Algorithm Recognition based on Demand-Driven Dataflow Analysis

Christophe Alias
PR*i*SM
U. of Versailles Saint-Quentin
Versailles, France
Christophe.Alias@prism.uvsq.fr

Denis Barthou
PR*i*SM
U. of Versailles Saint-Quentin
Versailles, France
Denis.Barthou@prism.uvsq.fr

Abstract

Algorithm recognition is an important problem in program analysis, optimization, and more particularly program comprehension. Basically, one would like to submit a piece of code, and get an answer like "Lines 10 to 23 are an implementation of Gauss-Jordan elimination". Existing approaches often perform a bottom-up recognition, using a database describing many implementation variations of a given algorithm, in a format making them difficult to maintain.

In this paper, we present a new method to recognize algorithm templates in a program. We perform a top-down recognition, stopping at template variables and thus limiting the need for many program variations. The code to analyze is given in SSA form, and the ϕ-functions are computed on-demand, when computation is possible, along the candidate slices.

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 comprehension and reverse engineering: we try to recognize in the program some codes that have a significant meaning.
- 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 part of the code by a completely different implementation, as for instance an iterative solver.
- 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 component) 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 [19] and its references. One issue of pattern matching is that it does not handle well the variations in the control or data flow of the program. Several complex pattern matching techniques have been proposed (see the recent book by Metzger and Wen [15] and its references) in order to tackle this issue. An alternative approach, explored by [3, 4] and [21] is to consider system of affine recurrences as the starting point of the algorithm recognition. 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 [22] for matrix manipulations, [12] for graph algorithms or [23] for the APP, to name a few.

The aim of this paper is to present a method for algorithm template recognition, based on the preliminary works described in [1, 4], enlarging the scope of the technique in [1] from *static control programs* [8] to any program. Given a program written in SSA form [6] and an algorithm template, our goal is to find out, if it exists, an instantiation of the template corresponding to the program. The ϕ-functions of the SSA form capture some dataflow information that may be needed for the matching to succeed. The on-demand evaluation of the ϕ-functions during the matching procedure provides a convenient way to combine the power of the previous analysis to a larger range of applications.

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 the conversion to a System of Recurrence Equations (SRE). Algorithm templates are given as static control programs, whereas the code to analyze is in SSA form. It has been shown that *static control programs* [8] can be automatically converted to Systems of Affine Recurrence Equations (SARE), and such a conversion already was the first step in [19]. As for codes in SSA form, we propose a way to encode it into an SRE. The next step is to design an equivalence test between SREs and SARE templates. This is the main theme of this paper.

Section 2 introduces some essential definitions about SREs and the transformation procedure from SSA form to SRE. Section 3 defines the problem at hand and Sections 4 and 5 build the semi-algorithm performing the match between the template and the code. Comparison with related works is performed in Section 7.

2 Background

We present the definition of systems of recurrence equations which are used in the rest of the paper, as well as how to transform a program in SSA form into an SRE.

The code in Figure 1.a provides a toy example for the notions presented in this section. This is a computation of the bit vector representing the local maxima in the input array `I`. Fig.1.b represents a template for the computation of a bit vector from an input array `Y`. `X` and `Y` are both template variables, `X` representing the operation computing the next bit.

```
MAX := I[0]
S := 0
do i := 1, n
  if (MAX < I[i]) then
     MAX := I[i]
     S := S+1
  endif
  S := S*2
enddo
OUTPUT := S
```

(a)

```
A := Y[0]
do i := 1, n
  A := X(A,Y[i])*2
enddo
OUTPUT = A
```

(b)

Figure 1. (a). Code to analyze (b). Template

2.1 Systems of Recurrence Equations

Systems of recurrence equations are a convenient way to represent algorithms: they already eliminate some syntactic aspects of the programs since they represent the computations with explicit dataflow information. Moreover, systems of affine recurrence equations, SAREs, can be obtained from static control programs by reaching definition analysis [8]. The basic reference on SAREs is [7]. Let us recall that a *static control program* is a program which manipulate arrays, iteration domains are bounded $\mathbb{Z}$-polyhedra and index functions of arrays are affine w.r.t. index variables. Figure 1.b gives an example of static control program.

Definition 1 *A* System of Recurrence Equations *is a set of equations called clauses, of the form:*

$$\forall i \in D_k : A[i] = f_k(\ldots B[u_{Bk}(i)]\ldots). \qquad (1)$$

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

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

The example of Fig.2 illustrates the transformation from the program of Fig.1.b to a SARE. The output, O_T is set to the

$$\begin{cases} i = 1: & A_2[i] = X(Y[0], Y[i]) * 2 \\ 2 \leq i \leq n: & A_2[i] = X(A_2[i-1], Y[i]) * 2 \\ & O_T = \quad A_2[n] \end{cases}$$

Figure 2. SARE template

last element of the recurrence in Fig.1.b, the input is the array Y and the variable A has been expanded into a one dimensional array A_2.

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

$$\forall i \in D_k : \theta(A, i) \geq \theta(B, u_{Bk}(i)) + 1$$

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 [20]. However, the existence of affine schedules for SAREs is decidable [9]. 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 can be a function variable and B an array variable. Both f_k and B can then be defined during the matching procedure. The SARE template representing the template of Fig.1.b is given in Fig.2.1. We assume that $n \geq 1$.

2.2 From SSA to SRE

Consider first a program in SSA form, where only scalars appear in the ϕ-functions. We assume in the following that lower and upper bounds of `do` loops are static (they depend on surrounding loop counters and on parameters). The procedure builds for each assignment in the program a clause of an SRE.

Algorithm 1 *SSA to SRE*
IN : A program in SSA form.
OUT : A SRE

1. *Give a different number to each ϕ-function*
2. *For each* `while` *loop, add a loop counter going from* 1 *to an upper bound denoted $u(i)$ where i is the iteration vector of the surrounding loops.*
3. *Perform scalar expansion. When a scalar,* `s`*, has an upward-exposed use in a loop, rewrite this use into* `s[prec(i)]` *where* `i` *is the iteration vector of the use statement.* `prec` *is a function providing the previous value of this vector, in the execution order.*
4. *Perform an if-convertion on the program.*
5. *For each assignment* `S[i] := t` *in the program, consider the intersection of the surrounding loop domains with the predicates of the if. This set may be defined by non-affine constraints in the loop counters and parameters. Let D denote the non-affine part of this set and L the affine part. We define the clause*

$$i \in L : S[i] = \mu_D(t)$$

*where μ_D is a new function symbol. Its semantics is $\mu_D(x) =$ **if** $i \in D$ **then** x **else** $\perp$ where $\perp$ denotes an undefined value.*

```
M0 := I[0]
S0 := 0
do i := 1, n
  S1 := φ(S0,S4)
  M1 := φ(M0,M3)
  if (M1 < I[i]) then
     M2 := I[i]
     S2 := S1+1
  endif
  S3 := φ(S1,S2)
  M3 := φ(M1,M2)
  S4 := S3*2
enddo
OUTPUT := φ(S0,S4[n])
```

Figure 3. SSA form of the code

The defining domains of the SRE built by this procedure are polyhedra, as in SAREs. However, the dependence functions may be non affine.

Fig.3 describes the SSA form of the code in Fig.1.a and Fig.4 shows its represention as a system of recurrence equations, as given by the previous algorithm.

The SSA form does not take into account the indices of the arrays. In order to transform a statement of the form `A3 := φ(A1,A2)` with `A1`, `A2` and `A3` arrays, into a clause of an SRE, instead of applying the previous steps, we compute the source of all elements $A_3[j]$, for all indices j, with an instance-wise dataflow analysis such as [8]. The code must then be a static control program and there is no more the advantages of an on-demand dataflow analysis.

$$\begin{cases} 1 \le i \le n : S_1[i] = & \phi_1(S_0, S_4[i-1]) \\ 1 \le i \le n : S_2[i] = & \mu_{M_1[i]<I[i]}(S_1[i]+1) \\ 1 \le i \le n : S_3[i] = & \phi_2(S_1[i], S_2[i]) \\ 1 \le i \le n : S_4[i] = & S_3[i]*2 \\ & M_0 = I[0] \\ 1 \le i \le n : M_1[i] = & \phi_3(M_0, M_3[i-1]) \\ 1 \le i \le n : M_2[i] = & \mu_{M_1[i]<I[i]}(I[i]) \\ 1 \le i \le n : M_3[i] = & \phi_4(M_1[i], M_2[i]) \\ & O_P = & \phi_5(S_0, S_4[n]) \end{cases}$$

Figure 4. SRE form of the code

3 Matching problem

We want to check whether a program P is an instance of a template T, where T is a static control program, and P is any program with scalar variables. A template is considered as a program with some redefinable functions and inputs, as templates in C++ operating on parametric types. For a program P in SRE form, $P\downarrow$ denotes the term obtained by completely unfolding the recurrence of P. This term is well-defined for a program P with a schedule (no infinite loop). Likewise, for a template T with a schedule, the term $T\downarrow$ exists. The matching problem is to find a substitution σ of the template variables such that $\sigma(T\downarrow)$ is syntactically equal to $P\downarrow$. Such a substitution, if it exists, is called a *unifier* of the matching problem between T and P. This problem depends clearly on the underlying algebra. 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 (no semantics). In particular ϕ-functions are not given any interpretation during the resolution of the matching problem.

The matching problem between two SAREs has been proved undecidable in [1], therefore the matching problem between a SARE template and a SRE, which is at least as complex, is also undecidable.

4 Principle of the algorithm

In order to give the intuition of the method, we study the case of the template and code in Fig.1.a when $n = 2$. The generalization will be described in the following section.

We can completely unfold the recurrence of the template:

$$O_T = X(X(Y[0], Y[1]) * 2, Y[2]) * 2$$

For the program, O_p is

$$\phi_5\left(0, \phi_2\left(\phi_1\left(0, S_3[1]*2\right), \mu\left(\phi_1\left(0, S_3[1]*2\right)+1\right)\right)*2\right)$$

with

$$S_3[1] = \phi_2\left(\phi_1\left(0, S_3[0]*2\right), \mu\left(\phi_1\left(0, S_3[0]*2\right)+1\right)\right)$$

Note that in the preceding expressions, $S_3[0]$ is not defined, its value will be denoted $\perp$.

The problem is to find out the possible values of X and Y such that the two expressions are the same, syntactically. Hence, the problem of matching is reduced to a semi-unification problem, between the terms of the template and of the program, where *unknowns* are template variables (here X and Y), and *closed variables* are program inputs (here $I[0]$, $I[1]$ and $I[2]$).

To unify two terms, we can apply Huet's procedure [10]. Basically, we try to decompose the problem by applying the rule: $f(\vec{t}) \stackrel{?}{=} f(\vec{t'}) \rightarrow t_1 \stackrel{?}{=} t'_1 \wedge \ldots \wedge t_n \stackrel{?}{=} t'_n$ until terms of the form $X(\vec{t}) \stackrel{?}{=} f(\vec{t'})$ are obtained, where X is a template variable. Then we try to construct an unifier by trying:

- Projections: $X(\ldots x_k \ldots) = x_k$;
- Imitation: $X(\vec{x}) = f(X_1\vec{x} \ldots X_n\vec{x})$, with the X_i new function variables.

When decomposition is not possible because the two head operators are different, then this is a failure, the program and template are not equivalent. When one of these operators is ϕ function then this function may be computed in order to avoid the failure of the method. If one of the variable in the ϕ-function is never in dependence with the variable defined by the ϕ-function, then a dependence analysis can prove it. Otherwise, more complex dataflow analyses are necessary [8, 17, 14]. The ϕ-function cannot be evaluated if the code containing it is outside the scope of the dataflow analysis. In this case, the ϕ-function remains unevaluated and the unification fails, the template is then said non-equivalent to the program. This is a safe approximation. Likewise, μ-functions could be evaluated if there were a dataflow analysis able to handle in an exact way non-affine constraints.

On the example, we try to decompose the head operator in both expressions: for the code, this is ϕ_5, for the template this is $*$. Thus ϕ_5 is computed: as we have supposed that $n \ge 0$, a dependence analysis can prove that O_P does not depend on its first argument, $\phi_5(x, y) = y$. This is equivalent to prove that at least one iteration of the loop is executed. The head operator is $*$ now in both terms. By decomposition, we have to match $X(X(\ldots), Y[2]) \stackrel{?}{=} \phi_2(\phi_1(\ldots), \mu(\ldots))$. By applying the rule imitate, we define $X(x, y)$ as $\phi_2(X_1(x, y), X_2(x, y))$ where X_1 and X_2 are new template variables and the next step is a decomposition for ϕ_2.

Proceeding in the same manner, we find out the following solution:

$$X(x, y) = \phi_2(\phi_1(0, x*2), \mu(\phi_1(0, x*2))+1)$$

and $Y[0] = 0$. This is one of the possible solutions which can be obtained by an exhaustive application of the proce-

dure. This solution is not very helpful as it is. The semantics of the ϕ functions is:

$$\phi_1(x,y) = \textbf{ if } i > 0 \textbf{ then } y \textbf{ else } x$$

$$\phi_2(x,y) = \textbf{ if } M_1[i] > I[i] \textbf{ then } y \textbf{ else } x.$$

Therefore, the function X of the template of Fig.1.b is, in a more natural way:

$$X(x,y) = \left|\begin{array}{ll} \textbf{if } M_1[i] < I[i] & \\ \quad \left|\begin{array}{ll} \textbf{if } M_1[i] < I[i] & \\ \textbf{then} & \left|\begin{array}{l} \textbf{if } i > 0 \\ \textbf{then } x * 2 \\ \textbf{else } 0 \end{array}\right. \\ \textbf{else } \bot & \end{array}\right. & \\ \textbf{then} & \\ \textbf{else} & \left|\begin{array}{l} \textbf{if } i > 0 \\ \textbf{then } x * 2 \\ \textbf{else } 0 \end{array}\right. \end{array}\right. + 1$$

Computation of M_1, used by X, can be added likewise.

In fact, as templates and programs can depend on a parameter (here n), computed terms can have a parametric length, making Huet's algorithm not applicable as it is. Let us try nevertheless to build a unification tree. We start with state $O_T \stackrel{?}{=} O_P$, then we unfold the recurrence, applying on demand Huet's rules. We then obtain an unification tree – with a parametric number of states – which gives us the solution. This tree as many similar nodes, and as in [4], the idea is to capture this unification tree into an automaton with a finite number of states and then analyze it to construct the final set of solutions.

5 Our algorithm

As said above, the idea of this semi-algorithm is to implement the procedure with an automaton and to 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 Automaton

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 in 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 [2] 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 $T \stackrel{?}{=} P$ a matching problem. We explain in this section how to build the corresponding MSA, by first describing the general form of a state, and then the different kind of transitions between two states. Next section gives an algorithm to analyze the matching MSA in order to compute the set of unifiers.

5.2.1 States

Each state of our matching MSA has two part: a clause $\sigma : t \stackrel{?}{=} t'$ with σ a substitution, and a vector of integers v_p, which is the concatenation of $\vec{i}$ and $\vec{i'}$, where $\vec{i}$ is the lhs iteration vector, and $\vec{i'}$ is the rhs iteration vector. The *initial state* is $Id : O_T[\vec{i}] \stackrel{?}{=} O_P[\vec{i'}]$, where O_T and O_P are corresponding outputs of T and P. Its reaching set is $\{\langle \vec{i}, \vec{i'} \rangle | \vec{i} = \vec{i'}\}$. The *final states* are either:

- $Y[\vec{i}] \stackrel{?}{=} t'$ where Y is an input variable of the template T;
- $\bot$ if there is no unifier.

5.2.2 Transitions

We describe thereafter the transitions:

- Decompose: From a state with label:

$$\sigma : f(\vec{t}(\vec{i})) \stackrel{?}{=} f'(\vec{t'}(\vec{i'}))$$

starts a transition to each state:

$$\sigma : t_k(\vec{i}) \stackrel{?}{=} t'_k(\vec{i'})$$

with the firing relation:

$$Id = \{\vec{i} \to \vec{i},\ \vec{i'} \to \vec{i'}\}$$

provided $f = f'$. All these transitions constitute an *and*-branching. If $f \neq f'$ then start a transition to a failure node $\perp$.

- Compute ϕ: Consider a state with label:

$$\sigma : f(\vec{t}(\vec{i})) \stackrel{?}{=} \phi(\vec{t'}(\vec{i'}))$$

If ϕ is not computable then start a transition to $\perp$. Otherwise compute its value (for instance with the methods in [8, 14], which is of the form: $\phi = \lambda\vec{x}.x_k\ \forall\vec{i'} \in D_k$ for some domain D_k. Create a transition to each state:

$$\sigma : f(\vec{t}(\vec{i})) \stackrel{?}{=} t'_k$$

with the firing relation:

$$\{\vec{i} \to \vec{i},\ \vec{i'} \to \vec{i'},\ \vec{i'} \in D_k\}$$

All these transitions constitute an and-branching.

- Generalize: From a state with label:

$$\sigma : X[u(\vec{i})] \stackrel{?}{=} t'$$

create a transition to state:

$$\sigma : X[\vec{i}] \stackrel{?}{=} t'$$

with the firing relation:

$$\{\vec{i} \to u(\vec{i}),\ \vec{i'} \to \vec{i'}\}$$

There is a similar rule for the rhs. This rule is important because it normalizes the form of the states, ensuring a finite number of states [1].

- Compute: If $X[\vec{i}] = t_k\ \forall\vec{i} \in D_k$, then from a state with label:

$$\sigma : X[\vec{i}] \stackrel{?}{=} t'$$

create a transition to each state:

$$\sigma : t_k(\vec{i}) \stackrel{?}{=} t'$$

with the firing relation:

$$\{\vec{i} \to \vec{i},\ \vec{i'} \to \vec{i'},\ \vec{i} \in D_k\}$$

All these transitions constitute an and-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.

- Substitute: Simply replace an already defined template variable by its value. Firing relation is Id.

Rule Compute ϕ does not change the following result proved in [1]:

Proposition 1 *Let $T \stackrel{?}{=} P$ be a matching problem, and A be its corresponding MSA. Then the number of states of A is finite.*

One can remark that the ϕ-functions are only computed when they are needed by the Decompose rule. Therefore, this limitates the cost due to dataflow analysis. Moreover, a ϕ-function can be imitated as it is. Hence, it allows us to handle programs with non-computable ϕ-functions.

5.3 Analysis of the matching MSA

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

Algorithm 2 *Match_SRE*

IN : *A matching problem $T \stackrel{?}{=} P$.*
OUT : *A set of unifiers $\{Sol_1 \ldots Sol_n\}$*

1. *Compute the MSA associate to $T \stackrel{?}{=} P$ by the method described above;*
2. *Compute the reaching set of each node. Eliminate nodes with an empty reaching set ;*
3. *Propagate $\perp$ by applying rules $\perp \wedge x = x \wedge \perp = \perp$ and $\perp \vee x = x \vee \perp = x$;*
4. *Repeat:*
   - *For each or-branching, enable one arc;*
   - *$Sol \leftarrow \emptyset$;*
   - *For each definition $[X \mapsto t]$ enabled:*
     - *Initialize the set E with the reaching set of the definition node;*
     - *Visit the following nodes, then add to E the reaching sets of the nodes which precede the substitution of X by t.*
     - *add $[X \mapsto t]\ \forall(\vec{i}, \vec{i'}) \in E$ to Sol*
   - *If there is no functional incoherence in Sol Then emit(Sol);*

Until they is no more or-branching to enable.

Step 2 allows us to eliminate unaccessible nodes due to systematic application of Compute and Compute ϕ rules. Step 3 avoids unnecessary work by eliminating the nodes which do not produce unifiers. Step 4 studies all possible combinations of unifiers in the remaining automaton. For each definition of a ϕ-function it collects all couples $(\vec{i}, \vec{i'})$ to which the definition is applied. The final solution is discarded if a variable X have two distinct definitions for a value of $\vec{i}$ (functional incoherence).

Rule Compute ϕ does not change the soundness result proved in [1]. However, our algorithm is unable to find functions with a recursive definition, so it is not complete. Another restriction is the construction of the transitive closure of a relation which occurs when a reaching set is computed. This is not an effective procedure [11] and the algorithm works only when transitive closures are computable.

6 An example

Let us apply our algorithm to the matching problem between the template of Figure 1.b and the program of Figure 1.a. Applying on demand the rules described in Section 5.2, we obtain an MSA of 72 states. For presentation reasons, we will just describe a small, but significant part of the MSA. Figure 5 give the first part of the MSA. For sake of clarity, arcs are labeled by shortened notations. Starting from state $A[n] \stackrel{?}{=} \phi_5(S_0, S_4[n])$, label $i = n$ indicates a generalization represented by the relation $\{(.,.) \rightarrow (i,.) | i = n\}$. $A[i]$ is computed with respect to its definition in the SARE template. We will only consider $i \geq 2$, which represents relation $\{(i,i') \rightarrow (i,i') | i \geq 2\}$. Because $X(A[i-1], Y[i]) * 2 \stackrel{?}{=} \phi_5(S_0, S_4[n])$ is a $rigid$-$rigid$ pair (the head symbol of lhs is the unary function $x \mapsto x * 2$), ϕ_5 must be computed. This explains the two next transitions. $S_4[n]$ is generalized in $S_4[i']$, then computed, and gives $S_3[i'] * 2$. The two terms are decomposed, then $S_3[i']$ is computed and gives the $flexible$-$rigid$ pair $X(A[i-1], Y[i]) \stackrel{?}{=} \phi_2(S_1[i'], S_2[i'])$. We can now try to build a value for X by applying Huet's rules. Figure 6 gives the part of the MSA obtained while trying $X \mapsto \lambda xy.x$. Since ϕ_2 is not computable, we obtain failures states ($\perp$) which are final states. While trying $X \mapsto \lambda xy.y$, we obtain $Y[i] \stackrel{?}{=} S_3[i']$, which is a final state. To achieve the construction of the MSA, we should try imitation of rhs head symbol (here ϕ_2) by trying $X \mapsto \lambda xy.\phi_2(H_1(x,y), H_2(x,y))$, where H_1 and H_2 are two new template variables. Since the corresponding part of MSA is too big, we will not show it.

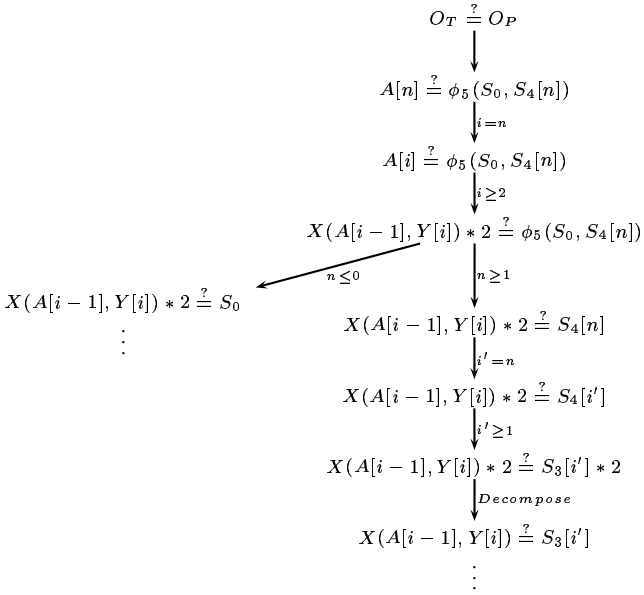

Figure 5. First part of the example MSA

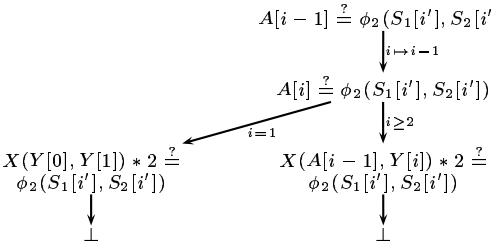

Figure 6. Part of MSA obtained while trying $X \mapsto \lambda xy.x$

Let us analyze the MSA. By propagating $\perp$ up to the first or-branching, we discard the projection $X \mapsto \lambda xy.x$. Assume the arc $X \mapsto \lambda xy.y$ is enabled. The reaching set of the definition node $X(A[i-1], Y[i]) \stackrel{?}{=} S_3[i']$ is $\{(n,n)\}$ so the following substitution is added to Sol:

$$[X \mapsto \lambda xy.y] \; \forall (i,i') \in \{(n,n)\}$$

In the same way, the reaching set of the final state $Y[i] \stackrel{?}{=} S_3[i']$ is $\{(n,n)\}$ so the following substitution is added to Sol:

$$[Y[i] \mapsto S_3[i']] \; \forall (i,i') \in \{(n,n)\}$$

Finally, the following solution is emitted:

$$\begin{array}{ll} [X \mapsto \lambda xy.y] & \forall (i,i') \in \{(n,n)\} \\ [Y[i] \mapsto S_3[i']] & \forall (i,i') \in \{(n,n)\} \end{array}$$

The solution emitted when the imitation arc is enabled is:

$$\begin{array}{ll} [X \mapsto \lambda xy.\phi_2(x*2, \mu(x*2)+1)] & \forall (i,i') \in \{(1,1) \ldots (n,n)\} \\ [Y[i-1] \mapsto 0] & \forall (i,i') \in \{(1,1)\} \end{array}$$

Which corresponds to the solution given in section 4.

7 Related works

It is a well-known fact that programmers spend most of their time to understand programs. Hence, tools which facilitate program comprehension are important. They can be divided in two complementary categories:

- Tools which *help* the user to understand a program ;
- Tools which try to *understand automatically* parts of a program by performing subcomponent extraction.

The first category includes *software visualization tools* (see for example [24] or [18]) which allows the user to understand software architecture by providing a graphical presentation of modules dependencies and searching facilities. Pintzger's revealer [16] extracts parts of the source code which corresponds to a given pattern by performing a lexical analysis. The user can then quickly find parts of the code containing characteristic statements, like for example the declaration of a Java server socket.

We are interested in automatic comprehension of program. But what does it really mean to "understand a program"? Biggerstaff et al. [5] defined the *concept assignment problem* as "*[...] a process of recognizing concepts within a computer program and building up an 'understanding' of the program by relating recognized concepts to portions of the program, its operational context and to one another.*". Since semantic equivalence is not decidable, the concept assignment problem which is at least as difficult, is not decidable. There exists several approaches to recognize a concept in a program. We will evaluate these with respect to the following criteria:

- *Cost* ;
- *Maintainability of pattern base*. Can we easily add our own patterns ?
- *Program variations handled*.

The last one is the most important. Indeed, the main difficulties come from various codes implementing the same algorithm. The different kinds of variation that can be found in a program are mentioned in [15]. To sum them up, they are:

- *Organization variation:* any permutation of independent statements and introduction of temporary variables ;
- *Data structure variation:* the same computation with a different data structure ;
- *Control variation:* any loop transformation, *e.g.* fusion, splitting, skewing, ... or control transformation *e.g.* if-conversion or dead-code suppression;
- *Semantic variation:* pieces of code equivalent modulo a theory defining properties on operators, *e.g.* associativity/commutativity of $+$.

Wills [25] represents programs by a particular kind of dependence graph called *flow-graphs*, and patterns by flow-graph *grammar rules*. The recognition is performed by parsing the program's graph according to the grammar rules. We finally obtain a *parsing tree* which represents a hierarchical description of a plausible project of the program. This approach is a pure bottom-up *code-driven* analysis based on exact graph matching. Wills has shown that the flow-graph recognition is NP-complete, and argues that even if the cost of her algorithm is exponential in the worst case, it is feasible to apply it to practical partial program recognition. Patterns are represented by grammars rules, encoding a hierarchy among them, but making the pattern base difficult to maintain. Organization variation is partially supported. Temporary variables can be handled by adding specific rules. All others variations can be handled only if they are explicitly described in the pattern base.

Di Martino et al. [13] propose an approach similar to Wills'. Flow graphs and patterns are coded by `prolog` clauses, then the recognition is performed by SLD-resolution. The differences are essentially in the abstract program representation (flow graph).

Metzger [15] normalizes the program's AST by applying various heuristics, and then compares it to the pattern AST. Of course this approach is scalable and low cost. Moreover, the patterns are given in the source form. This cannot handle much program variations though, in particular data structure variations. Organization variations are not handled by the algorithm itself, but by pre-treatments applied to the program. In the same way, the control variations supported are bounded to pre-treatments. One can remark that semantic variations are partly taken into account, with commutativity of usual operators.

Our approach is able to handle all variations described, except for semantic variation. Organization and data structure variations are normalized during the conversion to the SRE. Since our algorithm checks on-demand the order in which operations appear in the template and the program, it copes with any transformation preserving the order of the operations, in particular the control variations. Templates are given as source code, so new patterns can be added to the base without difficulties. However, our algorithm is expensive because of the reaching set computations and dataflow analyses. The evaluation of different approaches is summarized in Figure 7.

Criteria	Wills	DiMartino	Metzger	Us
Cost	high	high	**low**	high
Adding patterns is...	diffi cult	diffi cult	**easy**	**easy**
Organization variations	**yes**	**yes**	**yes**	**yes**
Data Structure variations	no	no	no	**yes**
Control variations	no	no	**partly**	**yes**
Semantics variations	no	no	**partly**	no

Figure 7. Evaluation of different approaches

8 Conclusion and Future work

Algorithm templates represent programming models that convey genericity, portability, that 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 an approach that provides such recognition for templates described by systems of affine recurrent equations that can be applied on any code. 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 [25] could recognize an algorithm made of several known algorithms, ours works also for unknown algorithms.

The method presented relies on a precise knowledge of the dataflow in the program. The on-demand dataflow analysis approach reduces the piece of code that must be in the scope of an exact dataflow analysis to the portion only needed by the template. Non static control programs can thus be analyzed with this method, provided the difficult parts of the program correspond to the template variables.

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. In order to make algorithm template recognition more efficient, several further developpements can be explored: faster heuristics, use of slicing techniques in order to target only good candidates for template recognition, and recognition of templates organized in a hierachical way: the template described in this paper is an instance of a template of reduction. This could drastically reduce the cost of the recognition with respect to the number of templates considered.

References

[1] C. Alias and D. Barthou. On the recognition of algorithm templates. In *International Workshop on Compilers Optimization Meets Compiler Verifi cation*, volume 82 of *ENTCS*. Elsevier Science, 2003.

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

[3] D. Barthou, P. Feautrier, and X. Redon. On the equivalence of two systems of affi ne recurrence equations. Technical Report RR-4285, INRIA, Oct. 2001.

[4] D. Barthou, P. Feautrier, and X. Redon. On the equivalence of two systems of affi ne recurrence equations. In *8th International Euro-Par Conference*, page 309. Springer, LNCS 2400, 2002.

[5] T. J. Biggerstaff, B. Mitbander, and D. Webster. The concept assignment problem in program understanding. In I. C. S. Press, editor, *15th International Conference on Software Engineering*, May 1993.

[6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Effi ciently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

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

[8] P. Feautrier. Datafbw analysis of scalar and array references. *Int. J. of Parallel Programming*, 20(1):23–53, Feb. 1991.

[9] P. Feautrier. Some effi cient solutions to the affi ne scheduling problem, II, multidimensional time. *Int. J. of Parallel Programming*, 21(6):389–420, Dec. 1992.

[10] G. Huet. A unifi cation algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[11] W. Kelly, W. Pugh, E. Rosser, and T. Shpeisman. Transitive closure of infi nite graphs and its applications. *Int. J. of Parallel Programming*, 24(6):579–598, 1996.

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

[13] B. D. Martino and G. Iannello. PAP recognizer: A tool for automatic recognition of parallelizable patterns. In *4th International Workshop on Program Comprehension (IWPC)*, pages 164–174. IEEE Computer Society Press, 1996.

[14] D. Maydan, S. Amarasinghe, and M. Lam. Array datafbw analysis and its use in array privatization. In *ACM Symp. on Principles of Programming Languages*, pages 2–15, Charleston, SC, Jan. 1993.

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

[16] M. Pinzger, M. Fischer, H. Gall, and M. Jazayeri. Revealer: A lexical pattern matcher for architecture recovery. In *9th Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society Press, 2002.

[17] 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.

[18] A. J. Quigley. Experience with FADE for the visualization and abstraction of software views. In *10th International Workshop on Program Comprehension (IWPC)*. IEEE Computer Society Press, 2002.

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

[20] Y. Saouter and P. Quinton. Computability of recurrence equations. *TCS*, 116(2):317–337, 1993.

[21] K. C. Shashidhar, M. Bruynooghe, F. Catthoor, and G. Janssens. Geometric model checking: An automatic verifi cation technique for loop and data reuse transformations. In *International Workshop on Compilers Optimization Meets Compiler Verifi cation*, volume 65 of *ENTCS*. Elsevier Science, 2002.

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

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

[24] A. Telea, A. Maccari, and C. Riva. An open visualization toolkit for reverse architecting. In *10th International Workshop on Program Comprehension (IWPC)*. IEEE Computer Society Press, 2002.

[25] L. M. Wills. *Automated Program Recognition by Graph Parsing*. PhD thesis, MIT, July 1992.