a term containing f and H_k, which is defined likewise. Thus f is a symbol repeated in the final substitution. □

5 Semi-algorithm for matching SARE templates

The matching procedure unfolds the recurrences defined by the SARE, thus may take a parametric number of steps. The idea of this semi-algorithm, following the approach of [4], is to implement the procedure with an automaton and analyze the automaton, without executing it, in order to construct the set of unifiers. The automaton, a *Memory State Automaton* (MSA) is described below.

5.1 Memory State Automata

5.1.1 Definition

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 denoted 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 occur 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 reachable iff there exists a finite sequence of transitions from the initial state to $\langle p, v_p\rangle$:

$$\exists p_1, \ldots, p_n, v_{p_1}, \ldots, v_{p_n} : (p_n = p \wedge \langle v_{p_{i-1}}, v_{p_i}\rangle \in F_{p_{i-1},p_i}).$$

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

5.1.2 Computing the Reachability Relation

One method for computing the reachability relation consists of 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 [1] allow one to associate to each state a regular expression representing all paths from the initial state to the current state. To obtain the reachability 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 reachable set is obtained by composing the result with the reachable set of the initial state.

5.2 Construction of the matching MSA

Let us consider a SARE matching problem $S \overset{?}{=} S'$. We assume the index variables of the left SARE will be denoted $\vec{\imath}$, of the right $\vec{\imath'}$.

5.2.1 States

Each state of our matching MSA has two part: a clause $\sigma : t \overset{?}{=} t'$ with σ a substitution, and a vector of integers v_p, which is the concatenation of $\vec{\imath}$ and $\vec{\imath'}$.

The *initial state* is $Id : O[\vec{\imath}] \overset{?}{=} O'[\vec{\imath'}]$, where O and O' are corresponding outputs of S and S'. Its reachable set is $\{\langle \vec{\imath}, \vec{\imath'}\rangle | \vec{\imath} = \vec{\imath'}\}$.

The *final states* are either:

- $\top_{\{(\sigma,E)\}}$, where σ is a substitution, and E is a context *i.e.* conditions on parameters for which σ is valid ;
- $\perp$.

5.2.2 Transitions

In order to make the correspondence between automaton and rewriting rules, the firing relations will correspond to the relations between the index variables defined by the contexts.

We describe thereafter the main transitions:

- *Decompose* From a state with label $\sigma : f(\vec{t}(\vec{\imath})) \stackrel{?}{=} f(\vec{t'}(\vec{\imath'}))$ starts a transition to each state $\sigma : t_k(\vec{\imath}) \stackrel{?}{=} t'_k(\vec{\imath'})$, with the firing relation $Id : \{\vec{\imath} \rightarrow \vec{\imath},\, \vec{\imath'} \rightarrow \vec{\imath'}\}$. All these transitions constitute an *and*-branching.
- *Generalize* From a state with label $\sigma : X[u(\vec{\imath})] \stackrel{?}{=} t'$ starts a transition to state $\sigma : X[\vec{\imath}] \stackrel{?}{=} t'$, with the firing relation: $\{\vec{\imath} \rightarrow u(\vec{\imath}),\, \vec{\imath'} \rightarrow \vec{\imath'}\}$, as seen in section 4. There is a similar rule for the rhs.
- *Compute* From a state with label $\sigma : X[\vec{\imath}] \stackrel{?}{=} t'$ starts an transition to each state $\sigma : t_k(\vec{\imath}) \stackrel{?}{=} t'$, with the firing relation: $\{\vec{\imath} \rightarrow \vec{\imath},\, \vec{\imath'} \rightarrow \vec{\imath'},\, \vec{\imath} \in D_k\}$. All these transitions constitute an $\sqcup$-branching. There is a similar rule for the rhs.
- *Huet's rules* produce an *or*-branching between each *Project* and *Imitate*. The firing relation is Id since they do not modify the index variables.
- *Input Variable* From a state with label: $\sigma : \psi_{\vec{\imath}} \stackrel{?}{=} t'$ starts a transition to $\top_{\{(\sigma\circ[(\psi_{\vec{\imath}} \mapsto t', E)], E)\}}$. Firing relation is Id. E will be computed during the MSA analysis.
- *Input Failure/Input Success* From a state with label: $\sigma : I[\vec{\imath}] \stackrel{?}{=} I[\vec{\imath'}]$ starts a transition to $\top_{\{(\sigma,E)\}}$ with the firing relation $\{\vec{\imath} \rightarrow \vec{\imath}, \vec{\imath'} \rightarrow \vec{\imath'}, \vec{\imath} = \vec{\imath'}\}$, and a transition to $\perp$ with firing relation $\{\vec{\imath} \rightarrow \vec{\imath}, \vec{\imath'} \rightarrow \vec{\imath'}, \vec{\imath} \neq \vec{\imath'}\}$. E will be computed during the MSA analysis.

Let us prove that the MSA defined has a finite number of states:

Proposition 5.1 *Let $S \stackrel{?}{=} S'$ be a SARE matching problem, with S a template and S' a SARE and $\mathcal{A}$ be its corresponding MSA. The number of states of $\mathcal{A}$ is finite.*

Proof. States of the automaton are of the form $\sigma : t \stackrel{?}{=} t'$. t' is one of the possible subterms of S', which are in finite number. t is either a subterm of S, or a function variable which takes subterms of S as arguments, or an array variable which has index variables as arguments. The number of function variables appearing in the template is finite. Assume there exists a parametric number of new function variables. Then Imitate is applied a parametric number of times. Because Imitate modifies the current substitution σ, these applications can only appear in a parametric length branch. Let $\varphi_i(\vec{t_i}) \stackrel{?}{=} t'_i$, $1 \leq i \leq p$ denote them. The restriction of Imitate rule ensures that $\varphi_i(\vec{t_i}) \stackrel{?}{=} t'_i \not\equiv \varphi_j(\vec{t_j}) \stackrel{?}{=} t'_j$, if $i \neq j$. This entails that $\vec{t_i} \neq \vec{t_j}$, or $t'_i \neq t'_j$. Consequently, we can find a parametric number of distinct sub-terms of S (or S'). This would lead to a contradiction. Thus, function variables are in finite number. □

5.3 Analysis of the matching MSA

We have now to analyze the matching MSA in order to decide whether the SARE is an instantiation of the template, and to find out the set of unifiers. This can be done by the following algorithm:

Algorithm 1 *Match*

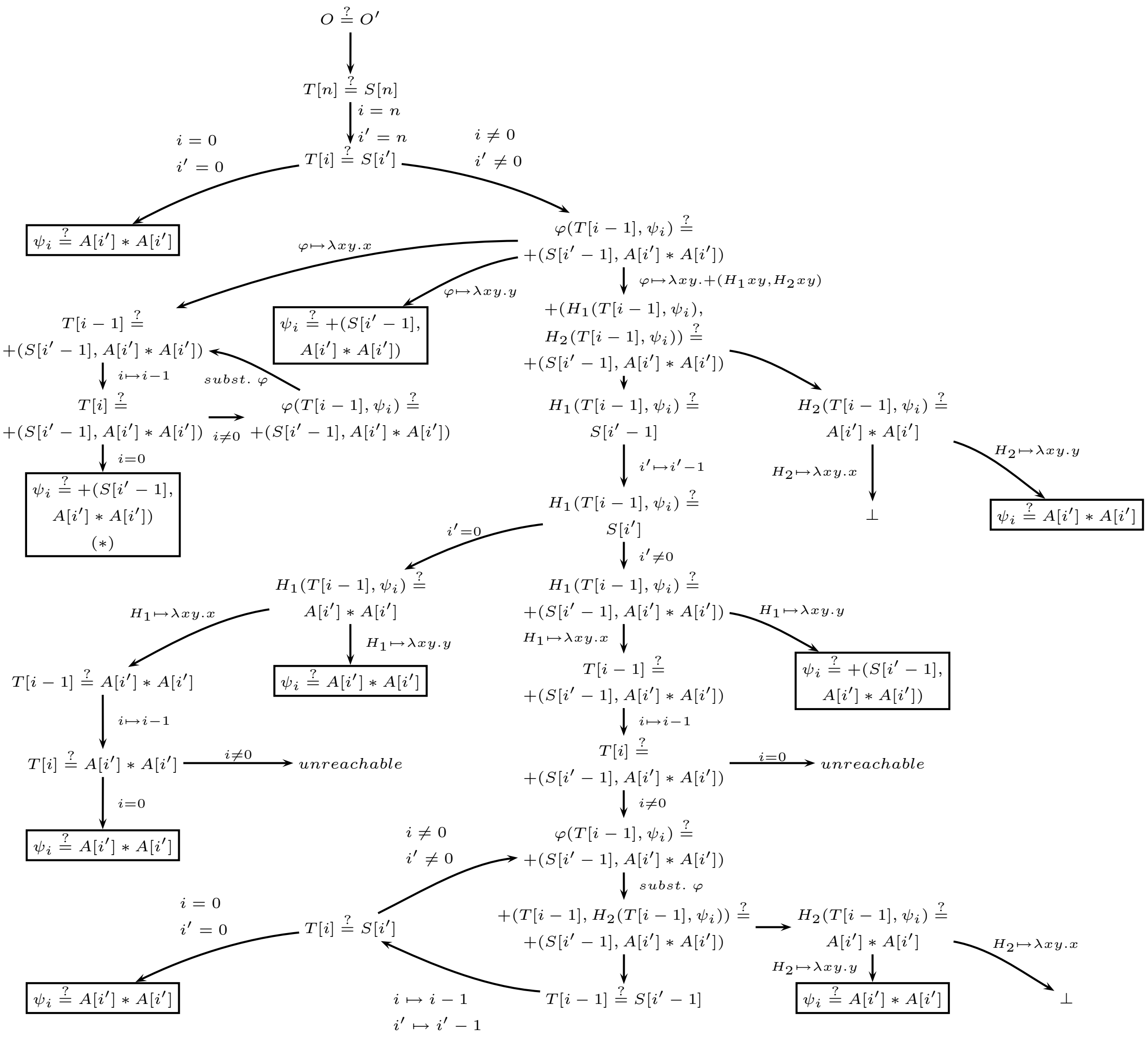
```
IN  : A SARE matching problem $S \overset{?}{=} S'$.
OUT : A set $\{(\sigma_1, c_1), \ldots, (\sigma_n, c_n)\}$ where $\sigma_i$ is an unifier of $S \overset{?}{=} S'$,
      and $c_i$ is a constraint on parameters for which $\sigma_i$ is valid.

(i)    Compute the MSA associate to $S =^? S'$ by the method describe above ;
(ii)   Compute reachability set $E$ of each node, then fix Input Variable and
       Input Success nodes ;
(iii)  For each Input Variable node : If $\exists(\vec{\imath}, \vec{\imath'}_1), (\vec{\imath}, \vec{\imath'}_2) \in E$ with $\vec{\imath'}_1 \neq \vec{\imath'}_2$ then
       replace node by $\perp$ ;
(iv)   Delete unaccessible nodes i.e. nodes whose reachability set is empty ;
(v)    Collapse cycles into one node ;
(vi)   Transform the obtained DAG into a tree by duplicating all nodes $x$ such
       that $\exists u, v, w$ nodes verifying $u \rightarrow^* v \rightarrow^* x$ and $u \rightarrow^* w \rightarrow^* x$, $v \neq w$. We
       have now a $\wedge,\vee,\sqcup$-tree, where each leaf is either $\top_{\{(\sigma,E)\}}$ or $\perp$ ;
(vii)  Compute the set of unifiers by recursively applying rules on $\wedge$, $\vee$ and $\sqcup$
       described in section 4, up to the root of the tree ;
(viii) If we obtain $\top_{Res}$ then return Res, else return $\emptyset$.
```

Step (ii) is correct because the reachability set gives us all possible values for $\vec{\imath}$ and $\vec{\imath'}$ in a state, corresponding to the values satisfying the context in the matching procedure. Step (iii) eliminates ambiguity in the Input Variable definition. Step (iv) corresponds to the application of the Empty rule. Step (v) can be applied, because reachability sets were already computed. One can notice that the MSA has the same transitions than the matching procedure, with the same context in firing relations. So it computes the same set of unifiers, which is correct (see correction proof in section 4). This justifies steps (vii) and (viii).

It may seem at first glance that the algorithm completely solves the matching problem. This is not the case, because the construction of the transitive closure of a relation is not an effective procedure [10]. So the algorithm works only when transitive closures are computable.

5.4 An Example

Let us apply our algorithm to the example of reduction presented in preliminaries. We obtain the following MSA :


$O \overset{?}{=} O'$
$T[n] \overset{?}{=} S[n]$
$i = n$
$i' = n$
$i = 0$
$i' = 0$
$T[i] \overset{?}{=} S[i']$
$i \neq 0$
$i' \neq 0$
$\psi_i \overset{?}{=} A[i'] * A[i']$
$\varphi(T[i-1], \psi_i) \overset{?}{=} +(S[i'-1], A[i'] * A[i'])$
$\varphi \mapsto \lambda xy.x$
$\varphi \mapsto \lambda xy.y$
$\varphi \mapsto \lambda xy.+(H_1 xy, H_2 xy)$
$T[i-1] \overset{?}{=} +(S[i'-1], A[i'] * A[i'])$
$\psi_i \overset{?}{=} +(S[i'-1], A[i'] * A[i'])$
$+(H_1(T[i-1], \psi_i), H_2(T[i-1], \psi_i)) \overset{?}{=} +(S[i'-1], A[i'] * A[i'])$
$i \mapsto i-1$
$subst.\ \varphi$
$T[i] \overset{?}{=} +(S[i'-1], A[i'] * A[i'])$
$i \neq 0$
$\varphi(T[i-1], \psi_i) \overset{?}{=} +(S[i'-1], A[i'] * A[i'])$
$H_1(T[i-1], \psi_i) \overset{?}{=} S[i'-1]$
$H_2(T[i-1], \psi_i) \overset{?}{=} A[i'] * A[i']$
$i=0$
$H_2 \mapsto \lambda xy.y$
$i' \mapsto i'-1$
$H_2 \mapsto \lambda xy.x$
$\psi_i \overset{?}{=} +(S[i'-1], A[i'] * A[i'])$
$(*)$
$H_1(T[i-1], \psi_i) \overset{?}{=} S[i']$
$\perp$
$\psi_i \overset{?}{=} A[i'] * A[i']$
$i'=0$
$i' \neq 0$
$H_1(T[i-1], \psi_i) \overset{?}{=} A[i'] * A[i']$
$H_1(T[i-1], \psi_i) \overset{?}{=} +(S[i'-1], A[i'] * A[i'])$
$H_1 \mapsto \lambda xy.x$
$H_1 \mapsto \lambda xy.y$
$H_1 \mapsto \lambda xy.y$
$H_1 \mapsto \lambda xy.x$
$T[i-1] \overset{?}{=} A[i'] * A[i']$
$\psi_i \overset{?}{=} A[i'] * A[i']$
$T[i-1] \overset{?}{=} +(S[i'-1], A[i'] * A[i'])$
$\psi_i \overset{?}{=} +(S[i'-1], A[i'] * A[i'])$
$i \mapsto i-1$
$i \mapsto i-1$
$T[i] \overset{?}{=} A[i'] * A[i']$
$i \neq 0$
$unreachable$
$T[i] \overset{?}{=} +(S[i'-1], A[i'] * A[i'])$
$i=0$
$unreachable$
$i=0$
$i \neq 0$
$\psi_i \overset{?}{=} A[i'] * A[i']$
$i \neq 0$
$i' \neq 0$
$\varphi(T[i-1], \psi_i) \overset{?}{=} +(S[i'-1], A[i'] * A[i'])$
$subst.\ \varphi$
$i = 0$
$i' = 0$
$T[i] \overset{?}{=} S[i']$
$+(T[i-1], H_2(T[i-1], \psi_i)) \overset{?}{=} +(S[i'-1], A[i'] * A[i'])$
$H_2(T[i-1], \psi_i) \overset{?}{=} A[i'] * A[i']$
$H_2 \mapsto \lambda xy.x$
$H_2 \mapsto \lambda xy.y$
$\psi_i \overset{?}{=} A[i'] * A[i']$
$i \mapsto i-1$
$i' \mapsto i'-1$
$T[i-1] \overset{?}{=} S[i'-1]$
$\psi_i \overset{?}{=} A[i'] * A[i']$
$\perp$

Final states are surrounded by rectangles. For sake of clarity, arcs are labeled by shortened notations. Starting from state $T[i] \overset{?}{=} S[i']$, label $i \neq 0$, $i' \neq 0$ represents relation $\{(i,i') \to (i,i') | i \neq 0, i' \neq 0\}$. Label $\varphi \mapsto \lambda xy.x$ indicates the addition of $\varphi \mapsto \lambda xy.x$ to the current substitution. The reaching set of the final state tagged with (*) is:

$$\left\{\begin{array}{c} .\to i\\ .\to i'\\ i=n\\ i'=n\end{array}\right\}\cdot\left\{\begin{array}{c} i\to i\\ i'\to i'\\ i\neq 0\\ i'\neq 0\end{array}\right\}\cdot\left(\left\{\begin{array}{c} i\to i-1\\ i'\to i'\end{array}\right\}\cdot\left\{\begin{array}{c} i\to i\\ i'\to i'\\ i\neq 0\end{array}\right\}\right)^*\cdot\left\{\begin{array}{c} i\to i-1\\ i'\to i'\end{array}\right\}\cdot\left\{\begin{array}{c} i\to i\\ i'\to i'\\ i=0\end{array}\right\}$$

Which boils down to: $\{. \to i, . \to i', i = 0, i' = n, n \geq 1\}$. And the substitution obtained is:

$$([\varphi \mapsto \lambda xy.x, \psi_0 \mapsto +(S[n-1], A[n] * A[n])], n \geq 1)$$

There remains to apply the same method to the other final states, and to combine unifiers by applying the rules of $\wedge$, $\vee$ and $\sqcup$. The final set of all possible solutions is:

$$\left\{\begin{array}{l}([\psi_0 \mapsto A[0] * A[0]], n = 0) \\ ([\varphi \mapsto \lambda xy.x, \psi_0 \mapsto S[n-1] + A[n] * A[n]], n \geq 1) \\ ([\varphi \mapsto \lambda xy.y, \psi_0 \mapsto S[n-1] + A[n] * A[n]], n \geq 1) \\ ([\varphi \mapsto \lambda xy.x + y, \psi_i = A[i] * A[i](0 \leq i \leq n)], n \geq 1)\end{array}\right\}$$

Each solution is defined by a substitution and the condition on the parameters for which it is valid. Note that only the last solution corresponds to a reduction, since in the others, φ is either not defined or not associative.

6 Conclusions

Algorithm templates represent programming models that convey genericity, portability, can be easily customized by the programmer to suit its need and at the same time have efficient implementations. Algorithm template recognition thus appears as a promising tool for code comprehension, validation and optimization. In this paper, we have presented a preliminary approach that provides such recognition for templates described by systems of affine recurrent equations. As a consequence, our analysis is able to recognize algorithms obtained by composition of other algorithms, since templates can be composed with other templates. While other analyses [18] could recognize an algorithm made of several known algorithms, ours works also for unknown algorithms.

In future work, we will investigate the feasibility of the approach on benchmark applications, with respect to the assumptions that have been made and by extending the existing prototype developed for the equivalence of SAREs. We would also like to address the recognition of templates parameterized by constructed types (such as matrices) so that the methods to be instantiated can be defined by the operations on the elementary types. Finally, the SARE templates have still some non variable definition domains and non variable dependence functions. Breaking these constraints would lead to possibly non-affine systems of recurrence equations and the applicability of our approach in this case need to be studied.

References

[1] Autebert, J.-M., J. Berstel and L. Boasson, *Context-free languages and push-down automata*, in: *Handbook of Formal Languages*, Springer Verlag, 1997 .

[2] Baader, F. and T. Nipkow, “Term Rewriting and all that,” Cambridge University Press, 1998.

[3] Barthou, D., P. Feautrier and X. Redon, *On the equivalence of two systems of affine recurrence equations*, Technical Report RR-4285, INRIA (2001).

[4] Barthou, D., P. Feautrier and X. Redon, *On the equivalence of two systems of affine recurrence equations*, in: *8th International Euro-Par Conference* (2002), p. 309.

[5] Darte, A., Y. Robert and F. Vivien, "Scheduling and automatic Parallelization," Birkhäuser, 2000.

[6] Feautrier, P., *Dataflow analysis of scalar and array references*, Int. J. of Parallel Programming **20** (1991), pp. 23–53.

[7] Feautrier, P., *Some efficient solutions to the affine scheduling problem, II, multidimensional time*, Int. J. of Parallel Programming **21** (1992), pp. 389–420.

[8] Hermann, M. and R. Galbavý, *Unification of infinite sets of terms schematized by primal grammars*, Theoretical Computer Science **176** (1997), pp. 111–158.

[9] Huet, G., *A unification algorithm for typed λ-calculus*, Theoretical Computer Science **1** (1975), pp. 27–57.

[10] Kelly, W., W. Pugh, E. Rosser and T. Shpeisman, *Transitive closure of infinite graphs and its applications*, Int. J. of Parallel Programming **24** (1996), pp. 579–598.

[11] Lee, L.-Q., J. G. Siek and A. Lumsdaine, *The generic graph component library*, in: *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1999, pp. 399–414.

[12] Metzger, R. and Z. Wen, "Automatic Algorithm Recognition: A New Approach to Program Optimization," MIT Press, 2000.

[13] Redon, X. and P. Feautrier, *Detection of scans in the polytope model*, Parallel Algorithms and Applications **15** (2000), pp. 229–263.

[14] Saouter, Y. and P. Quinton, *Computability of recurrence equations*, TCS **116** (1993), pp. 317–337.

[15] Shashidhar, K., M. Bruynooghe, F. Catthoor and G.Janssens, *Geometric model checking: An automatic verification technique for loop and data reuse transformations*, in: *International Workshop on Compilers Optimization Meets Compiler Verification*, ENTCS **65** (2002).

[16] Siek, J. G. and A. Lumsdaine, *The matrix template library: A generic programming approach to high performance numerical linear algebra*, in: *ISCOPE*, 1998, pp. 59–70.

[17] TayouDjameni, C., P. Quinton, S. Rajopadhye and T. Risset, *Derivation of systolic algorithm path problem by recurrence transformations*, in: *Parallel Computing*, 2000.

[18] Wills, L. M., *Using Attributed Flow Graph Parsing to Recognize Clichés in Programs*, in: *Proc. 5th Int. Workshop on Graph Grammars and their Application to Computer Science*, LNCS **1073** (1996), pp. 170–184.