Instancewise Array Dependence Test for Recursive Programs

Pierre Amiranoff, Albert Cohen, Paul Feautrier

Instancewise Array Dependence Test for Recursive Programs

Pierre Amiranoff [*][†] Albert Cohen [*] Paul Feautrier [*][‡]

[*] A3 group, INRIA Rocquencourt
[†] CEDRIC laboratory, CNAM Paris
[‡] LIP, ÉNS Lyon

Abstract

Starting from a generalization of induction variables, we present a dependence test framework for recursive programs. For a restricted class of programs, a statically computable function maps every run-time instance of a program statement to the data-structure elements it accesses. Statement instances and data structure layouts are described through formal language tools.

This framework is applied to the automatic detection of data and control parallelism. We extend the concept of instancewise data dependences to recursive programs and we further investigate a dependence test for recursive programs operating on arrays. This test is interpreted as a flow problem in a directed graph with weighted edges; and the problem is shown to be NP-complete. We provide an exponential but efficient algorithm based on integer linear programming and explore simple examples.

1. Introduction

Our work aims at the compile-time computation of array access locations along any program execution, possibly recursive. We will derive from this computation a method devoted to precise dependence test and aggressive parallelization of recursive programs.

This paper focuses on the case of arrays, but our method and implementation extend to all "monoid-addressable" data structures, e.g., lists, trees, combined arrays and trees, etc. Its basis is the theory of regular languages and finite state transducers. Static properties are evaluated at the level of each run-time statement instance, and expressed by the means of automata and transducers.

Research in the field of automatic parallelization of recursive programs is still at its early stages. Compared to our previous work [7, 4, 3], a much more efficient technique is presented, implemented and experimented on several realistic examples. From the latter technique, we present the first instancewise dependence test for a class of recursive programs operating on multi-dimensional arrays. Most related works address the analysis of recursive programs with pointers, but [12, 8] also deal with pointer arithmetic (and thus, arrays). They rely on more classical data-flow frameworks which blur the distinction between the multiple instances of a statement. Therefore, our instancewise technique yields much less false dependences, at the cost of a more constrained program model.

2. The Control Domain Model

Our basic principle is that the code operating on array locations is purely functional: variables appearing in array indices do not have any side-effects, i.e., they are fully characterized by the sequence of variable definition statements followed along an execution. Thus, instead of studying the trace of all statements along an execution of the program, we may limit the grain of the analysis to a sub-trace of the "meaningful statements". This model turns out to be more tractable than static analysis frameworks focusing on full execution traces.

As a general rule, static analysis brings only conservative results concerning control flow and variable values. Thus, we will not take conditional tests and loop bounds into account, and consider that *both* `then` and `else` branches *can* be taken. However, thanks to the instancewise analysis framework, we do not confuse the run-time properties of two exclusive branches. In other words, we avoid the pitfall of classical data-flow analyses which do not preserve the context of static properties.

To implement this framework, we have built up a special language named MOGUL, for MOnoid GUided Language. According to our objective, the MOGUL syntax concentrates on variables that carry the values of array indices, named *induction variables* (generalized from nested loops), ignoring all other variables. Thus, the principal constraint will concern operations over these induction variables.

```
let der = ... (* arrays'length - 1 *)
let ta = ref ...
let tb = ref ...;;

(* copy ta from x to r into tb at y *)

let rec copya y x r =
  if x <= r
  then begin
      List.nth !tb y :=
      !(List.nth !ta x);
      copya (y+1) (x+1) r
    end;;

let rec copyb y x r = ...;;

(* merge the sub-arrays ta[p..q-1]
 * and ta[q..der], both being sorted,
 * into tb with r = current index of tb *)

let rec merge_ab p q r =
  if p < q && q <= der
  then begin
      let elp = !(List.nth !ta p)
      and elq = !(List.nth !ta q)
      in
        if elp < elq
        then begin
            List.nth !tb r := elp;
            merge_ab (p+1) q (r+1)
          end
        else begin
            List.nth !tb r := elq;
            merge_ab p (q+1) (r+1)
          end
    end
  else begin
      if p < q
      then
        copya r p (q-1)
      else
        copya r q der
    end;;

let rec merge_ba p q r = ...;;

let rec sort_ab p q r =
  if p < r
  then begin
    if q <= (p+r)/2 (* dichotomy *)
    then
      sort_ab p (q+1) r
    else begin
      sort_ba p p (q-1);
      sort_ba q q r;
      merge_ab p q p
    end
  end

and sort_ba p q r = ...;;

copya 0 0 der;
sort_ab 0 0 der;;
```

```
    monoid Monoid_int [ 1, -1 | congr ];
    structure Monoid_int A;
    structure Monoid_int B;



    function Copya (Monoid_int X, Monoid_int Y) {
      if test {
        B[X] = A[Y];
        Copya (X.1, Y.1);
      }
    }

    function Copyb (Monoid_int X, Monoid_int Y) { ... }




    function Merge_ab (Monoid_int P, Monoid_int Q, Monoid_int R) {
      if test
        if test {
          B[R] = A[P];
          Merge_ab (P.1, Q, R.1);
        }
        else {
          B[R] = A[Q];
          Merge_ab (P, Q.1, R.1);
        }
      else
        if test
          Copya (R,P);
        else
          Copya (R,Q);
    }

    function Merge_ba (Monoid_int P, Monoid_int Q, Monoid_int R) { ... }

    function Sort_ab (Monoid_int P, Monoid_int Q, Monoid_int R) {
d     if test
f       if test
B         Sort_ab (P, Q.1, R);
e       else {
C         Sort_ba (P, P, Q.-1);
D         Sort_ba (Q, Q, R);
E         Merge_ab (P, Q, P);
       }
    }

    function Sort_ba (Monoid_int P, Monoid_int Q, Monoid_int R) { ... }

    function Main () {
A     Sort_ab (@, @, Der);
    }
```

Figure 1. Program To_&_fro

2.1. Presentation of the Running Examples

We took our choice of two running examples to clearly expose our framework. We give both the OCaml and MOGUL versions, labeling statements if necessary.

Figure 1 presents the To_&_fro example, a version of the recursive merge-sort algorithm, working alternatively over two arrays during the fusion phase, the first phase implementing a classical divide-and-conquer dichotomy. The recursive function Merge_ab (respectively Merge_ba) merges and sorts two halves of array A into array B (respectively of array B into array A). Parameters P and Q indicate the current elements of the sub-arrays, the third parameter indicates the next place to write into the sorted array. The

```
let ta = Array.make 6 1;;


let rec pascaline l =
  if l < 6
  then
    let x = ref ta.(0)
    and y = ref 0
    in
      begin
        for i = 1 to (l-1) do
          y := ta.(i);
          ta.(i) <- !x + !y;
          x := !y;
        done;
        pascaline (l+1)
      end;;

pascaline 0;;
```

```
monoid Monoid_int [1, -1 | congr];
structure Monoid_int A;

function Pascaline (Monoid_int L) {
  if test {
    for (Monoid_int I = 1; test; .1) {
       joker = A[I];
       A[I] = joker;
       joker = joker;
    }
    Pascaline (L.1);
  }
}




function Main () {
   Pascaline (@);
}
```

Figure 2. Program `Pascaline`

bound for the last element does not appear since it is used only in test conditions. The dichotomy requires a recursive incrementation of the index `Q`: this is dictated by the program model, as we will see below. The same artifice is in use for `Sort_ab` and `Sort_ba` functions.

Figure 2 shows the `Pascaline` example, a program to evaluate the binomial coefficients (a line of Pascal's triangle). It exhibits both a loop statement and a recursive call.

2.2. Interprocedural Control Flow Graph

We represent the execution of recursive programs with control flow graphs enriched with a stack. Its role is to track the succession of procedure calls and returns.

Figure 3 presents the control flow graph for `To_&_fro`, simplified for readability (we omitted the body of functions `Merge_ab` and `Merge_ba`).

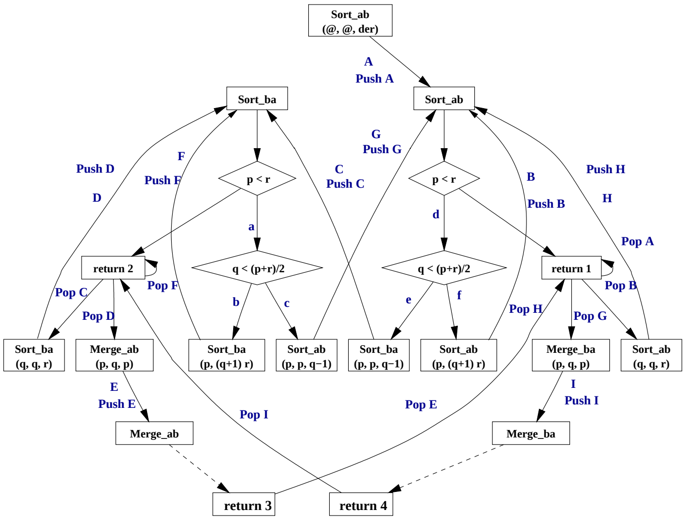


Figure 3. Control flow graph of `To_&_fro`

Different Kinds of Statements. MOGUL statements are split into terminal and compound statements. Notice that procedure returns figure among terminal statements; they are implicitly present in the source program in the form of body closing brackets. Compound statements are the conditional, `for` loop, procedure call, and sequence of several statements. We will see below that execution of any of these compound statements begins by defining *induction variables*.

Execution Trace. Naming each statement with a distinct label allows unambiguous denotation of any execution of the program with an execution trace. From the context-free grammar given by the MOGUL syntax, we may derive for any MOGUL program a corresponding context-free trace grammar, and the *interprocedural control flow graph* can be viewed as the graph of a pushdown automaton accepting the trace language.

A *trace prefix* denotes the part of a complete trace relative to an unachieved execution. An *instance* is a particular execution of a statement during a given execution of the program, it is associated with a unique *trace prefix*.

2.3. Control Words

The basic principle is as following: a partial or complete execution trace is structured as a nest of "blocks", each terminal or compound statement being such a block. Executing any statement (even a terminal one) will be interpreted as calling a function with dynamic stack allocation followed by deallocation when completed. In the special case of loops, each iteration is nested in the previous one and all are closed back at the loop completion.

Following this principle, we will consider the *extended stack* which carries the sequence of all labels representative of those statements that have been "opened" since the beginning of execution and not "closed" yet. The word stored in the extended stack is called *control word*. Combining this extended stack with the interprocedural control flow graph yields a pushdown automaton accepting the trace language. The stack language of a pushdown automaton is a rational language [11], hence the control words language is rational.

Example. Figure 4 shows a complete *trace*. Above the simplified version, we present a full version, i.e., enriched with the indication of statement closures (overlined). Below this trace, several associated control words are presented. We may deduce them from the full version trace, by considering that a pair of side by side corresponding labels, the first simple and the other overlined, annihilate each other from control words, according to the Dick languages reduction.

Control Automaton. We show on the running example how to construct a finite state automaton that recognizes the control words language.

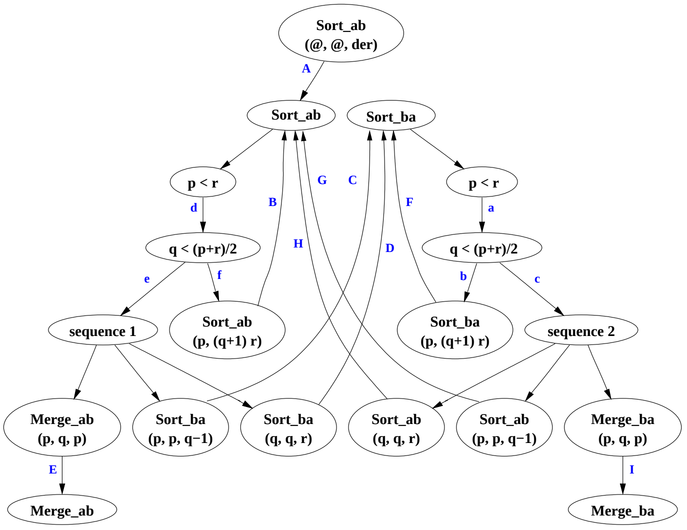


Figure 5. Control Automaton of `To_&_fro`

The control automaton is derived from the control flow graph: in the latter, a sequence of successive statements takes form as successive edges, while in the control automaton each of these statements is linked by an edge from the common enclosing statement, see Figure 5. All states are final. Since return nodes are useful only for execution traces but out of interest for dependence analysis, we will simply ignore them. Note that control automaton makes no distinction between the sequence (`;`) and the conditional.

2.4. Instances

Among all possible executions of the program (according to input data and test results), it is easy to see that a given control word may be the abstraction of many trace prefixes, possibly an infinite number. As an example, consider a trace prefix containing the trace of a conditional statement completed during the instance it represents, the control word has "forgotten" which branch has been executed. In this context, an instance is the equivalence class of all partial executions (trace prefixes) that generate the same control word.

Nevertheless, if we limit our instance domain to a particular execution of the program, it's easy to prove that no more that one trace prefix can be associated with a given control word. In that sense, a control word is representative of a unique instance during an execution.

3. Induction Variables and Binding Functions

Traditionally, addresses in a mono-dimensional array can be expressed via an integer index, and in a multi-dimensional one via a vector of integer indices. We limit ourselves to arrays indexed through affine expressions of some specific variables, from now onwards referred as *induction variables*.

3.1. Definition and Office

For each induction variable, a *binding function* maps each instance, i.e., each control word, to each value of this induction variable. Of course, the result makes sense only if the induction variable is syntactically alive at this instance. In `To_&_fro`, induction variables are the formal parameters of all functions. In `Pascaline`, the loop index is also an induction variable. As mentioned earlier, MOGUL syntax only allows definition of induction variables at the beginning of a block.

Since the control word loses the trail of any operation executed during a completed statement, the exigency of exactness in references computation dictates that a statement execution has no side effects upon induction variables defined outside of that statement. The environment is thus characterized by the extended stack, and induction variables are managed in single-assignment, quite similarly to a pure functional language.

Another constraint concerns the type of operations over the induction variables. Indeed, a modification of one induction variable has to be modeled as a transition in a transducer.

3.2. Operations Over the Induction Variables

Two types of operations are allowed over induction variables. Let us index respectively by 1 and 2 the values before and after the operations.

AdfBdfBdeCabFacG	$\overline{G}H$	$\overline{H}I$	$\overline{IcaFbaC}$	*DabFacG*	$\overline{G}H$	$\overline{H}I$	$\overline{IcaFbaD}$	*E*	$\overline{EedBfdBfdA}$	***Full trace***
AdfBdfBdeCabFacG	*H*	*I*		*DabFacG*	*H*	*I*		*E*		***Simplified trace***
AdfBdfBdeCabFac		*I*								***Control word***
AdfBdfBde				*D*						”
AdfBdfBde				*DabFac*		*I*				”
AdfBdfB								*E*		”

Figure 4. A trace and some associated *control words*

- Given a induction variable `i`, the operation may initialize the value of `i` to a constant: $i_2 = k$.

- Given two induction variables `i` and `j` (possibly identical), we may define `i` with the value of `j` plus an increment: $i_2 = j_1 + k$.

Of course, depending on the statement, several induction variables may be defined simultaneously according to these rules. For our both running examples, it's immediate that the required exigencies are satisfied.

3.3. Binding Transducer

The representation of binding functions is called the *product transducer*. The latter is built as follows: except for the initial state, we duplicate the control automaton so that each induction variable is assigned its own copy of each control automaton state. Each state is a "meeting point" for the values of a given induction variable, for any control word at this program point. In addition, each statement label is paired with its induction variable definitions, in order to map control words to the value of every induction variable. Indeed, each transition is labeled with a pair of a statement label and a constant, we call such pairs *bilabels*, see Figure 6. If the induction variable `q` inherits its value from `p` at the entrance of statement `s`, as $q = p + k$, then the product transducer has an edge from the node dedicated to `p` *before* `s` to the one dedicated to `q` *after* `s`.

The case of constant assignment $x = k$ is specific: we consider an additional induction variable named `z` whose value is always 0. When `x` is defined to k at the opening of statement `s`, the node dedicated to induction variable `x` after `s` is the arrival state of an edge coming from the node dedicated to `z` just before execution of `s`, with the bilabel $(s|k)$.

We may suppress from the tranducer any subgraph referring to an induction variable that is not alive at the corresponding program points, or — in the case of `z` — not useful for computation. Remaining nodes are final.

Figure 6 presents the product transducer for `Pascaline`, much simpler than `To_&_fro`'s one. Notice the loop iteration (g) is described by a loop edge. Moreover, the case of the initialization of induction variable `I` at loop entry G is interesting: it is captured by a "transversal" edge from a state associated with the additional induction variable `Z` to a state associated with `I`.

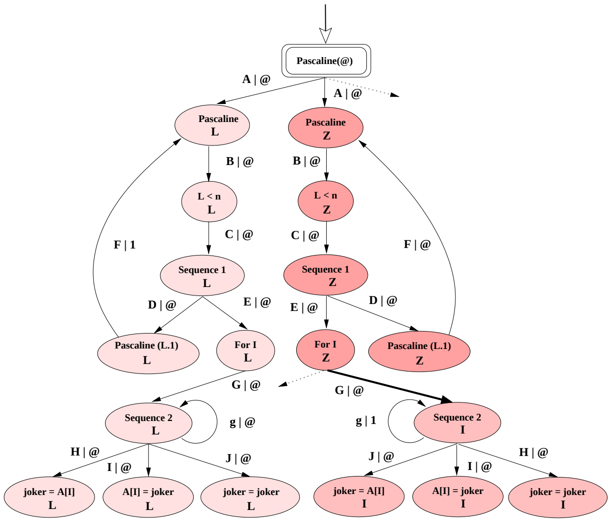


Figure 6. Product Transducer of `Pascaline`

Building the product transducer is *linear* in the number of statements and also *linear* in the number of induction variables. Surprisingly, the previously available algorithm was exponential in the worst case [3]: it relied on Gaussian elimination on the non-commutative semi-ring of regular expressions.

4. Instancewise Dependence Test

Let us now apply the instancewise information captured by the binding transducer to dependence test and parallelization. In this part, the analysis must be applied separately to each array, but we will omit the array name to simplify notations.

4.1. Exploitation of the Binding Information

We focus on two forms of automatic parallelization:

- scheduling of instructions and procedure calls within a program block;

- discovery of procedure calls that may safely be made asynchronous, enabling parallel execution of the callee with the following code in the caller.

Recursive programming blurs the distinction between data and control parallelism, since any data-parallel loop may be rewritten as a procedure with asynchronous recursive calls — with a low sequential overhead. These two forms of parallelization are based on a restricted type of block-related dependence information, the *synthetic dependence graph* [7]: considering an instruction s preceding an instruction s' in the same program block,[1] there is a synthetic dependence from s to s' if and only if two instances spawned by s and s' access the same memory location. If s and s' are not synthetically dependent, they may be safely reordered or executed asynchronously *within the enclosing block*.[2] Formally, the synthetic dependence relation, denoted by Δ is, where Σ^* is the control words free monoid:

$$s\Delta s' \iff \exists u, v, v' \in \Sigma^* : B(usv) = B(us'v'). \qquad (1)$$

The set of instances spawned by statement s will be called the *cone from summit* s. The partial order between instances s and s' defined in (1) is the *cone-to-cone order* $\ll_{s,s'}$.

An example will be presented in Section 5.

4.2. The Dual Dependence Test Transducer

Figure 7 supports the following presentation. The *conflict transduction* $\kappa = B^{-1} \circ B$ maps each instance to each of the instances which potentially access the same references. Because the intermediate monoid is $\mathbb{Z}^n$, κ is not a rational transduction. Instead, we stick to transduction compositions where the intermediate monoid is free: the Elgot and Mezei theorem [6, 1] shows that such a composition yields a rational transduction, and provides a quadratic algorithm to build the resulting transducer.

The *dependence transduction* $\delta_{s,s'} = \kappa \cap \ll_{s,s'}$ extracts from κ cone-to-cone ordered couples of control words, i.e., it describes the instancewise dependence relation between control words according to the binding function B and the cone-to-cone order $\ll_{s,s'}$. Dependence transductions and synthetic dependences are linked through the following result:

$$\delta_{s,s'} \neq \emptyset \iff s\Delta s'. \qquad (2)$$

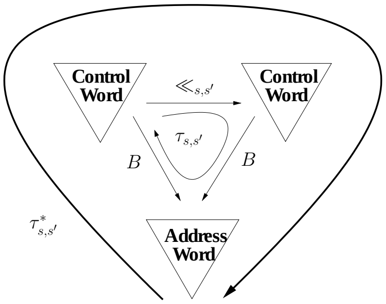


Figure 7. Dual Dependence Test Transducer

To decide emptiness of $\delta_{s,s'}$ is equivalent to test whether the *dependence test transduction* $\tau_{s,s'} = \kappa \circ \ll_{s,s'}$ is disjoint from the *identity transduction* $=$. Like κ, $\tau_{s,s'}$ is not a rational transduction. Therefore, this way of describing dependences does not lead to a computable method. Let us define the *dual dependence test transduction* $\tau^*_{s,s'} = B \circ \ll_{s,s'} \circ B^{-1}$; $\tau^*_{s,s'}$ is rational since the intermediate language during both compositions is the control words free monoid Σ^*. Figure 7 shows that the former $\delta_{s,s'}$ emptiness problem of boils down to the disjointness of $\tau^*_{s,s'}$ and the identity transduction $=$.

Building the dual dependence test transducer involves two compositions with transducers of the same order of magnitude in states and edges number, i.e., in the total number of statements. Its complexity is thus cubic in the number of statements. This sometimes leads to huge intermediate transducers with more than 10000 states and edges. Fortunately, these transducers are very redundant and a minimization procedure (including determinization) succeeds in achieving a simplification by a factor of 100 or more, leading to less than 10 states and edges in many cases. Improving the composition algorithm in order to limit the size of intermediate structures is a very promising topic, but the small size of the result is already sufficient to apply a somewhat more complex dependence test, as we will see in the next section.

4.3. Algorithm for Existence Test

Considering a d-dimensional array, we have just shown that dependence between two statements s and s' is equivalent to the existence of a d-dimensional vector v such that $v\tau^*_{s,s'}v$ (without taking loop bounds and conditionals into account).

From a transducer implementing $\tau^*_{s,s'}$, let us build a directed graph $G_{s,s'}$ whose vertices and edges are the states and the transitions of the transducer, respectively, labeling each edge with the difference between the transition's output and input vectors. E.g., a transition labeled $-1|2$ yields

[1] By definition, s and s' cannot label exclusive branches of a conditional.

[2] A block closing implicitly corresponds to a synchronization, waiting for completion of every statement in the block.

an edge labeled 3. Initial and final states in the transducer are called *initial* and *final vertices* in $G_{s,s'}$.

Dependence between s and s' is now equivalent to the following decision problem: "is there a zero-weight path between the initial and final vertices in $G_{s,s'}$?"

Decidability. Let us examine first the case of arrays of dimension 1. From $G_{s,s'}$, we may build a push-down automaton with empty alphabet and one stack symbol (one-counter automaton) where 0 in the counter is realized by the empty stack. Testing whether the language recognized by a push-down automaton is empty is a well-known decidable problem.

Concerning multi-dimensional arrays, note that computations over dimensions are independent from each other. So, its suffices to test emptiness of the intersection of each dimension languages.

A Flow Algorithm. This zero-weight path problem — though quite natural and similar to the flow problem — could not be found in the literature. It has been proved NP-complete, even in the acyclic case, through a simple reduction from the *bipartition* problem. Nevertheless, this problem is known to admit a rather efficient pseudo-polynomial solution, i.e., there is a polynomial algorithm when the integers involved are small. This is the case in practical graphs $G_{s,s'}$, since the labels are of the same order as the induction variable strides in the source program. We are thus encouraged in the search of an exact solution to the dependence problem.

Our proposed solution is not pseudo-polynomial yet, but seems very applicable to practical graphs. It relies on a two-step encoding into integer linear programming. The example in Figure 8 illustrates the formal presentation. Each edge e in $G_{s,s'} = (V, E)$ is associated with a variable k_e. We assume, without loss of generality, that $G_{s,s'}$ has only one initial (resp. final) vertex with no incoming (resp. outgoing) edges (this condition is easily achieved with two additional vertices and zero-labeled edges).

First of all, suppose that $G_{s,s'}$ is acyclic. Calling O the set of outgoing edges of the initial vertex and I the set of incoming edges of the final vertex, add constraints

$$\sum_{e \in O} k_e = 1 \quad \text{and} \quad \sum_{e \in I} k_e = 1. \qquad (3)$$

Then, calling O_v and I_v, respectively, the sets of outgoing and incoming edges of any non-initial and non-final vertex v, add constraint

$$\sum_{e \in O_v} k_e = \sum_{e \in I_v} k_e. \qquad (4)$$

The two previous constraints are analogous to the Kirchhoff law in electrical engineering. It is easy to show that any solution $(k_e)_{e \in E}$ satisfying these constraints describes a path between the initial and the final vertices: each edge e has k_e occurrences in this path (0 or 1 because the graph is acyclic). Finally, calling $(w_e^i)_{1 \leq i \leq d}$ the d-dimensional integer vector labeling edge e, add d constraints to enforce the zero-weight property.

$$\forall i, 1 \leq i \leq d : \sum_{e \in E} w_e^i k_e = 0. \qquad (5)$$

By construction, there is a dependence between s and s' if and only if the system of affine equalities $S_{s,s'} = (3, 4, 5)$ has a solution. We have just converted our flow problem into another, well known, NP-complete problem. Fortunately, system $(3, 4)$ is easy to solve using Gaussian elimination, since every minor of the constraint matrix is either singular or unimodular. Thus, the full system $S_{s,s'}$ can efficiently be addressed using integer linear programming techniques: traditional pivoting of the simplex method [13] applies to the unimodular part of the matrix, possibly with additional Gomory cuts [13] to handle the remaining constraints associated with (5).

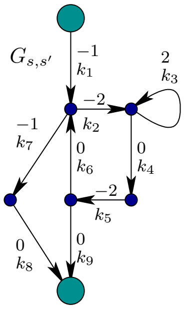
Now consider the case of a cyclic graph $G_{s,s'}$. The path constraint is not anymore enforced by $(3, 4)$: there may exist an isolated circuit disconnected from the main path between the initial and final vertices. Indeed, a circuit may only exist in a path if it is reachable from the initial vertex. In other words, performing a topological sort of $G_{s,s'}$, a backward edge targeting vertex v may be taken only if a forward edge targeting v is also taken. If we denote F_v and B_v the sets of forward and backward edges of v, respectively, this fact leads to a disjunction of affine constraints as follows:

$$\sum_{e \in B_v} k_e = 0 \quad \vee \quad \sum_{e \in F_v} k_e \geq 1.$$

In the worst case, each vertex v (targeted with backward edges) generates two additional constraints and, leading to an exponential number of systems. Feasibility of at least one of these systems is sufficient to show that s and s' are dependent. But an arbitrary association of such constraints yields no solution in general, because circuits are often sequenced on a single path or nested together in $G_{s,s'}$; e.g., a loop nest always leads to a linear number of systems.

Implementation and further study of the algorithm complexity are left for future work, but current results show that practical solutions to our dependence test problem do exist.

S_1 has no solution but S_2 has an infinite number of solutions, the shortest (in terms of the associated path) being $k_9 = 0$, $k_1 = k_2 = k_4 = k_5 = k_6 = k_7 = k_8 = 1$ and $k_3 = 3$.



$$S_1 = \begin{cases} k_1 = 1 \\ k_8 + k_9 = 1 \\ k_1 + k_6 = k_2 + k_7 \\ k_2 = k_4 \\ k_4 = k_5 \\ k_5 = k_6 + k_9 \\ k_7 = k_8 \\ k_3 = 0 \\ -k_1 - 2k_2 + 2k_3 - 2k_5 - k_7 = 0 \end{cases}$$

$$S_2 = \begin{cases} k_1 = 1 \\ k_8 + k_9 = 1 \\ k_1 + k_6 = k_2 + k_7 \\ k_2 = k_4 \\ k_4 = k_5 \\ k_5 = k_6 + k_9 \\ k_7 = k_8 \\ k_2 \geq 1 \\ -k_1 - 2k_2 + 2k_3 - 2k_5 - k_7 = 0 \end{cases}$$

Figure 8. Encoding into systems of affine inequalities

5. Implementation and Experiments

We realized in OCaml a complete implementation of the procedure that takes a MOGUL program source in input and returns the *dual dependence test transducer* according to the choice of data structure and *cone summits*.

In order to easily pass from any automaton or transducer to another one of either type, we have implemented a library of generic tools, parameterized by the types of state labels and transition labels. Graphs of automata and transducers are designed through the *dot* software [9]. Thanks to several steps of minimization, trimming and ad-hoc reduction, the dual dependence test transducer graph appeared very simple for the programs we tested, and answer to the dependence test might be found directly while examining it.

Experiments with the To_&_fro Program. Function Sort_ab is recalled in Figure 10.

Applying the dependence test algorithm on the dual dependence test transducers $\tau^*_{C,E}$ and $\tau^*_{D,E}$ yields two synthetic dependences, as expected intuitively. Figure 9 is the result of our implementation for $\tau^*_{C,E}$. Since all states, including the initial one, are final, dependence test answer is obviously "yes".

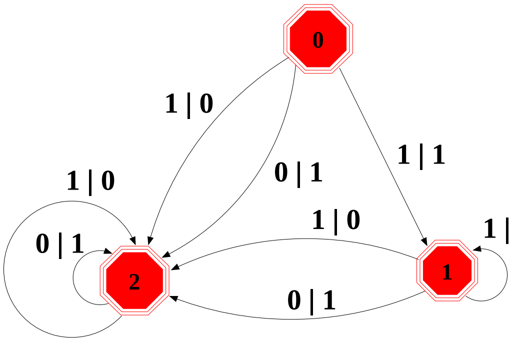


Figure 9. Test Transducer for $\tau^*_{C,E}$.

However, considering the dual dependence test transducer $\tau^*_{C,D}$, the algorithm should not be able to find a zero-weight path between the initial and final nodes on $G_{C,D}$, since the recursive calls to Sort_ba affect disjoint parts of the arrays (separated by the induction variable Q). Unfortunately, $G_{C,D}$ *does* hold such a zero-weight path, because the conditional expression enforcing the separation ($p < r$) has *not* been taken into account, thus responsible for the false dependence $C \Delta D$.

This remaining false dependence forbids the parallelization of function Sort_ab. Of course, our technique is fully successful on other interesting recursive programs, but this disappointing result is a motivation to further explore the direct handling of conditional expressions and loop bounds into the binding function. Updating the dependence test to the resulting multi-counter transducers (a.k.a. Minsky machines) will be difficult: the dependence test problem becomes undecidable. But several solutions are at hand, based on decidable sub-classes [2, 5].

Figure 11 shows the "corrected" synthetic dependence graph of function Sort_ab. Using the spawn/sync asynchronous function call from the Cilk parallelization tool [10], one may automatically deduce the parallel program in Figure 12.

Beyond the To_&_fro program. Figure 13 shows some facts about the recursive programs we implemented in MOGUL. Since our last publication in the field [3], we discovered many recursive algorithms suitable for implementation in MOGUL and instancewise dependence analysis. Therefore, it seems that the program model encompasses many implementations of practical algorithms despite its severe constraints.

6. Conclusion and Perspectives

We introduced by examples a language dedicated to a class of recursive programs. This enabled us to extract from them the instancewise mapping between run-time instances and array elements. This analysis is exploited for an instancewise dependence test that can handle multi-dimensional arrays. This problem was previously consid-

Code name	OCaml lines	Array references	Labels	Loops	Function calls	Product transducer nodes
Pascaline	21	2	15	1	2	42
Matrix-multiply	17	5	15	1	3	45
n-Queens	46	2	22	2	2	79
To_&_fro	115	12	82	0	19	416
Merge_sort+terminal_bubble_sort	78	14	71	2	14	602
Sort_3_colors	80	4	49	0	11	162

Figure 13. Examples of applicable recursive programs

```
  function Sort_ab (...) {
d   if test
f     if test
B       Sort_ab (P, Q.1, R);
e     else {
C       Sort_ba (P, P, Q.-1);
D       Sort_ba (Q, Q, R);
E       Merge_ab (P, Q, P);
      }
  }
```

Figure 10. Function `Sort_ab`

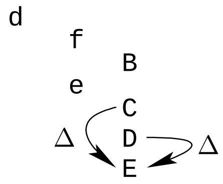


Figure 11. Synthetic dependence graph

```
  function Sort_ab (...) {
d   if test
f     if test {
B       spawn Sort_ab (P, Q.1, R);
        sync;
e     } else {
C       spawn Sort_ba (P, P, Q.-1);
D       spawn Sort_ba (Q, Q, R);
        sync;
E       spawn Merge_ab (P, Q, P);
        sync;
      }
  }
```

Figure 12. Parallel version of `Sort_ab`

ered undecidable in the general case [3]. An efficient algorithm has been presented: the binding function computation phase is polynomial and the dependence test itself has a “reasonable” exponential complexity (it belongs to a “good” class of integer linear programming).

However, much work is underway to make this technique practical. First of all, we are aware that many useful algorithms are not compatible with the induction variable constraints. In addition, recall that conditional expressions and loop bounds have not been taken into account, leading to several examples with false dependences. Generalization and/or graceful degradations of the framework should thus be developed. We are currently investigating both approaches.

Acknowledgments. We would like to thank Véronique Donzeau-Gouge and Catherine Dubois for their important contributions to the semantical foundation of the control word and induction variable concepts. The dependence test complexity has been studied with the help of Guillaume Huard who designed a simple reduction from the bipartion problem.

References

[1] J. Berstel. *Transductions and Context-Free Languages*. Teubner, Stuttgart, Germany, 1979.

[2] B. Boigelot and P. Wolper. Symbolic verification with periodic sets. In *Proc. of the 6^{th}conference on Computer-Aided Verification*, number 818 in LNCS, pages 55–76. Springer-Verlag, 1994.

[3] A. Cohen. *Program Analysis and Transformation: from the Polytope Model to Formal Languages*. PhD thesis, Université de Versailles, France, Dec. 1999. http://www-rocq.inria.fr/~acohen/publications/thesis.ps.gz.

[4] A. Cohen and J.-F. Collard. Instance-wise reaching definition analysis for recursive programs using context-free transductions. In *Parallel Architectures and Compilation Techniques*, pages 332–340, Paris, France, Oct. 1998. IEEE Computer Society Press.

[5] H. Comon and Y. Jurski. Multiple counters automata, safety analysis and presburger arithmetic. In A. Hu and M. Vardi, editors, *Proc. Computer Aided Verification*, volume 1427 of *LNCS*, pages 268–279, Vancouver, British Columbia, Canada, 1998. Springer-Verlag.

[6] C. C. Elgot and J. E. Mezei. On relations defined by generalized finite automata. *IBM Journal of Research and Development*, pages 45–68, 1965.

[7] P. Feautrier. A parallelization framework for recursive tree programs. In *EuroPar’98*, LNCS, Southampton, UK, Sept. 1998. Springer-Verlag.

[8] **M. Gupta, S. Mukhopadhyay, and N. Sinha. Automatic parallelization of recursive procedures. In** *Parallel Architectures and Compilation Techniques (PACT'99)*, **pages 139–148. IEEE Computer Society Press, Oct. 1999.**

[9] **E. Koutsofios and S. North.** *Drawing graph with* **dot, Feb. 2002.** http://www.research.att.com/sw/tools/graphviz/dotguide.pdf**.**

[10] **K. H. R. M. Frigo, C. E. Leiserson. The implementation of the cilk-5 multithreaded language. In** *ACM Symp. on Programming Language Design and Implementation (PLDI'98)*, **pages 212–223, Montreal, Quebec, Canada, June 1998.**

[11] **G. Rozenberg and A. Salomaa, editors.** *Handbook of Formal Languages*, **volume 1: Word Language Grammar. Springer-Verlag, 1997.**

[12] **R. Rugina and M. Rinard. Automatic parallelization of divide and conquer algorithms. In** *7th ACM Symp. on Principles and Practice of Parallel Programming (PPoPP'99)*, **Atlanta, Georgia, USA, May 1999.**

[13] **A. Schrijver.** *Theory of Linear and Integer Programming*. **John Wiley and Sons, Chichester, UK, 1986.**