Instancewise Program Analysis

Pierre Amiranoff *‡ Albert Cohen * Paul Feautrier †

* INRIA Rocquencourt, A3 Group
† INRIA Rhône-Alpes, CompSys Group, LIP, ÉNS Lyon
‡ CEDRIC Laboratory, CNAM Paris

Abstract

We introduce a general static analysis framework to reason about program properties at an infinite number of runtime control points, called instances. Infinite sets of instances are represented by rational languages. Based on this instancewise framework, we extend the concept of induction variables to recursive programs. For a class of monoid-based data structures, including arrays and trees, induction variables capture the exact memory location accessed at every step of the execution. This compile-time characterization is computed in polynomial time as a rational function.

1 Introduction

Static program analysis aims at the compile-time computation of program properties. Despite tremendous theoretical progresses and application success stories, this old problem is still a difficult one for two main reasons.

First of all, general properties of the concrete program semantics are *undecidable*, and most practical analyses evaluate conservative *approximations*. There are three main paradigms for static analysis [26]: type systems, data-flow (or constraint-based) analysis and abstract interpretation. Together, they contribute to the formalization, proof, computation and implementation of approximate static analyses and their applications. These approximate analyses do not capture cases in which exact properties can be evaluated on Turing-incomplete domain-specific languages — like bounded memory usage in synchronous languages [3] or array dependences on Fortran loop nests [15].

For a wide area of static analysis problems, there is another, more technical, but probably more important difficulty: static analyses lack the ability to *attach properties at an infinite set of control points*. Indeed, program semantics assigns "meaning" to a *finite set of syntactic elements* — statements or variables — using inductive definitions (rules, sequents, etc.). It is very natural to attach static properties to these syntactic elements: e.g., constant propagation [1] is interested in computing a property of a variable v at a statement s, asking whether v has some value v before s executes. For more complex analyses, attaching properties to a finite set of syntactic elements is not practical. E.g., *induction variable recognition* [18] captures the value of some variable v at a statement s as a function f_{v} of the number of times s has been executed. In other words, it captures v as a *function of the execution path* itself. Of course, the value of a variable at any stage of the execution is a function of the initial contents of memory and of the execution path leading to this stage. For complexity reasons, the execution path may not be recoverable from memory. In the case of induction variables, we may assume the number of executions of s is recorded as a genuine loop counter. From such a function f_{v}, we can discover other induction variables using analyses of linear constraints [12].

1.1 Statementwise Analysis.

We use the term *statementwise* to refer to the classical type systems, data-flow analysis and abstract interpretation frameworks, that define and compute program properties at each program statement. A typical example is static analysis by abstract interpretation [11, 9, 10]: it relies on the *collecting semantics* to operate on a lattice of abstract properties. This restricts the attachment of properties to a *finite set of control points*. Few works addressed the attachment of static properties at a finer grain than syntactic program elements. Refinement of this coarse grain abstraction involves a previous *partitioning* [9] of the *control points*: e.g., *polyvariant* analysis distinguishes the context of function calls, and *loop unfolding* virtually unrolls a loop several times. *Dynamic partitioning* [5] integrates partitioning into the analysis itself; but we are not aware of any type-system, abstract interpretation or data-flow analysis allowing the attachment of program properties to a *finitely-presented, unbounded set of control points*.[1]

1.2 Instancewise Analysis.

On the other hand, domain-specific approaches to static analysis are able to compute program properties at an *infinite number of control points*. The so-called *polytope model* encompasses most works on analysis and transformation of the (Turing-incomplete) class of *static-control programs* [15, 28], roughly defined as nested loops with affine loop bounds and array accesses. An *iteration vector* abstracts the runtime control point corresponding to a given iteration of a statement. Program properties are expressed and computed for each vector of values of the surrounding loops counters. Instead of iteratively merging data-flow properties, most analyses in the polytope model use algebraic solvers for the direct computation of symbolic relations: e.g., array dependence analysis uses integer linear programming [15]. Iteration vectors are quite different from time-stamps

[1] However, unbounded lattices have long been used to capture abstract properties in statementwise analyses [12, 13].

in control point partitioning techniques [5]: they are *multi-dimensional*, lexicographically ordered, *unbounded*, and constrained by Presburger formula [29].

First Contribution. We introduce a general static analysis framework for sequential procedural languages. Within this framework, one may *define, abstract and compute* program properties at an *infinite* number of *runtime control points*. Our framework is called *instancewise* and runtime points are further referenced as *instances*. We will formally define instances as *trace abstractions*, understood as iteration vectors extended to arbitrary recursive programs. The mathematical foundation for instancewise analysis is *formal language theory*: rational languages finitely represent infinite set of instances, and instancewise properties may be captured by rational relations [4]. This paper goes far beyond our previous attempts to extend iteration vectors to recursive programs, for the analysis of arrays [8, 7, 6, 2] or recursive data structures [16, 6].

Second Contribution. Building on the instancewise framework, we extend the concept of *induction variables* to arbitrary recursive programs. The valuation of induction variables is analog to parameter passing in a purely functional language: each statement is considered as a function, binding and initializing one or more induction variables. We propose two algorithms for the *exact* (i.e., non approximate) evaluation of induction variables. The result of these algorithms is a *binding function* mapping instances to the abstract memory locations they access. It is a *rational function* on the Cartesian product of two monoids and can be efficiently represented as a *rational transducer* [4].

To focus on the core concepts and contributions, we introduce MoGuL, a language with high-level constructs for traversing data structures addressed by induction variables in a *finitely presented monoid*. In a general-purpose (imperative or functional) language, our technique would require additional information about the shape of data structures, using dedicated annotations [22, 23, 17] or shape analyses [19, 31]. Despite the generality of the control structures in MoGuL, binding functions are *exact* and may be used to derive *alias* and *dependence* information of recursive programs with an unprecedented precision [6, 2].

Organization of the Paper. Section 2 describes the control structures and trace semantics of the MoGuL language. Section 3 defines the abstraction of runtime control points into instances. Section 4 extends induction variables to recursive control and data structures. Section 5 states the existence of rational binding functions. Section 6 addresses the computation and representation of binding functions as rational transducers. We consider practical examples in Section 7, before we conclude and outline ongoing and future work.

2 Control Structures and Execution Traces

We consider a simplified notion of *execution trace* with emphasis on the identification of runtime control points. For our purpose, a *trace* is a sequence of symbols called *labels* that denotes a *complete* execution of a program. Each label registers either the *beginning* of a statement execution or its *completion*. A *trace prefix* is the trace of a partial execution, given by a prefix of a complete trace. In the remainder, we will consider trace prefixes instead of the intuitive notion of runtime control point.

Figure 1 presents our running example. It features a recursive call to the `Toy` function, nested in the body of a *for* loop, operating on an array `A`. Thus, there is no simple way to remove the recursion. In this paper, *we will construct a finite-state representation for the infinite set of trace prefixes of* `Toy`*, then compute an exact finite-state characterization of the elements of* `A` *accessed by a given trace prefix.*

2.1 Control Structures in the MoGuL Language

Figure 2 gives the MoGuL version of `Toy`. It abstracts the shape of array `A` through a monoid type `Monoid_int`. Induction variables `i` and `k` are bound to values in this monoid. Traversals of `A` are expressed through `i`, `k` and the monoid operation ".". Further explanations about MoGuL data structures and induction variables are deferred to Section 4. We present in Figure 3 a simplified version of the MoGuL syntax, focusing on the control structures.

This is a C-like syntax with some specific concepts. Italic non-terminals are defined elsewhere in the syntax: ***elementary_statement*** covers the usual atomic statements, including assignments, input/output statements, void statements, etc.; ***predicate*** is a boolean expression; ***init_list*** contains a list of initializations for one or more loop variables, and ***translation_list*** is the associated list of constant translations for those induction variables.

Every executable part of a program is labeled, either by hand or by the parser.

2.2 Interprocedural Control Flow Graph

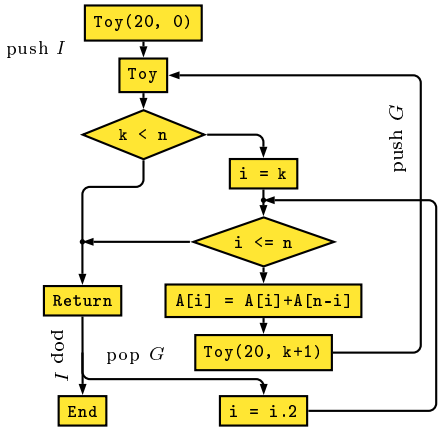


Figure 4: Interprocedural Control Flow Graph

We start with an intuitive presentation of the trace semantics of a MoGuL program, using the Interprocedural Control Flow Graph (ICFG): an extended control flow graph [1] with function call and return nodes. The ICFG associated to `Toy` is shown in Figure 4.

Each elementary statement, conditional and function call is a node of the ICFG, and more specifically:

- one node corresponds to each `block` entry;
- `for` loops generate three nodes: initialization (entry), condition (termination), and iteration;

```
int A[20];

void Toy(int n, int k) {
  if (k < n)
  {
    for (int i=k; i<=n;
         i+=2)
      {
        A[i] = A[i] + A[n-i];
        Toy(n, k+1);
      }
  }
  return
}

int main() {
  Toy(20, 0);
}
```

Figure 1: Program `Toy` in C

```
   structure Monoid_int A;

A  function Toy(Monoid_int n, Monoid_int k) {
B    if (k < n)
C    {
D      for (Monoid_int i=k; i<=n;
d           i=i.2)
E      {
F          A[i] = A[i] + A[n-i] ;
G          Toy(n, k.1);
       }
     }

   }

H  function main() {
I    Toy(20, 0);
   }
```

Figure 2: Program `Toy` in MoGuL

```
program          ::=  function                                                          (S1)
                    |  function program                                                 (S2)

function         ::=  'function' ident '(' formal_parameter_list ')' block              (S3)

block            ::=  LABEL ':' '{' init_list statement_list '}'                        (S4)
                    |  LABEL ':' '{' statement_list '}'                                 (S5)

statement_list   ::=  ε                                                                 (S6)
                    |  LABEL ':' statement statement_list                               (S7)

statement        ::=  elementary_statement ';'                                          (S8)
                    |  ident '(' actual_parameter_list ')' ';'                          (S9)
                    |  'if' predicate block 'else' block                               (S10)
                    |  'for' '(' init_list ';' predicate '; LABEL ':' translation_list ')' block   (S11)
                    |  block                                                           (S12)
```

Figure 3: Simplified MoGuL syntax (control structures)

- `return` nodes are implicitly added.

The iteration node follows the last node of the loop block and leads to the condition node. Given a function call `c` in the program source, there is an edge in the ICFG from the node associated to `c` to the corresponding function body. Moreover, there is an edge from the `return` node to the statement following the function call in the source program.

To forbid impossible matchings of function calls and returns, i.e., to preserve context-sensitivity [26], we provide the ICFG with a control stack [1], see Figure 4. The result is the graph of a pushdown automaton.

Due to loops and conditionals, some accepted paths correspond to valid execution traces, but others may still take wrong branches. Since we focus on a static scheme to name runtime control points, our trace semantics will make the same simplifying assumption and we will consider a superset of the valid traces.

2.3 The Pushdown Trace Automaton

Although MoGuL uses a C syntax, the instancewise framework in Section 3 considers each statement as a call to a function implementing elementary operations, conditional branches and iteration (as in a purely functional language). We extend the control stack of the ICFG to take these implicit calls into account. The stack alphabet now holds every statement label. Each statement is also provided an additional label to separate the implicit function call from the implicit return. If ℓ a label, ℓ corresponds to the beginning of the execution of a statement, and $\overline{\ell}$ indicates its completion. The first one labels arcs targetting the statement node, the second labels arcs departing from the node. Regarding the control stack, ℓ pushes ℓ while $\overline{\ell}$ pops ℓ.

The result is called the *pushdown trace automaton* and the recognized words are the *execution traces*.

When all states are considered final, the automaton recognizes all *trace prefixes*. It also recognizes prefixes of *non-terminating* traces when the program loops indefinitely. We thus exclude non-terminating programs in the following.

Figure 5 presents the trace pushdown automaton of the `Toy` program: $IBDF\overline{F}GBDF\overline{F}G\overline{G}dF\overline{F}G\overline{G}d\overline{ddDBG}dF$ is a prefix of a valid trace.

2.4 The Trace Grammar

After the intuitive presentation above, this section gives a formal definition of traces. There is one context-free trace grammar G_P per program P.

1. For each call to a function `id`, i.e., each derivation of production $(S9)$, there is a production schema

$$C_{\texttt{id}} \quad ::= \quad \texttt{Label}\ B_{\texttt{id}}\ \overline{\texttt{Label}} \qquad (1)$$

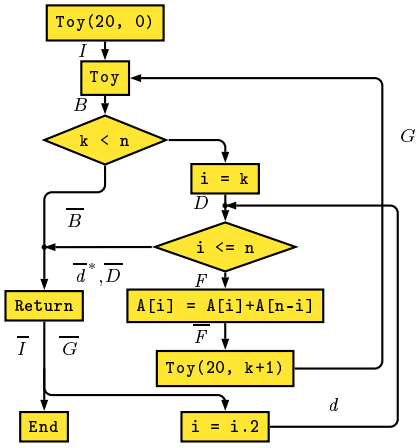


Figure 5: Pushdown Trace Automaton

where C_{id} and B_{id} are the respective non-terminals of the function call and body. `Label` is the terminal label of the call to function `id`, and $\overline{\texttt{Label}}$ marks the end of the statement, here a `return` statement.

2. For each loop statement s, i.e., each derivation of production $(S11)$, there are four production schemas

$$L_s \quad ::= \quad \varepsilon \quad | \quad \texttt{Label}_\texttt{e}\ B_s\ O_s\ \overline{\texttt{Label}_\texttt{e}} \qquad (2)$$

$$O_s \quad ::= \quad \varepsilon \quad | \quad \texttt{Label}_\texttt{i}\ B_s\ O_s\ \overline{\texttt{Label}_\texttt{i}} \qquad (3)$$

where the three non-terminals L_s, O_s and B_s correspond to the loop entry, iteration and body, respectively. $\texttt{Label}_\texttt{e}$ and $\texttt{Label}_\texttt{i}$ are terminals, they are the labels of the loop entry and iteration, respectively.

3. For each conditional s, i.e., each derivation of production $(S10)$, there are two productions schemas

$$I_s \quad ::= \quad \texttt{Label}\ T_s\ \overline{\texttt{Label}} \quad | \quad \texttt{Label}\ F_s\ \overline{\texttt{Label}} \qquad (4)$$

where the three non-terminals I_s, T_s and F_s correspond to the conditional, `then` branch and `else` branch, respectively. `Label` is the terminal label of the conditional.

4. For each block s, i.e., each derivation of productions $(S4)$ or $(S5)$, there is a production schema

$$B_s \quad ::= \quad \texttt{Label}\ S_1\ \ldots\ S_n\ \overline{\texttt{Label}} \qquad (5)$$

where the non-terminal B_s corresponds to the block and non-terminals $S_1, \ldots, S_n$ correspond to each statement in the block. `Label` is the terminal label of B_s.

5. For each elementary statement s, there is a production schema

$$S_s \quad ::= \quad \texttt{Label}\ \overline{\texttt{Label}} \qquad (6)$$

where `Label` is the terminal label of statement s.

The axiom of the trace grammar is the non-terminal associated with the block of the `main` function.

Definition 1 (Trace Language) *The set of traces of a program P — called the* trace language *of P — is the set of terminal sentences of G_P.*

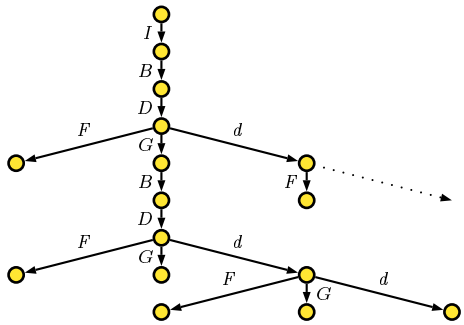
For a given execution trace t, runtime control points are sequentially ordered according to the appearance of statement labels in t.

Definition 2 (Sequential Order) *The* sequential order $<_{seq}$ *is the strict prefix order of the trace prefixes. It is a total order for a given execution trace.*

Calling L_{ab} the alphabet of labels, the *trace language* recognized by G_P is a context-free (a.k.a. algebraic) subset of the free monoid L_{ab}^*, and ε denotes its empty word. Clearly, the trace language fits the intuition about program execution and the previous presentation in terms of the interprocedural control flow graph: the pushdown trace automaton recognizes the trace language.

Grammar G_P generates many terminal sentences that are possible execution sequences for P. These sentences depend on choices between productions (1) to (6). In a real execution of P, these choices are dictated by the outcome of loop and test predicates, which our grammar does not take into account. It is customary to say that predicates are not interpreted (in the model theory sense), or that P is a *program schema* [25]. We are free to select which predicates and operations should be interpreted: e.g., the polytope model interprets every loop bound and array subscript in number theory [28]. In this paper, we will interpret address computations in the theory of finitely-presented monoids; everything else will remain uninterpreted.

Eventually, a runtime execution may be represented in the shape of an *activation tree* [1]: the sequential execution flow corresponds to the depth-first traversal of the activation tree. This representation is used in the formal definition of instances. Figure 6 shows an activation tree for `Toy`. We label each arc according to the target node statement. The trace is obtained while reading the word along the depth-first traversal: each downward step produces the arc label, and each upward step produces the associated overlined label.



Trace prefix: $IBDF\overline{F}GBDF\overline{F}G\overline{G}dF\overline{F}G\overline{G}d\overline{ddDBG}dF\ldots$

Figure 6: Activation tree

3 The Instancewise Model

This section is dedicated to the first part of our framework: the abstraction of *trace prefixes* into *control words*, the formal representation of *instances*. The control word abstraction characterizes an infinite set of trace prefixes in a tractable, finite-state representation. We present the properties of control words from several points of view: pushdown trace automata, trace prefixes, activation trees, and

MoGuL grammar. This last insight introduces a control words grammar that generates a superset of control words. We then investigate the conditions realizing the equivalence of the language generated by the control words grammar and the set of control words. This section ends with the description of the control word language in the form of a finite-state automaton, a counterpart of the pushdown trace automaton. Finally, we expose one of the main results of this work, justifying the introduction of control words as the basis for instancewise analysis.

3.1 From the Pushdown Trace Automaton to Control Words

The pushdown trace automaton will help us prove an important property of control words.

Definition 3 (Stack Word Language) *The stack word language of a pushdown automaton $\mathcal{A}$ is the set of stack words u such that there exist a state q in $\mathcal{A}$ for which the configuration (q, u) is both accessible and co-accessible — there is an accepting path traversing q with stack word u.*

Definition 4 (Control Word) *The stack word language of the pushdown trace automaton is called the* control word language. *A* control word *is the sequence of labels of all statements that have begun their execution but not yet completed it. Any trace prefix has a corresponding control word.*

Since the stack word language of a pushdown automaton is rational [30], we have:

Theorem 1 *The language of control words is rational.*

The activation tree is a convenient representation of control words. When the label of node n is at the top of the control stack, the control word is the sequence of labels along the *branch* of n in the activation tree, i.e., the path from the root to node n [1]. Conversely, a word labeling a branch of the activation tree is a control word. For example, $IBDdF$ is the control word of trace prefix $IBDF\overline{F}GBDF\overline{F}G\overline{G}dF\overline{F}G\overline{G}d\overline{ddDBG}dF$ in Figure 6.

3.2 From Traces to Control Words

The trace language is a Dyck language [4], i.e., a hierarchical parenthesis language. The restricted Dyck congruence over L_{ab}^* is the congruence generated by $\ell\overline{\ell} \equiv \varepsilon$, for all $\ell \in L_{\mathrm{ab}}$.[2] This definition induces a rewriting rule over L_{ab}^*, obviously confluent. This rule is the direct transposition of the control stack behavior. Applying it to any trace prefix p we can associate a minimal word w.

Lemma 1 *The control word w associated to the trace prefix p is the shortest element in the class of p modulo the restricted Dyck congruence.*

Definition 5 (Slimming Function) *The* slimming function *maps each trace prefix to its associated control word.*

Theorem 2 *The set of control words is the quotient set of trace prefixes modulo the restricted Dyck congruence, and the slimming function is the canonical projection of trace prefixes over control words.*

[2] The *restricted* qualifier means that only $\ell\overline{\ell}$ couples are considered, $\overline{\ell}\ell$ being a nonsensical sub-word for the trace grammar.

From now on, the restricted Dyck congruence will be called the *slimming congruence*. The following table illustrates the effect of the slimming function on a few trace prefixes.

Trace prefix	$IBDF\overline{F}GBDF$
Control word	$IBD\quad GBDF$
Trace prefix	$IBDF\overline{F}GBDF\overline{F}G\overline{G}dF\overline{F}G$
Control word	$IBD\quad GBD\quad d\quad G$
Trace prefix	$IBDF\overline{F}GBDF\overline{F}G\overline{G}dF\overline{F}G\overline{G}d\overline{ddDBG}dF$
Control word	$IBD\qquad\qquad\qquad\qquad\qquad dF$

The slimming function extends Harrison's Net function, and control words are very similar to his *procedure strings* [21]. Harrison introduced these concepts for a statementwise analysis with dynamic partitioning.

3.3 From the Trace Grammar to Control Words

We may also derive a *control words grammar* from the trace grammar. This grammar significantly differs from the trace grammar in three ways.

1. Control words contain no overlined labels.
   The control stack ignores overlined labels.
2. Each non-terminal is provided an empty production.
   A control word is associated to each trace prefix.
3. If the right-hand side of a production consists of multiple non-terminals, it is replaced by an individual production for each non-terminal.
   Only the last statement of an uncompleted sequence remains in the control stack, i.e., in the control word.

Under these considerations, the productions for the control words grammar are the following, with the same notations and comments as the trace grammar.

1. For each function call `id`, i.e., each derivation of production $(S9)$, there are two productions
$$C_{\texttt{id}} \;::=\; \texttt{Label}\ B_{\texttt{id}} \;\mid\; \varepsilon$$
2. For each loop statement s, i.e., each derivation of production $(S11)$, there are six productions
$$L_s \;::=\; \texttt{Label}_{\texttt{e}}\ B_s \;\mid\; \texttt{Label}_{\texttt{e}}\ O_s \;\mid\; \varepsilon$$
$$O_s \;::=\; \texttt{Label}_{\texttt{i}}\ B_s \;\mid\; \texttt{Label}_{\texttt{i}}\ O_s \;\mid\; \varepsilon$$
3. For each conditional s, i.e., each derivation of production $(S10)$, there are three productions
$$I_s \;::=\; \texttt{Label}\ T_s \;\mid\; \texttt{Label}\ F_s \;\mid\; \varepsilon$$
4. For each block s enclosing n statements, i.e., each derivation of $(S4)$ or $(S5)$, there are $n+1$ productions
$$B_s \;::=\; \texttt{Label}\ S_1 \;\mid\; \cdots \;\mid\; \texttt{Label}\ S_n \;\mid\; \varepsilon$$
5. For each elementary statement s,
$$S_s \;::=\; \texttt{Label} \;\mid\; \varepsilon$$

The axiom of this grammar is the block of the `main` function.

The control words grammar grammar above is right linear,[3] hence its generated language is rational.

[3] At most one non-terminal in the right-hand side, and non-terminals are right factors.

Lemma 2 *The language of control words is a subset of the language generated by the control words grammar.*

The proof comes from the three above observations that translate the effect of the slimming function. For each trace grammar derivation, we associate a corresponding derivation of the control words grammar. The control words grammar generates any stack word corresponding to a path — accepting or not — in the pushdown trace automaton.

The next section will show that the control words grammar only generates control words, assuming the trace grammar satisfies a termination criterion.

3.4 Control Words and Program Termination

Assuming *any incomplete execution can be completed until the termination of the program*, stack words corresponding to a path of the pushdown automaton are all stack words of trace prefixes, i.e., control words.

Conversely, if a partial execution has entered a step where the last opened statement can never be completed, a recursive cycle in the trace derivation cannot be avoided.

Example. Consider the following trace grammar:

$$\begin{array}{lcllcl} S & \to & aA\underline{bba} & B & \to & fC\underline{f} \\ A & \to & cB\underline{c} & C & \to & gB\underline{g} \\ A & \to & de\underline{ed} & & & \end{array}$$

a labels the body of function `main` and b labels an elementary statement. A is a non-terminal for a conditional test; function B is called in the `then` branch, while elementary statement s is executed in the `else` one. Function B calls function C and conversely. Thus, the `then` branch may never terminate. The corresponding control words grammar is:

$$\begin{array}{lcllcl} S & \to & aA & A & \to & \varepsilon \\ S & \to & ab & B & \to & fC \\ S & \to & \varepsilon & B & \to & \varepsilon \\ A & \to & cB & C & \to & gB \\ A & \to & de & C & \to & \varepsilon \end{array}$$

This grammar generates ac, thanks to the derivation

$$S \to aA;\ A \to cB;\ B \to \varepsilon.$$

However, no trace prefix can be generated by the trace grammar for which the control word is ac, hence ac is not a control word. To avoid this, we need a criterion that forbids recursive trap cycles. This criterion is defined through the structure of the trace grammar; we refer to the definition of a *reduced grammar* [32].

Definition 6 (Reduced Grammar) *A* reduced *grammar is a context-free grammar such that:*

1. *there is no $A \to A$ rule;*
2. *any grammar symbol occurs in some sentential form (a sentential form is any derivative from the axiom);*
3. *any non-terminal produces some part of a terminal sentence.*

The third rule is the criterion we are looking for: a non-terminal which produces some part of a terminal sentence is said *active*. The control words grammar of the program must have only active non-terminals; it is called an *unlooping* grammar. In the previous example, B and C are *not active*.

Termination criterion for the trace grammar. Starting from a set of non-terminals N, we recall an inductive algorithm that determines the set of active non-terminals $N' \subseteq N$; if $N = N'$, the grammar is unlooping [32]. The initial set N'_1 contains active non-terminals that immediately produce a part of a terminal sentence; Φ denotes the set of grammar rules, T is the set of terminals, and m is the cardinal of N.

Algorithm 1

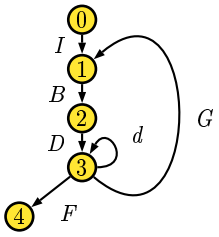
$N'_1 \leftarrow \{A \mid A \to \alpha \in \Phi \ \wedge \ \alpha \in T^*\}$
For $k = 2, 3, \ldots, m$
  $N'_k \leftarrow N'_{k-1} \bigcup \{A \mid A \to \alpha \in \Phi \wedge \alpha \in (T \bigcup N'_{k-1})^+\}$
  If $N'_k = N'_{k-1} \ \vee \ k = m$
  Then $N' \leftarrow N'_k$

Applied to our example where $N = \{S, A, B, C\}$:
$N'_1 = \{A\};\ N'_2 = \{A, S\};\ N'_3 = N'_2;\ N' = \{A, S\};\ N \neq N'$.

Thanks to Lemma 2, we may state a necessary and sufficient condition for the control words grammar to only generate control words.

Theorem 3 *Let P be a program given by its trace grammar G_P, and let G'_P be the associated control words grammar. The control words language of P is generated by G'_P if and only if Algorithm 1 concludes that G_P is unlooping.*

3.5 The Control Automaton



All states are final.

A few control words:
IBDdF,
IBDGBDF,
IBDGBDdG.

Figure 7: Example Control Automaton

We now assume the program satisfies Theorem 3.

It is easy to build a finite-state automaton accepting the language generated by the right-linear control words grammar, i.e., a finite-state automaton recognizing the language of control words. We call the latter the *control automaton*.

Figure 7 shows the control automaton for `Toy`; the control word language is $I + IB + IBD(d + GBD)^*(\varepsilon + F + G + GB)$.

The transformation from traces to control words is a systematic procedure. A similar transformation exists from the pushdown trace automaton to the control automaton; this is important for the design of efficient instancewise analysis algorithms (see Section 5).

- In the pushdown trace automaton, a sequence of successive statements takes is a chain of arcs, while, in the control automaton, each of these statement is linked by an edge from the common enclosing block, see Figure 8. Thus, the control automaton makes no distinction between the sequence and the conditional.
- As in the pushdown automaton for trace prefixes, all states are final.
- Since a `return` statement closes the corresponding function call and deletes every label relative to it in the control word, `return` nodes are not needed anymore.

Each statement in a sequence is linked to the enclosing block.

Figure 8: Construction of the Control Automaton

3.6 Instances and Control Words

Consider any trace t of a MoGuL program and any trace prefix p of t. The slimming function returns a unique control word. Conversely, it is easy to see that a given control word may be the abstraction of many trace prefixes, possibly an infinity. E.g., consider two trace prefixes differing only by the sub-trace of a completed conditional statement:[4] their control words are the same.

This section will prove that, during any execution of a MoGuL program, the stack that registers the control word at runtime cannot register twice the same control word (i.e., for two distinct trance prefixes). In others words, control words characterize runtime control points in a more compact way than trace prefixes. For the demonstration, we introduce a strict order over control words.

Definition 7 (Lexicographic Order) *We first define the partial* textual order $<_{lab}$ *over labels. Given* s_1 *and* s_2 *two labels in* L_{ab}, $s_1 <_{lab} s_2$ *if and only if*

- *there is a production generated by (5) in the trace grammar, such as* s_1 *is the label of* S_i *and* s_2 *is the label of* S_j, *with* $1 \leq i < j \leq n$;
- *or there is a production generated by (2) or (3) such as* s_1 *is the label of* B_s *and* s_2 *is the label of* O_s.

We denote by $<_{lex}$ *the strict lexicographic order over control words induced by* $<_{lab}$.

In other words, $<_{\text{lab}}$ is the textual order of appearance of statements within blocks, considering the loop iteration statement as textually ordered after the loop body.

Lemma 3 *The sequential order* $<_{seq}$ *over prefix traces is compatible with the slimming congruence. The lexicographic order* $<_{lex}$ *is the quotient order induced by* $<_{seq}$ *through the slimming congruence.*

The proof takes two steps. First of all, let t be a trace and T its activation tree. The set of all paths in T is ordered by a strict lexicographic order, $<_T$, isomorphic to $<_{\text{lex}}$.

Then, let α be the function mapping any path in T to the last label of the path word (accurately speaking of the control word labeling this path). Given a trace prefix p and the $<_T$ ordered sequence $\{b_0 = \varepsilon, b_1, ..., b_n\}$ of all paths in T, the (partial) depth-first traversal of T until p yields the following word:

$$\text{dft}(p) \triangleq \alpha(b_0)\alpha(b_1)...\alpha(b_q),$$

where b_q is the branch of p, $q \leq n$. Now, the definition of $\text{dft}(p)$ is precisely p.

Let p_q and p_r be two prefixes of t, p_q being a prefix of p_r itself, and write

$$p_q = \alpha(b_0)\alpha(b_1)...\alpha(b_q), p_r = \alpha(b_0)\alpha(b_1)...\alpha(b_r).$$

[4] I.e., after both branches have been completed, the first sub-trace denoting the `then` branch and the other the `else` one

We have the following: $p_q <_{\text{seq}} p_r \iff b_q <_T b_r$. Together with the first step, $p_q <_{\text{seq}} p_r \iff b_q <_{\text{lex}} b_r$.

We now come to the formal definition of instances.

Definition 8 (Instance) *For a* MoGuL *program, an* instance *is a class of trace prefixes modulo the slimming congruence.*

It is fundamental to notice that, in this definition, instances do not depend on any particular execution.

From Lemma 3 and Theorem 2 (the slimming function is the canonical projection of trace prefixes to control words), we may state the two main properties of control words.

Theorem 4 *Given one execution trace of a* MoGuL *program, trace prefixes are in bijection with control words.*

Theorem 5 *For a given* MoGuL *program, instances are in bijection with control words.*

Theorem 4 ensures the correspondence between runtime control points and control words. Theorem 5 is just a rewording of Theorem 2, it states the meaning of control words across multiple executions of a program.

In the following, we will refer to instances or control words interchangeably, without naming a particular trace prefix representative.

4 Data Structure Model and Induction Variables

This section and the following ones apply instancewise analysis to the *exact characterization of memory locations* accessed by a MoGuL program. For decidability reasons, we will only consider a restricted class of data structures and addressing schemes:

- data structures do not support destructive updates (deletion of nodes and non-leaf insertions);[5]
- addressing data-structures is done through so called induction variables whose only authorized operations are the initialization to a constant and the associative operation of a monoid.

In this context, we will show that the value of an induction variable at some runtime control point — or the memory location accessed at this point — only depends on the instance. Exact characterization of induction variables will be possible at compile-time by means of so-called *binding functions* from control words to abstract memory locations (monoid elements), independently of the execution.

4.1 Data Model

To simplify the formalism and exposition, MoGuL data structures with side-effects must be *global*. This is not really an issue since any local structure may be "expanded" along the activation tree (e.g., several local lists may be seen as a global stack of lists).

[5] Leaf insertions are harmless if data-structures are implicitly expanded when accessed.

4.1.1 Data Structure Monoids

A *finitely-generated monoid* $M = (G, \equiv)$ is specified by a *finite* list of *generators* G and a *congruence* $\equiv$ given by a *finite* list of equations over words in G^*. Elements of M are equivalence classes of words in G^* modulo $\equiv$. When the congruence is empty, M is a *free monoid*. The operation of M is the quotient of the concatenation on the free monoid G^* modulo $\equiv$; it is an associative operation denoted by $\cdot$ with neutral element ε_m.

Definition 9 (Abstract Location) *An abstract memory location is a pair of a* data structure name *and an element of a finitely-generated monoid $M = (G, \equiv)$. It is represented by an* address word *in G^*. By definition, two congruent address words represent the same memory location.*

Typical examples are the n-ary tree — the free monoid with n generators (with an empty congruence) — and the n-dimensional array — the free commutative monoid $\mathbb{Z}^n$ (with vector commutation and inversion). See Section A.1 for a wider coverage.

4.2 Induction Variables

Traditionally, induction variables are scalar variables within loop nests with a tight relationship with the surrounding loop counters [1, 18]. This relationship, deduced from the regularity of the induction variable updates, is a critical information for many analyses (dependence, array region, array bound checking) and optimizations (strength-reduction, loop transformations, hoisting).

A *basic linear induction variable* `x` is assigned (once or more) in a loop, each assignment being of the form `x = `c or `x = x + `c, where c is a constant known at compile-time. More generally, a variable `x` is called a *linear induction variable* if on every iteration of the surrounding loop, `x` is added a constant value. This is the case when assignments to `x` in the cycle are in the basic form or in the form `x = y + `c, `y` being another induction variable. The value of `x` may then be computed as an affine function of the surrounding loop counters.

MoGuL extensions are twofold:

- induction variables are not restricted to arrays but handle all monoid-based data structures;
- both loops and recursive function calls are considered.

As a consequence, induction variables represent abstract addresses in data structures, and the basic operation over induction variables becomes the monoid operation.

Definition 10 (Induction Variable) *A variable* `x` *is an induction variable if and only if the three following conditions are satisfied:*

a. `x` *is defined at a* `block` *entry, a* `for` *loop initialization, or* `x` *is a formal parameter;*

b. `x` *is constant in the* `block`*, the* `for` *loop or the function where it has been defined;*

c. *the definition of* `x` *(according to* **a***) is in one of the forms:*

1. `x = `c*, and c is a constant known at compile-time,*
2. `x = y` $\cdot$ c*, and* `y` *is an induction variable, possibly equal to* `x`*.*

A MoGuL induction variable can be used in different address expressions which reference *distinct* data structures, provided these structures are defined over the same monoid. This separation between data structure and shape follows the approach of the declarative language $8_{1/2}$ [20]. It is a convenient way to expose more semantics to the static analyzer, compared with C pointers or variables of product types in ML.

Eventually, the MoGuL syntax is designed such that *every variable of a monoid type is an induction variable*, other variables being ignored. The only valid definitions and operations on MoGuL variables are those satisfying Definition 10. Data structure accesses follow the C array syntax: `D[x]` denotes element `x` of structure `D`. The same syntax holds for all monoid shapes.[6]

5 The Binding Function

In MoGuL, the computations on two induction variables in two distinct monoids are completely separate. Thus, without loss of generality, we suppose that all induction variables belong to a single monoid M_{loc}, with operation $\cdot$ and neutral element ε_m, called the *data structure monoid*.

5.1 From Instances to Memory Locations

In a purely functional language, function application is the only way to define a variable. In MoGuL, every statement is handled that way; the scope of a variable is restricted to the statement at the beginning of which it has been declared, and an induction variable is constant in its scope.

Since overloading of variable names occurs at the beginning of each statement, the value of an induction variable depends on the runtime control point of interest. Let `x` be an induction variable, we define the *binding for* `x` as the pair $(p,\ v_p)$, where p is a trace prefix and v_p the value of `x` after executing p.

Consider two trace prefixes p_1 and p_2 representative of the same instance. The previous rules guarantee that all induction variables living right after p_1 (resp. p_2) have been defined in statements not closed yet. Now, the respective sequences of non-closed statements for p_1 and p_2 are identical and equal to the control word of p_1 and p_2. Thus the bindings of `x` for p_1 and p_2 are equal. In others words, the function that binds the trace prefix to the value of `x` is compatible with the slimming congruence.

Theorem 6 *Given an induction variable* `x` *in a* MoGuL *program, the function mapping a trace prefix p to the value of* `x` *only depends on the instance associated to p, i.e., on the control word.*

In other words, given an execution trace the bindings at any trace prefix are identified by the control word (i.e., the instance).

Definition 11 (Binding Function) *A* binding for `x` *is a couple (w, v), where w is a control word and v the value of* `x` *at the instance w.*

$\Lambda_{\mathtt{x}}$ *denotes the* binding function for `x`*, mapping control words to the corresponding value of* `x`*.*

[6] If `A` is an array (i.e., `A` is addressed in a free commutative group), the affine subscript `A[i+2j-1]` is not a valid MoGuL syntax. This is not a real limitation, however, since affine subscripts may be replaced by new induction variables defined every-time `i` or `j` are defined.

5.2 Bilabels

We now describe the mathematical framework to compute binding functions.

Definition 12 (Bilabel) *A* bilabel *is a pair in the set* $L^*_{ab} \times M_{loc}$. *The first part of the pair is called the* input label, *the second one is called the* output label.

$B = L^*_{\mathrm{ab}} \times M_{\mathrm{loc}}$ denotes the set of bilabels. From the *direct product* of the control word free monoid L^*_{ab} and the data monoid M_{loc}, B is provided with a monoid structure: its operation $\bullet$ is defined componentwise on L^*_{ab} and M_{loc},

$$(\alpha|a) \bullet (\beta|b) \stackrel{def}{=} (\alpha\beta|a \cdot b). \tag{7}$$

A binding for an induction variable is a bilabel. Every statement updates the binding of induction variables according to their definitions and scope rules, the corresponding equations will be studied in Section 5.3.

Definition 13 *The set of* rational subsets *of a monoid* M *is the least set that contains the finite subsets of* M, *closed by union, product and the star operation [4].*

A rational relation *over two monoids* M *and* M' *is a rational subset of the monoid* $M \times M'$.

We focus on the family B_{rat} of rational subsets of B.

Definition 14 *A semiring is a monoid for two binary operations, the "addition"* $+$, *which is commutative, and the "product"* $\times$, *distributive over* $+$; *the neutral element for* $+$ *is the zero for* $\times$.

The powerset of a monoid M is a semiring for union and the operation of M [4]. The set of rational subsets of M is a sub-semiring of the latter [4]; it can be expressed through the set of rational expressions in M. Thus B_{rat} is a semiring.

We overload $\bullet$ to denote the product operation in B_{rat}; $\emptyset$ is the zero element (the empty set of bilabels); and the neutral element for $\bullet$ is $\mathcal{E} = \{(\varepsilon, \varepsilon_m)\}$. From now on, we identify B_{rat} with the set of rational expressions in M, and we also identify a singleton with the bilabel inside it: $\{(s|c)\}$ may be written $(s|c)$.

5.3 Building Recurrence Equations

To compute a finite representation of the binding function for each induction variable, we show that the bindings can be expressed as a finite number of rational sets. First of all, bindings can be grouped according to the last executed statement, i.e., the last label of the control word. We build a system of equations in which unknowns are sets of bindings for induction variable `x` at state n of the control automaton. Given $\mathcal{A}_n$ the control automaton modified so that n is the unique final state, let $\mathcal{L}_n$ be the language recognized by $\mathcal{A}_n$. The *binding function for* `x` *at state* n, $\Lambda^n_{\mathtt{x}}$, is the binding function for `x` restricted to $\mathcal{L}_n$. We also introduce a new induction variable `z`, *constant and equal to* ε_m.

The system of equations is a direct translation of the semantics of induction variable definitions; it follows the syntax of a MoGuL program P; we illustrate each rule on the running example.

1. At the initial state 0 and for any induction variable `x`,

$$\Lambda^0_{\mathtt{x}} = \mathcal{E} \tag{8}$$

E.g., the `Toy` program involves three induction variable, the loop counter `i` and the formal parameters `k` and `n`. We will not consider `n` since it does not subscript any data structure. The output monoid is $\mathbb{Z}$, its neutral element ε_m is 0.

$$\Lambda^0_{\mathtt{k}} = \Lambda^0_{\mathtt{i}} = (\varepsilon|0).$$

2. $\Lambda^n_{\mathtt{z}}$ denotes the set defined by

$$\Lambda^n_{\mathtt{z}} = \bigcup_{w \in \mathcal{L}_n} (w|\varepsilon_m). \tag{9}$$

$\Lambda^n_{\mathtt{z}}$ is the binding function for the new induction variable `z` restricted to $\mathcal{L}_n$; it is constant and equal to ε_m.

For each statement s defining an induction variable `x` to $c_{s\mathtt{x}}$ (case **c**.1 of Definition 10), and calling d and a the respective departure and arrival states of s in the control automaton,

$$\Lambda^a_{\mathtt{x}} \supseteq \Lambda^d_{\mathtt{z}} \bullet (s|c_{s\mathtt{x}}). \tag{10}$$

Since $\Lambda^d_z \bullet (s|c_{s\mathtt{x}}) = \bigcup_{w \in \mathcal{L}_d} (ws|c_{s\mathtt{x}})$, (10) means: if $w \in \mathcal{L}_d$ is a control word, ws is also a control word and its binding for `x` is $(ws|c_{s\mathtt{x}})$.

The control automaton automaton of `Toy` has 5 states. For the case **c**.1 of Definition 10,

$$\text{statement } I: \quad \mathtt{k = 0}, \tag{11}$$

and (10) yields

$$\Lambda^1_{\mathtt{k}} \supseteq \Lambda^0_{\mathtt{z}} \bullet (I|0).$$

3. For each statement s defining an induction variable `x` to `y` $\cdot$ c (case **c**.2 of Definition 10), and d and a the respective departure and arrival states of s,

$$\Lambda^a_{\mathtt{x}} \supseteq \Lambda^d_{\mathtt{x}} \bullet (s|c_{s\mathtt{x}}). \tag{12}$$

To complete the system, we add for every induction variable `x` unchanged by s a set of equations in the form (12), where $c_{s\mathtt{x}} = \varepsilon_m$.

E.g., for case **c**.2 of Definition 10,

$$\text{statement } G: \quad \mathtt{k = k \cdot 1} \tag{13}$$
$$\text{statement } d: \quad \mathtt{i = i \cdot 2} \tag{14}$$
$$\text{statement } D: \quad \mathtt{i = k} \tag{15}$$

and (12) yields

$$\begin{array}{ll}
\Lambda^1_{\mathtt{i}} \supseteq \Lambda^3_{\mathtt{i}} \bullet (G|0) & \Lambda^4_{\mathtt{i}} \supseteq \Lambda^3_{\mathtt{i}} \bullet (F|0) \\
\Lambda^1_{\mathtt{k}} \supseteq \Lambda^3_{\mathtt{k}} \bullet (G|1) & \Lambda^4_{\mathtt{k}} \supseteq \Lambda^3_{\mathtt{k}} \bullet (F|0) \\
\Lambda^2_{\mathtt{i}} \supseteq \Lambda^1_{\mathtt{i}} \bullet (B|0) & \Lambda^1_{\mathtt{z}} \supseteq \Lambda^0_{\mathtt{z}} \bullet (I|0) \\
\Lambda^2_{\mathtt{k}} \supseteq \Lambda^1_{\mathtt{k}} \bullet (B|0) & \Lambda^1_{\mathtt{z}} \supseteq \Lambda^3_{\mathtt{z}} \bullet (G|0) \\
\Lambda^3_{\mathtt{i}} \supseteq \Lambda^2_{\mathtt{k}} \bullet (D|0) & \Lambda^2_{\mathtt{z}} \supseteq \Lambda^1_{\mathtt{z}} \bullet (B|0) \\
\Lambda^3_{\mathtt{i}} \supseteq \Lambda^3_{\mathtt{i}} \bullet (d|2) & \Lambda^3_{\mathtt{z}} \supseteq \Lambda^2_{\mathtt{z}} \bullet (D|0) \\
\Lambda^3_{\mathtt{k}} \supseteq \Lambda^2_{\mathtt{k}} \bullet (D|0) & \Lambda^3_{\mathtt{z}} \supseteq \Lambda^2_{\mathtt{z}} \bullet (d|0) \\
\Lambda^3_{\mathtt{k}} \supseteq \Lambda^2_{\mathtt{k}} \bullet (d|0) & \Lambda^4_{\mathtt{z}} \supseteq \Lambda^3_{\mathtt{z}} \bullet (F|0)
\end{array}$$

Gathering all equations generated from (8), (10) and (12) yields a system $(\mathcal{S})$ of $n_v \times n_s$ equations with $n_v \times n_s$ unknowns, where n_v is the number of induction variables, including `z`, and n_s the number of statements in the program.[7]

`Toy` yields the system

[7] Some unknown sets correspond to variables that are not bound at the node of interest, they are useless.

$$
\begin{array}{ll}
\Lambda^0_{\mathtt{i}} = \mathcal{E} & \Lambda^3_{\mathtt{i}} = \Lambda^3_{\mathtt{i}} \bullet (d|2) + \Lambda^2_{\mathtt{k}} \bullet (D|0) \\
\Lambda^0_{\mathtt{k}} = \mathcal{E} & \Lambda^3_{\mathtt{k}} = \Lambda^3_{\mathtt{k}} \bullet (d|0) + \Lambda^2_{\mathtt{k}} \bullet (D|0) \\
\Lambda^0_{\mathtt{z}} = \mathcal{E} & \Lambda^4_{\mathtt{i}} = \Lambda^3_{\mathtt{i}} \bullet (F|0) \\
\Lambda^1_{\mathtt{i}} = \Lambda^3_{\mathtt{i}} \bullet (G|0) + (I|0) & \Lambda^4_{\mathtt{k}} = \Lambda^3_{\mathtt{k}} \bullet (F|0) \\
\Lambda^1_{\mathtt{k}} = \Lambda^3_{\mathtt{k}} \bullet (G|1) + (I|0) & \Lambda^1_{\mathtt{z}} = \Lambda^3_{\mathtt{z}} \bullet (G|0) + (I|0) \\
\Lambda^2_{\mathtt{i}} = \Lambda^1_{\mathtt{i}} \bullet (B|0) & \Lambda^2_{\mathtt{z}} = \Lambda^1_{\mathtt{z}} \bullet (B|0) \\
\Lambda^2_{\mathtt{k}} = \Lambda^1_{\mathtt{k}} \bullet (B|0) & \Lambda^3_{\mathtt{z}} = \Lambda^2_{\mathtt{z}} \bullet (D|0) + \Lambda^2_{\mathtt{z}} \bullet (d|0) \\
 & \Lambda^4_{\mathtt{z}} = \Lambda^3_{\mathtt{z}} \bullet (F|0)
\end{array}
$$

Let Λ be the set of unknowns for $(\mathcal{S})$, i.e., the set of $\Lambda^n_{\mathtt{x}}$ for all induction variables $\mathtt{x}$ and nodes n in the control automaton. Let C be the set of constant coefficients in the system. $(\mathcal{S})$ is a *left linear system of equations over* (Λ, C) [30]. Let X_i be the unknown in Λ appearing in the left-hand side of the i^{th} equation of $(\mathcal{S})$. If $+$ denotes the union in B_{rat}, we may rewrite the system in the form

$$\forall i \in \{1, \ldots, m\}, X_i = \sum_{j=1}^{m} X_j \bullet C_{i,j} + R_i, \qquad (16)$$

where R_i results from the terms $\Lambda^0_{\mathtt{x}} = \mathcal{E}$ in right-hand side. Note that $C_{i,j}$ is either $\emptyset$ or a bilabel singleton of B_{rat}. Thus $(\mathcal{S})$ is a *strict* system, and as such, it has a unique solution [30]; moreover, this solution can be characterized by a *rational expression* for each unknown set in Λ.

Definition 15 (Rational Function) *If M and M' are two monoids, a* rational function *is a function from M to M' whose graph is a rational relation.*

Combined with Theorem 6, we may conclude that the solution of $(\mathcal{S})$ is a characterization of each unknown set X_i in Λ as a rational function.

Theorem 7 *The binding function for a* MoGuL *program is a finite set of rational functions $\Lambda^n_{\mathtt{x}}$, for all induction variables $\mathtt{x}$ and nodes n in the control automaton.*

Properties of rational relations and functions are similar to those of rational languages [4]: membership, inclusion, equality, emptiness and finiteness are decidable, projection on the input or output monoid yields a rational sub-monoid, and rational relations are closed for union, star, product and inverse morphism, to cite only the most common properties. The main difference is that they are not closed for complementation and intersection, although a useful subclass of rational relations has this closure property — independently discovered in [27] and [6]. Since most of these properties are associated with polynomial algorithms, binding functions can be used in many analyses, see [7, 16, 6, 2] for our previous and ongoing applications to the automatic parallelization of recursive programs.

6 Computing the Binding Function

This section investigates the resolution of $(\mathcal{S})$. Starting from (16), one may compute the last unknown in terms of others:

$$X_m = C^*_{m,m}\Big(\sum_{i=1}^{m-1} X_j \bullet C_{i,j} + R_m\Big). \qquad (17)$$

The solution of $(\mathcal{S})$ can be computed by iterating this process analogous to Gaussian elimination. This was the first proposed algorithm [6]; but Gaussian elimination on non-commutative semirings leads to exponential space requirements. We propose two alternative methods to compute and represent the binding function effectively. The first one improves on Gaussian elimination but keeps an exponential complexity; it has a strong theoretical interest because it captures the *relations between all induction variables* in a *single representation*, see Section A.2. If we only need to represent induction variables *separately* from each other, this Section presents a polynomial algorithm.

We recall a few definitions and results about transducers [4].

Definition 16 *A* rational transducer *is a finite-state automaton where each transition is labeled by a pair of* input *and* output *symbols (borrowing from Definition 12), a symbol being a letter of the alphabet or the empty word.*[8]

A pair of words (u, v) is recognized *by a rational transducer if there is a path from an initial to a final state whose input word is equal to u and output word is equal to v.*[9]

Theorem 8 *A rational transducer recognizes a rational relation, and reciprocally.*

A transducer offers either a static point of view — as a machine that recognizes pairs of words — or a dynamic point of view — the machine reads an input word and outputs the set of image words.

The use of transducers lightens the burden of solving a system of regular expressions, but we lose the ability to capture all induction variables and their relations in a single object. The representation for the binding function of an induction variable is called the *binding transducer*.

Algorithm 2

Given the control automaton and a monoid with n_v induction variables (including $\mathtt{z}$), the binding transducer is built as follows.

- *For each control automaton state, create a set of n_v states, called a* product-state*; each state of a product-state is dedicated to a specific induction variable.*
- *Initial (resp. final) states correspond to the product-states of all initial (resp. final) states of the control automaton.*
- *For each statement s, i.e., for each transition (d, a) labeled s in the control automaton; call P^d and P^a the corresponding product-states; and create an associated* product-transition *t_s. It is a set of n_v transitions, each one is dedicated to a specific induction variable. We consider again the two cases mentioned in Definition (10.**c**).*
  - *case **c**.1: the transition runs from state $P^d_{\mathtt{z}}$ in P^d to the state $P^a_{\mathtt{x}}$ in P^a. The input label is s, the output label is the initialization constant c;*
  - *case **c**.2: the transition runs from state $P^d_{\mathtt{y}}$ in P^d to state $P^a_{\mathtt{x}}$ in P^a. The input label is s, the output label is the constant c;*

[8] Pair of words lead to an equivalent definition.

[9] A transducer is not reducible to an automaton with bilabels as elementary symbols for its alphabet; as an illustration, both paths $(x|\varepsilon)(y|z)$ and $(x|z)(y|\varepsilon)$ recognize the pair of words $(xy|z)$.

The binding transducer for `Toy` is shown in Figure 9. Notice that nodes allocated to the virtual induction variable `z` are not co-accessible except the initial state (there is no path from them to a final state), and initial states dedicated to `i` and `k` are not co-accessible either. These states are useless, they are trimmed from the binding transducer.

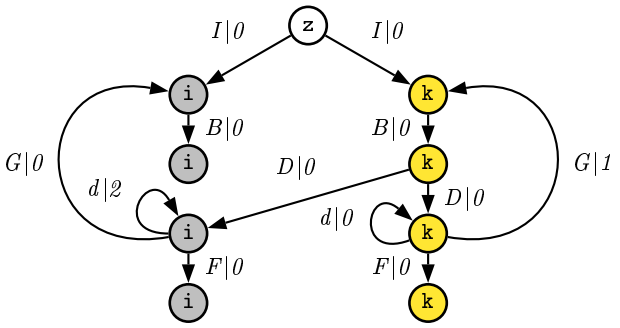


Figure 9: Binding Transducer for `Toy`

The binding transducer does not directly describe the binding function. A binding transducer is *dedicated* to an induction variable `x` when its final states are restricted to the states dedicated to `x` in the final product-states.

Theorem 9 *The binding transducer dedicated to an induction variable* `x` *recognizes the binding function for* `x`.

This result is a corollary of Theorem 7.

7 Experiments

The construction of the binding transducer is fully implemented in OCaml. Starting from a MoGuL program, the analyzer returns the binding transducer according to the choice of monoid. This analyzer is a part of a more ambitious framework including dependence test algorithms based on the binding transducer [2]. Our implementation is as generic as the framework for data structure and binding function computation: operations on automata and transducers are parameterized by the types of state names and transition labels. Graphs of automata and transducers are drawn by the free `dot` software [24].

Section A.3 presents two examples processed by our instancewise analyzer of MoGuL programs. The first one operates on an array, the second one on a tree.

Figure 10 summarizes some results about recursive programs we implemented in MoGuL. Since the first survey of instancewise analyses techniques [6], 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.

Program `n-Queens` is the classical problem to place n Queens on a $n \times n$ chessboard. `To_&_fro` is the recursive merge-sort algorithm alternating over two arrays. It is optimized in `To_&_fro+Terminal_insert_sort` by using an insertion sort for the leaves of the recursion (on small intervals of the original array). `Sort_3_colors` consists in sorting an array of balls according to one color among three, using only swaps. `Vlsi_test` simulates a test-bed to filter-out good chips from an array of untested ones; the process relies on peer-to-peer test of two chips, a good chip giving a certified correct answer about the other.

8 Conclusion and Perspectives

The instancewise paradigm paves the way for better, more precise program analyses. It decouples static analyses from the program syntax, allowing to evaluate semantic program properties on an infinite set of runtime control points. This paradigm abstracts runtime execution states (or trace prefixes) in a finitely-presented, infinite set of control words. Instancewise analysis is also an extension of the domain-specific iteration-vector approach (the so-called polytope model) to general recursive programs.

As an application of the instancewise framework, we extend the concept of induction variables to recursive programs. For a restricted class of data structures (including arrays and recursive structures), induction variables capture the exact memory location accessed at every step of the execution. This compile-time characterization, called the binding function, is a rational function mapping control words to abstract memory locations. We give a polynomial algorithm for the computation of binding functions.

Our current work focuses on instancewise alias and dependence analysis, for the automatic parallelization and optimization of recursive programs. We also look after new benchmark applications and data-structures to assess the applicability of binding functions; multi-grid and sparse codes are interesting candidates. We would also like to release a few constraints on the data structures and induction variables, aiming for the computation of approximate binding functions through abstract interpretation.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] P. Amiranoff, A. Cohen, and P. Feautrier. Instancewise array dependence test for recursive programs. In *Proc. of the 10th Workshop on Compilers for Parallel Computers*, Amsterdam, NL, Jan. 2003. University of Leiden.

[3] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1), Jan. 2003.

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

[5] F. Bourdoncle. Abstract interpretation by dynamic partitioning. *Journal of Functional Programming*, 2(4):407–423, 1992.

[6] 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`.

[7] A. Cohen and J.-F. Collard. Instancewise 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.

Code name	Data structure	Lines	Data references	Loops	Function calls	Transducer nodes
`Pascaline`	1D array	21	2	1	2	13
`Multiplication table`	2D array	17	5	1	3	22
`n-Queens`	1D array	39	2	2	2	27
`To_&_fro`	1D array	115	12	0	19	164
`Merge_sort_tree`	ternary tree	75	8	0	8	80
`To_&_fro+Terminal_insert_sort`	1D array	162	17	2	26	195
`Sort_3_colors`	1D array	80	4	0	11	97
`Vlsi_test`	linked lists	58	2	0	7	97

Figure 10: Sample recursive programs applicable to binding function analysis

[8] A. Cohen, J.-F. Collard, and M. Griebl. Data-flow analysis of recursive structures. In *Proc. of the 6th Workshop on Compilers for Parallel Computers*, pages 181–192, Aachen, Germany, Dec. 1996. Forschungszentrum Jülich.

[9] P. Cousot. *Semantic foundations of programs analysis.* Prentice-Hall, 1981.

[10] P. Cousot. Program analysis: The abstract interpretation perspective. *ACM Computing Surveys*, 28A(4es), Dec. 1996.

[11] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *4th ACM Symp. on Principles of Programming Languages*, pages 238–252, Los Angeles, California, USA, Jan. 1977.

[12] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th ACM Symp. on Principles of Programming Languages*, pages 84–96, Jan. 1978.

[13] A. Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *ACM Symp. on Programming Language Design and Implementation (PLDI'94)*, pages 230–241, Orlando, Florida, USA, June 1994.

[14] D. B. A. Epstein, J. W. Cannon, D. F. Holt, S. V. F. Levy, M. Paterson, and W. Thurston. *Word Processing in Groups.* Jones and Bartlett Publishers, Boston, 1992.

[15] P. Feautrier. Array expansion. In *ACM Int. Conf. on Supercomputing*, pages 429–441, St. Malo, France, July 1988.

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

[17] P. Fradet and D. L. Metayer. Shape types. In *24th ACM Symp. on Principles of Programming Languages (PoPL'97)*, pages 27–39, Paris, France, Jan. 1997.

[18] M. P. Gerlek, E. Stoltz, and M. Wolfe. Beyond induction variables: detecting and classifying sequences using a demand-driven ssa form. *ACM Trans. on Programming Languages and Systems*, 17(1):85–122, Jan. 1995.

[19] R. Ghiya and L. J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *23rd ACM Symp. on Principles of Programming Languages (PoPL'96)*, pages 1–15, St. Petersburg Beach, Florida, USA, Jan. 1996.

[20] J.-L. Giavitto, O. Michel, and J.-P. Sansonnet. Group-based fields. In *Proc. of the Parallel Symbolic Languages and Systems*, Oct. 1995. For a comprehensive presentation, see "Design and Implementation of $8_{1/2}$, a Declarative Data-Parallel Language, Research Report 1012, Laboratoire de Recherche en Informatique, Université Paris Sud (Paris XI), France, 1995".

[21] W. L. Harrison. The interprocedural analysis and automatic parallelisation of scheme programs. *Lisp and Symbolic Computation*, 2(3):176–396, Oct. 1989.

[22] L. J. Hendren, J. Hummel, , and A. Nicolau. Abstractions for recursive pointer data structures: improving the analysis and transformation of imperative programs. In *ACM Symp. on Programming Language Design and Implementation (PLDI'92)*, pages 249–260, San Francisco, Calfifornia, USA, June 1992.

[23] N. Klarlund and M. I. Schwartzbach. Graph types. In *20th ACM Symp. on Principles of Programming Languages (PoPL'93)*, pages 196–205, Charleston, South Carolina, USA, Jan. 1993.

[24] E. Koutsofios and S. North. *Drawing Graphs With* `dot`, Feb. 2002.
`http://www.research.att.com/sw/tools/graphviz/dotguide.pdf`.

[25] Z. Manna. *Mathematical Theory of Computation.* McGraw-Hill, 1974.

[26] F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer-Verlag, 1999.

[27] M. Pelletier and J. Sakarovitch. On the representation of finite deterministic 2-tape automata. *Theoretical Computer Science*, 225(1-2):1–63, 1999.

[28] G.-R. Perrin and A. Darte, editors. *The Data Parallel Programming Model.* Number 1132 in LNCS. Springer-Verlag, 1996.

[29] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):27–47, Aug. 1992.

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

[31] S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *26th ACM Symp. on Principles of Programming Languages (PoPL'99)*, pages 105–118, San Antonio, Texas, USA, Jan. 1999.

[32] J.-P. Tremblay and P.-G. Sorenson. *The theory and practice of compiler writing.* McGraw-Hill, 1985.

A Appendix

The following sections will not be included in the final version of the paper. The interested reader will be referred to a research report.

A.1 Monoid-Based Data Structures

Figure 11 lists some practical examples of monoid-based data structures.

A.2 Binding Matrix

M_{rat} denotes the set $B_{\text{rat}}^{m\times m}$ of square matrices of dimension m with elements in B_{rat}; M_{rat} is a semiring for the induced matrix addition and product and M_{rat} is closed by star operation [30]. The neutral element of M_{rat} is

$$\mathbb{E} = \begin{bmatrix} \mathcal{E} & & \emptyset \\ & \ddots & \\ \emptyset & & \mathcal{E} \end{bmatrix}. \tag{18}$$

Practical computation of the transitive closure of a square matrix C is an inductive process, using the following block decomposition where a and d are square matrices:

$$C = \begin{bmatrix} a & c \\ b & d \end{bmatrix}.$$

The formula is illustrated by the finite-state automaton in Figure 12; its alphabet is constituted of labels $\{a, b, c, d\}$ of the block matrices; i and j are the two states, they are both initial and final. If i and j denote the languages computed iteratively for the two states, and matrix C represents a linear transformation of the vector (i, j): $(i_1, j_1) = (i_0 a + j_0 b, i_0 c + j_0 d)$. We compute the transitive closure of C as the union of all words labeling a path terminated in states i or j, respectively, after zero, one, or more applications of C: $(i_*, j_*) = \big((i_0 + j_0 d^* b)(a + c d^* b)^*, (j_0 + i_0 a^* c)(d + b a^* c)^*\big)$. Writing $P = (a + c d^* b)^*$ and $Q = (d + b a^* c)^*$,

$$C^* = \begin{bmatrix} a & c \\ b & d \end{bmatrix}^* = \begin{bmatrix} P & d^* b P \\ a^* c Q & Q \end{bmatrix}. \tag{19}$$

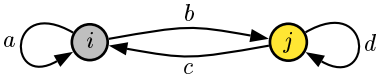


Figure 12: Computation of a matrix star

From (16), system $(\mathcal{S})$ can be written $X = XC + R$, where matrix $C = (C_{i,j})_{1\le i,j\le m}$ and vectors $R = (R_1, \ldots, R_m)$, $X = (X_1, \ldots, X_m)$. Vector RC^* is the solution of $(\mathcal{S})$, but direct application of (19) is still laborious, given the size of C.

Matrix Automaton. Our solution relies on the sparsity of C: we represent the system of equations in the form of an automaton $\mathcal{A}$, called the matrix automaton.

The graph of the matrix automaton is the same as the graph of the control automaton. Each statement s is represented by a unique transition, gathering all information about induction variable updates while executing s. The *binding function for* `x` *after statement* s, $\Lambda_{s\mathtt{x}}$, maps control words ended by s to the value of `x`. It is the set of all possible bindings for `x` after s. $\overrightarrow{\Lambda^n}$ denotes the *binding vector at state* n, i.e., the tuple of binding functions for all induction variables at state n (including `z`). Conversely, $\overrightarrow{\Lambda_s}$ denotes the *binding vector after statement* s, i.e., the tuple of binding functions for all induction variables after executing statement s.

With d the departure state of the transition associated to statement s, we gather the previous linear equations referring to s and present them in the form:

$$\forall \mathbb{S} \in M_{\text{rat}}, \overrightarrow{\Lambda_s} = \overrightarrow{\Lambda^d} \times \mathbb{S}. \tag{20}$$

As an example, we give the result for statement G of `Toy`:

$$\Lambda_{G\mathtt{i}} = \Lambda^3_{\mathtt{i}} \bullet (G|0),\ \Lambda_{G\mathtt{k}} = \Lambda^3_{\mathtt{k}} \bullet (G|1),\ \Lambda_{G\mathtt{z}} = \Lambda^3_{\mathtt{z}} \bullet (G|0)$$

$$\overrightarrow{\Lambda_G} = \overrightarrow{\Lambda^3} \times \begin{bmatrix} (G|0) & \emptyset & \emptyset \\ \emptyset & (G|1) & \emptyset \\ \emptyset & \emptyset & (G|0) \end{bmatrix}.$$

Now, the transition of statement s in $\mathcal{A}$ is labeled by the *statement matrix* $\mathbb{S}$. Thus, $\mathcal{A}$ recognizes words with alphabet in M_{rat}: concatenation is the matrix product and words are rational expression in M_{rat}, hence elements of M_{rat}. Grouping equations according to the transitions' arrival state, we get, for each state a,

$$\overrightarrow{\Lambda^a} = \sum_{d\in \text{pred}(a)} \overrightarrow{\Lambda^d} \times \mathbb{S}_{da}, \mathbb{S}_{da} \in M_{\text{rat}}, \tag{21}$$

where $\text{pred}(a)$ is the set of predecessor states of a and $\mathbb{S}_{da}$ is the statement matrix associated to the transition from d to a.

E.g., state number 1 in the matrix automaton of `Toy` yields

$$\overrightarrow{\Lambda^1} = \overrightarrow{\Lambda_I} + \overrightarrow{\Lambda_G} = \overrightarrow{\Lambda^0} \times \mathbb{I} + \overrightarrow{\Lambda^3} \times \mathbb{G}.$$

Theorem 10 *Let* $\overrightarrow{\Lambda^0} = (\mathcal{E}, \ldots, \mathcal{E})$ *be the binding vector at the beginning of the execution. The binding vector for any state* f *can be computed as*

$$\overrightarrow{\Lambda^f} = \overrightarrow{\Lambda^0} \times \mathbb{L}, \tag{22}$$

where $\mathbb{L}$ *is a matrix of regular expressions of bilabels;* $\mathbb{L}$ *is computed from the regular expression associated to the matrix automaton* $\mathcal{A}$*, when its unique final state is* f*.*

This result is a corollary of Theorem 7.

Application to the Running Example. We now give the statement matrices associated with equations (11) to (15). With the three induction variables `i`, `k` and `z`, the binding vector after statement I, $\overrightarrow{\Lambda_I} = (\Lambda_{I\mathtt{i}}, \Lambda_{I\mathtt{k}}, \Lambda_{I\mathtt{z}})$ and $\mathbb{I}$ the statement matrix for I, we have:

$$\overrightarrow{\Lambda_I} = \overrightarrow{\Lambda^0} \times \mathbb{I}, \quad \overrightarrow{\Lambda_B} = \overrightarrow{\Lambda^1} \times \mathbb{B}, \quad \overrightarrow{\Lambda_D} = \overrightarrow{\Lambda^2} \times \mathbb{D}$$
$$\overrightarrow{\Lambda_d} = \overrightarrow{\Lambda^3} \times , \quad \overrightarrow{\Lambda_G} = \overrightarrow{\Lambda^3} \times \mathbb{G}, \quad \overrightarrow{\Lambda_F} = \overrightarrow{\Lambda^3} \times \mathbb{F}$$

Free monoid.
$G = \{\texttt{right}, \texttt{left}\}$, $\equiv$ is empty, $\cdot$ is the concatenation: monoid elements address a binary tree.

Free group.
$G = \{\texttt{right}, \texttt{left}, \texttt{right}^{-1}, \texttt{left}^{-1}\}$, $\equiv$ is the inversion of `left` and `right` (without commutation): Cayley graphs [14, 20].

Free commutative group.
$G = \{(0,1),(1,0),(0,-1),(-1,0)\}$, $\equiv$ is the vector inversion and commutation, $\cdot$ is vector addition: a two-dimensional array.

Free commutative monoid.
$G = \{(0,1),(1,0)\}$, $\equiv$ is vector commutation: a two-dimensional grid.

Commutative monoid.
$G = \{(0,1),(1,0)\}$, $\equiv$ is vector commutation and $(0,1)\cdot(0,1) \equiv \varepsilon_m$: a two-dimensional grid folded on the torus $\mathbb{Z} \times \frac{\mathbb{Z}}{2\mathbb{Z}}$.

Free partially-commutative monoid.
$G = \{\texttt{next}, 1, -1\}$, $\equiv$ is the inversion and commutation of 1: nested trees, lists and arrays.

Monoid with right-inverse.
$G = \{\texttt{right}, \texttt{left}, \texttt{parent}\}$, $\texttt{right}\cdot\texttt{parent} \equiv \varepsilon_m$, $\texttt{left}\cdot\texttt{parent} \equiv \varepsilon_m$: a tree with backward edges.

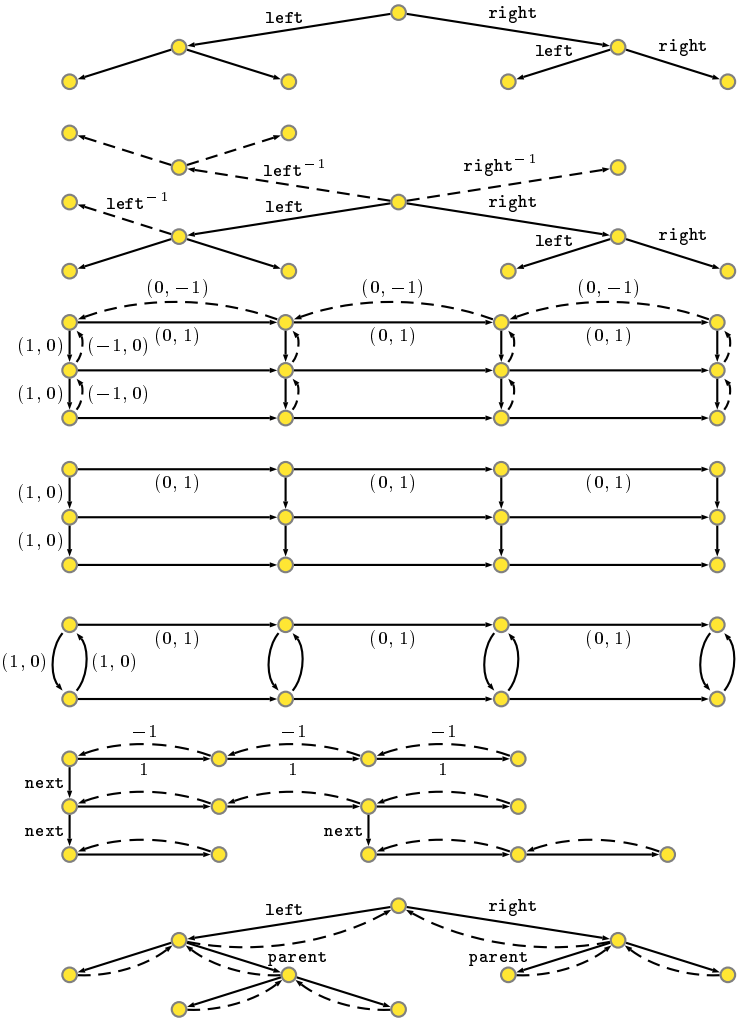


Figure 11: Monoid-based structures

with the following statement matrices:

$$\text{statement } I: \quad \mathbb{I} = \begin{bmatrix} I|0 & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset \\ \emptyset & I|0 & I|0 \end{bmatrix}$$

$$\text{statement } G: \quad \mathbb{G} = \begin{bmatrix} G|0 & \emptyset & \emptyset \\ \emptyset & G|1 & \emptyset \\ \emptyset & \emptyset & G|0 \end{bmatrix}$$

$$\text{statement } d: \quad = \begin{bmatrix} d|2 & \emptyset & \emptyset \\ \emptyset & d|0 & \emptyset \\ \emptyset & \emptyset & d|0 \end{bmatrix}$$

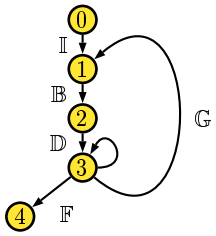
$$\text{statement } D: \quad \mathbb{D} = \begin{bmatrix} \emptyset & \emptyset & \emptyset \\ D|0 & D|0 & \emptyset \\ \emptyset & \emptyset & D|0 \end{bmatrix}$$

The other statements matrices let unchanged the induction variables.

$$\text{statement } B: \quad \mathbb{B} = \begin{bmatrix} B|0 & \emptyset & \emptyset \\ \emptyset & B|0 & \emptyset \\ \emptyset & \emptyset & B|0 \end{bmatrix}$$

$$\text{statement } F: \quad \mathbb{F} = \begin{bmatrix} F|0 & \emptyset & \emptyset \\ \emptyset & F|0 & \emptyset \\ \emptyset & \emptyset & F|0 \end{bmatrix}$$

The resulting matrix automaton is shown in Figure 13 (all states are final).



$$\mathbb{L} = \mathbb{I} + \mathbb{I}\mathbb{B} + \mathbb{I}\mathbb{B}\mathbb{D}\big(+\mathbb{G}\mathbb{B}\mathbb{D}\big)^* \big(\mathbb{E} + \mathbb{F} + \mathbb{G} + \mathbb{G}\mathbb{B}\big)$$

($\mathbb{E}$ is the neutral element of M_{rat}.)

Figure 13: Example of matrix automaton

About Complexity The exponential complexity of the matrix method has two explanations:

- the size of a regular expression can be exponentially larger than an equivalent finite-state automaton;
- despite our efforts to reduce the complexity of the transitive closure, we still achieve the simultaneous charac-

terization of all induction variables and their relations; this leads to a large number of (non-commutative) cross-products between regular expressions of different induction variables.

Indeed, a list of binding transducers for every individual induction variable may *not* be converted into a transducer for the full binding function in polynomial time. Intuitively, the alphabet of the latter must deal with *tuples of induction variables* with diverging evolutions in the control automaton. This shows that the exponential complexity of the matrix method is more fundamental than the fact we use regular expressions.

A.3 Examples

The `Pascaline` Program. Figure 14 shows a program to evaluate the binomial coefficients (a line of Pascal's triangle). It exhibits both a loop statement and a recursive call, two induction variables `I` and `L` plus the constant induction variable `n`; `x` and `y` are not induction variables. Statement D, `x = 1`, is an elementary statement without induction variables: MoGuL simply ignores it. The `else` branch of the conditional is empty: it ensures the termination of recursive calls.

```
    structure Monoid_int A;

A   function Pascaline(Monoid_int L, Monoid_int n) {
      int x, y;
B     if (L < n)
C     {
D       x = 1;
E       for (Monoid_int I=1; I<n;
e            I=I.1)
F       {
G           y = A[I];
H           A[I] = x + y;
I           x = A[I];
        }
J       Pascaline(L.1, n);
      }
    }

K   function Main() {
L     Pascaline(0, 10);
    }
```

Figure 14: Program `Pascaline`

Figure 15 shows the binding transducer for `Pascaline`, as generated by the software. The transducer is drawn by hand to enhance readability, and in complement with the indication of the dedicated induction variable, we filled each node of the graph with a statement borrowed from the program: the statement is written in the arrival nodes of the associated transitions. Nodes dedicated to the induction variable `n` are not used; they have been trimmed. Notice the use of induction variable `z` to initialize loop counter `I`.

The `Merge_sort_tree` Program. Figure 16 shows an implementation of the merge sort algorithm, implemented over a binary tree of lists, called `Tree`. The three functions `Split`, `Merge` and `Sort` are recursive. Induction variables `A`, `B` and `C` are locations in the tree; they are overloaded and exchanged as formal parameters of the three functions. Parameter `n` of `Split` is an independent induction variable not used for memory accesses, and `p`, `q` and `r` are not induction variables. `@` denotes the empty word, i.e., the root of the tree.

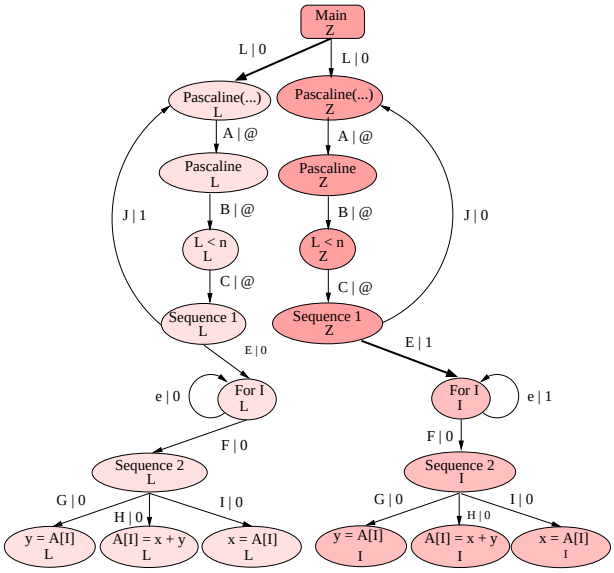


Figure 15: Binding transducer for `Pascaline`

At the beginning, the unsorted list is stored in the `next` branch of the tree named $Tree$. It is split in two halves stored in the `left` and `right` branches. Both these lists are recursively sorted, then merged back in the root node. Figure 17 shows the binding transducer for `Merge_sort_tree` as drawn by `dot` [24] from the MoGuL software output. Octogonal states correspond to the tree references at the elementary statements. These states are useful for the computation of data dependences. Indeed, from this binding transducer, we developed algorithms to detect that the two calls to the `Sort` function (j and k) can be run in parallel [16, 6].

```
    monoid Monoid_tree [next, left, right];
    structure Monoid_tree Tree;

t   function Main() {
s     Sort(Q, 37);
    }

S   function Split(Monoid_tree A, Monoid_tree B,
                   Monoid_tree C, Monoid_int n) {
F     if (n>0)
B     {
A       Tree[B] = Tree[A];
      }
L     if (n>1)
H     {
G       Tree[C] = Tree[A.next];
      }
R     if (n>2)
N     {
M       Split(A.next.next, B.next, C.next, n-2);
      }
    }
```

```
h   function Merge(Monoid_tree A, Monoid_tree B,
                   Monoid_tree C, int p, int q) {
g     if ((q != 0) && ( p = 0 || Tree[B] < Tree[C]))
V     {
T       Tree[A] = Tree[B];
U       Merge(A.next, B.next, C, q-1, p );
      }
e     else
d     {
c       if (p != 0)
Y       {
W         Tree[A] = Tree[C];
X         Merge(A.next, B, C.next, q, p-1);
        }
      }
    }

r   function Sort(Monoid_tree T, int r) {
q     if (r > 1)
m     {
i       Split(T, T.left, T.right, r);
j       Sort(T.left, (r+1)/2);
k       Sort(T.right, r/2);
l       Merge(T, T.left, T.right,(r+1)/2, r/2);
      }
    }
```

Figure 16: Program `Merge_sort_tree`

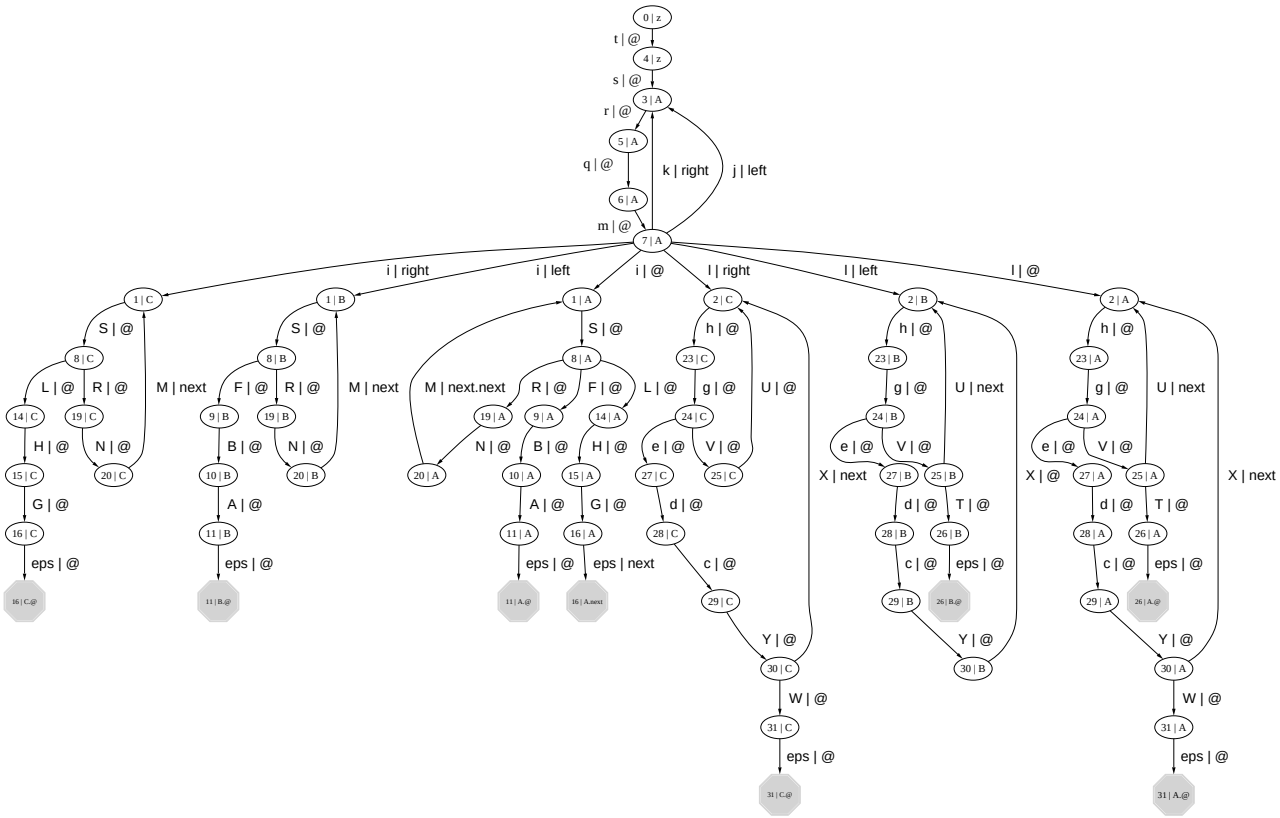


Figure 17: Binding transducer for `Merge_sort_tree`