Instancewise Program Analysis

Pierre Amiranoff and Albert Cohen
INRIA Futurs, ALCHEMY Group

Paul Feautrier
ENS Lyon, LIP, CompSys Group

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, an undecidable problem in general. Aiming at conservative approximations of static properties, three popular analysis paradigms have enabled tremendous theoretical progresses and a wide area of applications [41]: type systems, data-flow (or constraint-based) analysis and abstract interpretation. They contribute to the formalization, proof and implementation of analyses and their applications. These three paradigms are directly rooted into formal program semantics. Whether denotational, operational or axiomatic, program semantics assigns "meaning" to a *finite set of syntactic elements* — statements or variables — using inductive definitions (rules, sequents, etc.). When designing a static analysis framework to reason about programs (e.g., program transformation or verification), it is natural to attach static properties to these syntactic elements.

For example, it is natural to formalize constant-propagation [1] as a type system, in a data-flow setting or in abstract interpretation: to substitute a variable `v` by a value v in a statement s, one must analyze whether `v` has the value v before s executes, in every possible execution of the program.

Now, for some program analysis problems, computing static properties as functions of a finite set of syntactic elements is not practical. Consider *induction variable recognition* [28], a kind of extension of constant propagation to characterize the value of a variable `v` in a statement s as a function $f_{\mathtt{v}}$ of the number of times s has been executed (in the enclosing control structures). In other words, induction variable recognition 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 $f_{\mathtt{v}}$ for s, we can discover other induction variables using analyses of linear constraints [17, 40].

Besides induction variable recognition, many compilation problems have driven researchers off the three main analysis paradigms, to reason in different, ad-hoc terms instead. Analyses for loop-restructuring frameworks are typical examples [56], targeting vectorization, instruction-level, thread-level or data parallelism, scheduling, locality optimization, etc. [45, 2]. These problems share the need to characterize static properties as *functions of an infinite set of runtime control points, not as functions of syntactic program elements*. Indeed, it would be rather awkward to design a type system or data-flow analysis whose abstraction of the program semantics maps every individual iteration of a loop to a different property, as in the induction variable example [28].

1.1 Motivating Examples

Figure 1 shows a synthetic example where an array `A` is initialized in a loop nest and read in a recursive procedure. The footprint of reads to `A` in procedure `line` is a "chessboard": the procedure only reads elements `A[i][j]` such that $i+j$ is an even number. This observation leads to a simple optimization: half of the dynamic assignments to `A` in the loop nest are useless, they can be avoided through a simple transformation of the bounds and strides. We call this optimization *instancewise dead-code elimination*.

A typical technique to address this problem is called

array-region analysis [17, 18, 10]. However, because the "chessboard" footprint is not a convex polyhedron, all the array-region analyses we are aware of will fail on this example. In theory, recovering such precise information seems possible by abstract interpretation, provided the widening operator for $\mathbb{Z}$-polyhedra (also called lattice polyhedra) can handle some level of non-convexity [52, 36], which is not the case in the current state of the art [6]. In addition, such precision may only be achieved by a *context-sensitive* analysis [53, 33].

Figure 2 shows a slightly obfuscated version of the previous code: the procedure recursively swaps its arguments and only one the two initial calls remains in the loop. Although it may not seem obvious, this code has the same "chessboard" footprint as the previous one for reads to `A`, and the useless array assignments can still be removed. In this new form, it is much harder to imagine a precise enough widening operator. Recent techniques based on model-checking of push-down systems [22, 23, 8, 48] would suffer from a similar limitation: although such techniques provide virtually unlimited context-sensitivity, operations on polyhedra will incur to necessary approximations and lead to a convex region instead of a "chessboard".

This paper proposes a static analysis framework suitable for this kind of problems. Instead of searching for more precision in the lattice, *it provides unlimited precision and context-sensitivity in the control domain, computing static properties as functions of an infinite set of run-time program points*.

1.2 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 [16, 14, 15]: 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* [14] 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* [9, 38] integrates the refinement into the analysis itself. Control points can be extended with *call strings* (abstract call stacks) and *timestamps*, but ultimately rely on *k-limiting* [53, 33, 31, 55] or *summarization* heuristics [47] to achieve convergence. Although complex unbounded lattices are commonly used to capture properties [17, 19]), few works considered the computation of data-flow facts using an *unbounded set of control points*, following the seminal paper by Esparza and Knoop [22]. This approach is the closest to our work; it builds on model-checking of push-down systems to extend precision and context sensitivity, without sacrificing efficiency [23, 8, 48], but it ultimately results in the computation of data-flow properties as functions of a *finite* set of control points.

1.3 Instancewise Analysis

On the other hand, ad-hoc approaches to static analysis can represent and compute static program properties as *functions defined on an infinite (or unbounded) number of runtime control points*. For example, the *polytope model* encompasses most works on analysis and transformation of the (Turing-incomplete) class of *static-control programs* [24, 45], 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 as functions of vectors of values of the surrounding loop counters. In general, the result of the analysis is a mapping from the infinite set of iteration vectors (the runtime control points) to an arbitrary (analysis-specific) vector space. 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 [24]. Iteration vectors differ from time-stamps in control point partitioning techniques [9, 38]: they are multidimensional, lexicographically ordered, *unbounded*, and constrained by Presburger formula [52, 46].

1.4 Contributions

We introduce a general framework that uncompasses most ad-hoc formalisms for the fine grain analysis of loops and arrays in sequential procedural languages. Within this framework, one may define, abstract and compute program properties as functions of an infinite number of runtime control points. Our framework is called *instancewise* and runtime points are further referenced as *instances*. We define instances as trace abstractions, understood as iteration vectors extended to arbitrary recursive programs. Rational (a.k.a. regular) languages finitely represent infinite sets of instances, and instancewise properties may be captured by rational relations or functions [7]. This paper goes far beyond our previous attempts to extend iteration vectors to recursive programs, for the analysis of arrays [12, 11, 13, 3] or recursive data structures [26, 13, 11].

To illustrate instancewise analysis, we extend the concept of *induction variables* to arbitrary recursive programs. This demonstrates the characterization of static program properties as functions of an infinite set of runtime control points, beyond the domain of static-control Fortran loop nests. Technically, the valuation of induction variables is

```
int A[10][10];

void line (int i, int j) {
  ... = A[i][j];
  if (j<10) line (i, j+2)
}

int main () {
  for (i=0; i<10; i++)
    for (j=0; j<10; j++)
      A[i][j] = ...;
  for (i=0; i<10; i+=2) {
    line(i, j);
    line(i+1, j+1);
}
```

Figure 1: Dead-code

```
int A[10][10];

void line (int i, int j,
           int k, int l) {
  ... = A[i][j];
  if (j<10) line (k, l, i, j+2)
}

int main () {
  for (i=0; i<10; i++)
    for (j=0; j<10; j++)
      A[i][j] = ...;
  for (i=0; i<10; i+=2) {
    line(i, j, i+1, j+1);
}
```

Figure 2: Obfuscation

```
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 3: `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 4: `Toy` in MOGUL

analog to parameter passing in a purely functional language: each statement is considered as a function, binding and initializing one or more induction variables. Our analysis does not take the outcome of loop and test predicates into account,[1] hence we will consider a superset of the valid traces. We propose a polynomial algorithm, computing for each induction variable, a *binding function* mapping instances to the abstract memory locations they access. Each binding function is a *rational function* on the Cartesian product of monoids and can be represented as a *rational transducer* [7].

To focus on the core concepts and contributions, we introduce MOGUL, a domain-specific 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 [32, 34, 27] or shape analyses [29, 51]. Despite the generality of the control structures in MOGUL, binding functions give *exact* values for valid traces and this may be used to derive *alias* and *dependence* information of recursive programs with an unprecedented precision [11, 13, 3]. We will survey the current applications of instancewise analysis for recursive programs; the reader interested in more details (for loop nests or more general recursive programs) may refer to [13] for a pedagogical and synthetic presentation.

1.5 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 describe our implementation and some experiments with practical examples in Section 7. Section 8 studies applications of instancewise analysis to program optimization.

[1] This limitation can be overcome thanks to approximations and higher complexity algorithms. We will present solutions and applications in another paper.

2 Control Structures and Execution Traces

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 3 presents our running example. It features a recursive call to the `Toy` function, nested in the body of a loop, operating on an array `A`; there is no simple way to remove the recursion. We will construct a finite-state abstraction of the infinite set of trace prefixes of `Toy`, then compute a finite-state characterization of the function mapping trace prefixes to the elements of `A` it reads or writes.

2.1 Control Structures in MOGUL

Figure 4 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 $\cdot$. Further explanations about MOGUL data structures and induction variables are deferred to Section 4. We present in Figure 5 a simplified version of the MOGUL syntax, focusing on control structures.

This is a C-like syntax with some specific concepts. Italic non-terminals are defined elsewhere in the syntax: *`elem_stmt`* 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 *`trans_list`* is the associated list of constant translations for those induction variables; `block` collects a sequence of statements, possibly defining some induction variables. Every executable part of a program is labeled, either by hand or by the parser.

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

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

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

stmt_list::=ε                                                 (S6)
           |LABEL ':' stmt stmt_list                          (S7)

stmt     ::=elem_stmt ';'                                     (S8)
           |ident '(' actual_p_list ')' ';'                   (S9)
           |'if' predicate block 'else' block                (S10)
           |'for' '(' init_list ';'
                      LABEL ':' predicate ';'
                      LABEL ':' trans_list ')'
                    block                                    (S11)
           |block                                            (S12)
```

Figure 5: Simplified MOGUL syntax

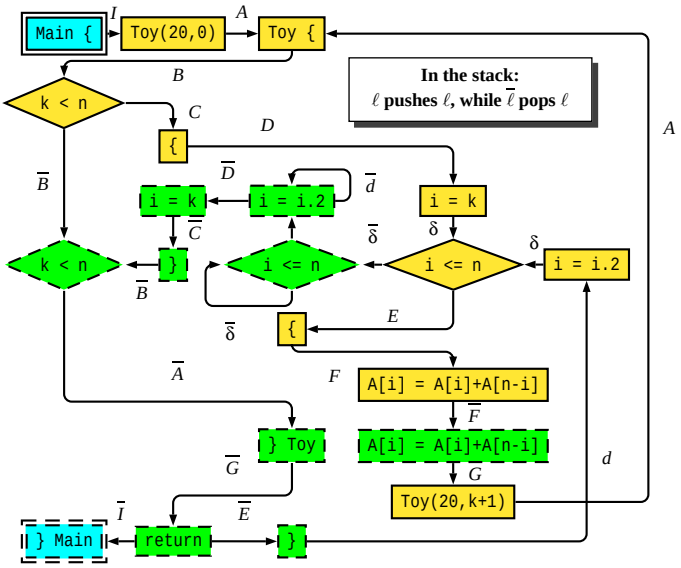


Figure 6: Pushdown trace automaton

2.2 The Pushdown Trace Automaton

We start with an intuitive presentation of the trace semantics of a MOGUL program, using an extended control flow graph [1] with function call/return nodes and a control stack.

Our framework considers each statement as a call to a function implementing elementary operations, conditional branches and iteration — as in a purely functional language. Each statement is provided with an additional label to separate the implicit function call from the implicit return. If ℓ is a label of a MOGUL statement, ℓ corresponds to the beginning of the execution of a statement, and $\overline{\ell}$ indicates its completion. Regarding the control stack, ℓ pushes ℓ while $\overline{\ell}$ pops ℓ. An additional state, called *return state*, is associated to the completion of each statement. Loops are just syntactic sugar for terminal recursion: the iteration node in a loop follows the last node in its body and leads to the condition node; a special state pops all iteration labels from the stack at loop exit. The result is called the *pushdown trace automaton*: it recognizes the *trace language*, i.e., the set of *execution traces*. Calling L_{ab} the alphabet of labels, the trace language is a context-free (a.k.a. algebraic) subset of the free monoid L_{ab}^*, and ε denotes its empty word. Figure 6 presents the trace automaton of the `Toy` program.

Since the underlying control-flow graph lacks the outcome of loop and test predicates, some accepted paths may still take wrong branches. To design a static scheme to name runtime control points, our trace semantics will make the same simplifying assumption and accept a superset of the valid traces.

For a given trace t, runtime control points are sequentially ordered according to label appearance in t: the *sequential order* $<_{\text{seq}}$ is the strict prefix order of the trace prefixes. It is a total order for a given execution trace.

When all states are considered final, the automaton recognizes all *trace prefixes*. It also accepts prefixes of *non-terminating* traces in case the program does not terminate. We exclude this possibility in the following. The following word is a prefix of a valid trace: $IABCD\delta EF\overline{F}GABCD\delta EF\overline{F}GAB\overline{B}A\overline{G}E\delta d\delta EF$.

For the sake of clarity, we simplify the push-down trace automaton, omitting return states, except for `Toy` calls, `block` statements and loop predicates. Now, the previous prefix reduces to: $IBDF\overline{F}GBDF\overline{F}GB\overline{B}G\overline{d}F$. We will use this simplified representation of traces in the following. To complement this intuitive presentation, Section A.1 gives a formal definition of traces.

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, and we conclude with a natural but fundamental property of control words, justifying their introduction as the basis for instancewise analysis.

3.1 From Traces to Control Words

The stack word language of a pushdown automaton $\mathcal{A}$ is the set of stack words u s.t. there is 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 1 (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 [49], the language of control words is rational.

A runtime execution may be represented in the shape of an *activation tree* [1], where sequential execution corresponds to depth-first traversal. Figure 7 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.

Activation trees provide a convenient interpretation 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 for the black node (runtime control point) in Figure 7.

Notice the trace language is a Dyck language [7], i.e., a hierarchical parenthesis language. The restricted Dyck congruence over L_{ab}^* is the congruence generated by $\ell\bar{\ell} \equiv \varepsilon$, for all $\ell \in L_{\text{ab}}$.[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: the control word w associated to the trace prefix p is the shortest element in the class of p modulo the restricted Dyck congruence. Let the *slimming function* denote the mapping of each trace prefix to its associated control word.

Theorem 1 *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.*

The restricted Dyck congruence is called the *slimming congruence*. The table in Figure 8 illustrates the effect of the slimming function on a few trace prefixes. The slimming function extends Harrison's NET function, and control words are very similar to his *procedure strings* [31]. Harrison introduced these concepts for a statementwise analysis with dynamic partitioning.

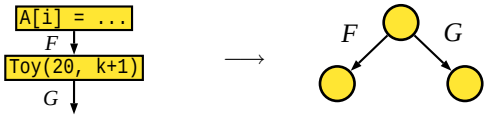
A formal construction from the trace grammar is presented in Section A.2.

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

3.2 The Control Automaton

It is easy to build a finite-state automaton recognizing the language of control words. We call the latter the *control automaton*.

Figure 9 shows the control automaton for `Toy`; the control word language is $I + IB + IBD(d + GBD)^*(\varepsilon + F + G + GB)$.



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

Figure 10: Construction

There is a systematic transformation from the pushdown trace automaton to the control automaton; it is important for the design of analysis algorithms.

- In the trace automaton, successive statements are chained in sequence, while in the control automaton, each statement is directly rooted in its enclosing block, see Figure 10 (as for conditional branches).
- As in the pushdown automaton for trace prefixes, all states are final.
- `return` nodes are not needed anymore.

3.3 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:[3] their control words are the same.

This section proves 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 trace 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.

We first define the partial *textual order* $<_{\text{lab}}$ over labels: $<_{\text{lab}}$ is the order of appearance of statements within blocks, considering the loop iteration statement as textually ordered after the loop body. $<_{\text{lex}}$ denotes the strict lexicographic order over control words induced by $<_{\text{lab}}$.

Lemma 1 *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.*

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

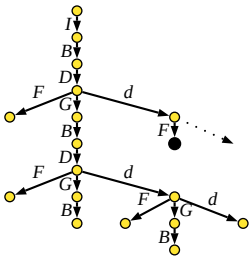


Figure 7: Activation tree

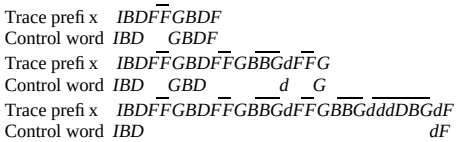
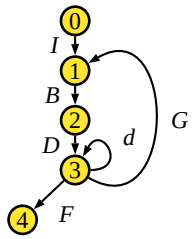


Figure 8: Slimming function



All states are final.

A few control words:
IBDdF,
IBDGBDF,
IBDGBDdG.

Figure 9: Control automaton

Section A.3 gives the proof. We now come to the formal definition of an instance.

Definition 2 (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 1 and Theorem 1 (the slimming function is the canonical projection of trace prefixes to control words), we may state the two main properties of control words.

Theorem 2 *Given one execution trace of a* MOGUL *program, trace prefixes are in bijection with control words.*

Theorem 3 *For a given* MOGUL *program, instances are in bijection with control words.*

Theorem 2 ensures the correspondence between runtime control points and control words. Theorem 3 is a rewording of Theorem 1, 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 Structures and Induction Variables

This section and the following ones apply instancewise analysis to the characterization of memory locations accessed by a MOGUL program as functions of runtime control points. For decidability reasons, we consider a restricted class of data structures and addressing schemes:

- destructive updates are forbidden (deletions and non-leaf insertions);[4]
- addressing data-structures is done through *induction variables* whose only authorized operations are the initialization to a constant and the associative operation of a monoid.

These restrictions are reminiscent of *purely functional data structures* [42].

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

In this context, we will show that the value of an induction variable at some runtime control 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.

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

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.

A data structure is a pair of a *data structure name* and a finitely-generated monoid $M = (G, \equiv)$. An abstract memory location in this data structure is an element of the monoid. 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). More details can be found in Figure 15 in the appendix.

Traditionally, induction variables are scalar variables within loop nests with a tight relationship with the surrounding loop counters [1, 28]. This relationship, deduced from the regularity of the inductive 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 in 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 address all monoid-based structures, not only arrays;
- both loops and recursive function calls are considered.

Thus, induction variables represent abstract addresses in data structures, and the basic operation over induction variables is the monoid operation.

Definition 3 (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 $8_{1/2}$ [30]. 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 3. For any monoid shape, data structure accesses follow the *C* array syntax: `D[a]` denotes element with address *a* of structure `D`, where *a* is in the form `x` or `x` $\cdot$ *c*, `x` an induction variable and *c* a constant.

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 when necessary while `i` or `j` are defined. As an illustration, let `k` be the induction variable equal to `i+2j-1`, the subscript in the reference above. We have to build, through a backward motion, static chains of induction variables from the program start point to the considered reference. Suppose the last modification of the subscript before the considered program point is given by the statement `j= h` denoted by *s*, where `h` is another induction variable. We have to define a new induction variable `g = i+2h-1`, living before this statement, and to consider that *s* initializes `k` through an additional assignment `k= g`. This work has to be done recursively for all paths in the control flow graph until reaching the start point.

5 Binding Functions

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 4 *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 4 (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`*.*

We now describe the mathematical framework to compute binding functions. A *bilabel* is a pair in the set $L_{\text{ab}}^* \times M_{\text{loc}}$. The first part of the pair is called the *input label*, the second one is called the *output label*. $B = L_{\text{ab}}^* \times M_{\text{loc}}$ 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},

$$(\alpha|a) \bullet (\beta|b) \stackrel{def}{=} (\alpha\beta|a \cdot b). \tag{1}$$

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 5.2.

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 [7]. 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. 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 [7]. The set of rational subsets of M is a sub-semiring of the latter [7]; 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 $(s|c)$.

5.2 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. Next, 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, Λ_{x}^n, 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 initial state 0 and for any induction variable x,

$$\Lambda_{\mathsf{x}}^0 = \mathcal{E} \tag{2}$$

E.g., the Toy program involves three induction variables, 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: $\Lambda_{\mathsf{k}}^0 = \Lambda_{\mathsf{i}}^0 = (\varepsilon|0)$.

2. Λ_{z}^n denotes the set defined by

$$\Lambda_{\mathsf{z}}^n = \bigcup_{w \in \mathcal{L}_n} (w|\varepsilon_m). \tag{3}$$

Λ_{z}^n 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 $c_{s\mathsf{x}}$ (case **c**.1 of Definition 3), and calling d and a the respective departure and arrival states of s in the control automaton,

$$\Lambda_{\mathsf{x}}^a \supseteq \Lambda_{\mathsf{z}}^d \bullet (s|c_{s\mathsf{x}}). \tag{4}$$

Since $\Lambda_{\mathsf{z}}^d \bullet (s|c_{s\mathsf{x}}) = \bigcup_{w \in \mathcal{L}_d} (ws|c_{s\mathsf{x}})$, (4) means: if $w \in \mathcal{L}_d$ is a control word, ws is also a control word and its binding for x is $(ws|c_{s\mathsf{x}})$.

The control automaton automaton of Toy has 5 states. For the case **c**.1 of Definition 3, statement I: k = 0, and (4) yields $\Lambda_{\mathsf{k}}^1 \supseteq \Lambda_{\mathsf{z}}^0 \bullet (I|0)$.

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

$$\Lambda_{\mathsf{x}}^a \supseteq \Lambda_{\mathsf{x}}^d \bullet (s|c_{s\mathsf{x}}). \tag{5}$$

To complete the system, we add for every induction variable x unchanged by s a set of equations in the form (5), where $c_{s\mathsf{x}} = \varepsilon_m$.

E.g., for case **c**.2 of Definition 3, statement G: k = k · 1, statement d: i = i · 2, statement D: i = k, and (5) yields

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

Gathering all equations generated from (2), (4) and (5) 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.[5]

Toy yields the system

$$\begin{array}{ll}
\Lambda_{\mathsf{i}}^0 = \mathcal{E} & \Lambda_{\mathsf{i}}^1 = \Lambda_{\mathsf{i}}^3 \bullet (G|0) + (I|0) \\
\Lambda_{\mathsf{k}}^0 = \mathcal{E} & \Lambda_{\mathsf{k}}^1 = \Lambda_{\mathsf{k}}^3 \bullet (G|1) + (I|0) \\
\Lambda_{\mathsf{z}}^0 = \mathcal{E} & \Lambda_{\mathsf{i}}^2 = \Lambda_{\mathsf{i}}^1 \bullet (B|0) \\
 & \Lambda_{\mathsf{k}}^2 = \Lambda_{\mathsf{k}}^1 \bullet (B|0)
\end{array}$$

[5] Some unknown sets are useless, they correspond to unbound variables.

$$\Lambda_i^3 = \Lambda_i^3 \bullet (d|2) + \Lambda_k^2 \bullet (D|0) \qquad \Lambda_z^1 = \Lambda_z^3 \bullet (G|0) + (I|0)$$
$$\Lambda_k^3 = \Lambda_k^3 \bullet (d|0) + \Lambda_k^2 \bullet (D|0) \qquad \Lambda_z^2 = \Lambda_z^1 \bullet (B|0)$$
$$\Lambda_i^4 = \Lambda_i^3 \bullet (F|0) \qquad \Lambda_z^3 = \Lambda_z^2 \bullet (D|0) + \Lambda_z^2 \bullet (d|0)$$
$$\Lambda_k^4 = \Lambda_k^3 \bullet (F|0) \qquad \Lambda_z^4 = \Lambda_z^3 \bullet (F|0)$$

Let Λ be the set of unknowns for $(\mathcal{S})$, i.e., the set of Λ_x^n for all induction variables 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) [49]. 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

$$\forall i \in \{1,\ldots,m\}, X_i = \sum_{j=1}^{m} X_j \bullet C_{i,j} + R_i, \tag{6}$$

where R_i results from the terms $\Lambda_x^0 = \mathcal{E}$ 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 [49]; moreover, this solution can be characterized by a *rational expression* for each unknown set in Λ.

If M and M' are two monoids, a *rational function* is a function from M to M' whose graph is a rational relation.

We may conclude that the solution of $(\mathcal{S})$ is a characterization of each unknown set X_i in Λ as a rational function: for any induction variable x and node n in the control automaton, the binding function for x restricted to $\mathcal{L}_n$ Λ_x^n is a rational function.

Since functions Λ_x^n are defined on disjoint subsets of control words, partitioned according to the suffix n, we eventually prove our main result.

Theorem 5 *For any induction variable* x*, the binding function for* x Λ_x *is a rational function.*

Properties of rational relations and functions are similar to those of rational languages [7]: 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 sub-class of rational relations has this closure property — independently discovered in [44] and [11]. Since most of these properties are associated with polynomial algorithms, binding functions can be used in many analyses, see [12, 26, 11, 3] for our previous applications to the automatic parallelization of recursive programs.

6 Computing Binding Functions

This section investigates the resolution of $(\mathcal{S})$. Starting from (6), one may compute the last unknown in terms of others:

$$X_m = C_{m,m}^* \Big(\sum_{i=1}^{m-1} X_j \bullet C_{i,j} + R_m \Big). \tag{7}$$

The solution of $(\mathcal{S})$ can be computed by iterating this process analogous to Gaussian elimination. This was the first proposed algorithm [11]; but Gaussian elimination on non-commutative semirings leads to exponential space requirements. We propose two alternative methods to compute and represent binding functions effectively. The first one improves on Gaussian elimination but keeps an exponential complexity; its theoretical interest is to capture the *relations between all induction variables* along a single path on the control automaton. If we only need to represent the computation of induction variables *separately* from each other, Section 6.2 presents a polynomial algorithm.

6.1 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 [49]. The neutral element of M_{rat} is

$$\mathbb{E} = \begin{bmatrix} \mathcal{E} & \ddots & \emptyset \\ \emptyset & & \mathcal{E} \end{bmatrix}. \tag{8}$$

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 11; 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): $(i_1,j_1) = (i_0a + j_0b, i_0c + j_0d)$. 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: $(i_*,j_*) = \big((i_0 + j_0d^*b)(a+cd^*b)^*, (j_0 + i_0a^*c)(d+ba^*c)^*\big)$. Writing $P = (a+cd^*b)^*$ and $Q = (d+ba^*c)^*$,

$$C^* = \begin{bmatrix} a & c \\ b & d \end{bmatrix}^* = \begin{bmatrix} P & d^*bP \\ a^*cQ & Q \end{bmatrix}. \tag{9}$$

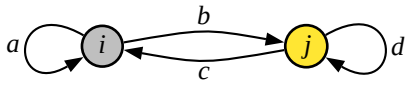


Figure 11: Computation of a matrix star

From (6), 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 (9) 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:

$$\forall \mathbb{S} \in M_{\text{rat}}, \overrightarrow{\Lambda_s} = \overrightarrow{\Lambda^d} \times \mathbb{S}. \tag{10}$$

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,

$$\overrightarrow{\Lambda^a} = \sum_{d \in \text{pred}(a)} \overrightarrow{\Lambda^d} \times \mathbb{S}_{da}, \mathbb{S}_{da} \in M_{\text{rat}}, \tag{11}$$

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 6 *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*

$$\overrightarrow{\Lambda^f} = \overrightarrow{\Lambda^0} \times \mathbb{L}, \tag{12}$$

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 5.

Because this method operates on regular expressions, it has a worst-case exponential complexity in the number of states and induction variables. However, this worst-case behavior is not likely on typical examples.

Application to the running example. We now give the statement matrices for the Toy example. With the three induction variables i, k and z, the binding vector after statement I, $\overrightarrow{\Lambda_I} = (\Lambda_{Ii}, \Lambda_{Ik}, \Lambda_{Iz})$ and $\mathbb{I}$ the statement matrix for I, we have:

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

with the following statement matrices:

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

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

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

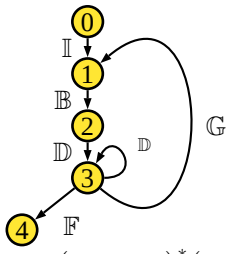
$$\text{statement } D: \quad \mathbb{D} = \begin{bmatrix} \emptyset & \emptyset & \emptyset \\ D|0 & D|0 & \emptyset \\ \emptyset & \emptyset & D|0 \end{bmatrix}$$

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 12 (all states are final).



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

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

Figure 12: Matrix automaton for Toy

6.2 Binding Transducer

We recall a few definitions and results about transducers [7]. A *rational transducer* is a finite-state automaton where each transition is labeled by a pair of *input* and *output* symbols, a symbol being a letter of the alphabet or the empty word.[6] 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.[7] 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 1

Given the control automaton and a monoid with n_v induction variables (including 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 (3.**c**).*
  - *case **c**.1: the transition runs from state P^d_{z} in P^d to the state P^a_{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_{y} in P^d to state P^a_{x} in P^a. The input label is s, the output label is the constant c.*

The binding transducer for `Toy` is shown in Figure 13. 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.

The binding transducer does not directly describe the binding functions. 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 7 *The binding transducer dedicated to an induction variable* x *recognizes the binding function for* x*.*

This result is a corollary of Theorem 5.

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 [3]. 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 [35].

Figure 14 summarizes some results about recursive programs we implemented in MOGUL. The last column of the table gives the number of states in the binding transducer. Since the first survey of instancewise analyses techniques [11], 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.

`Pascaline` is a small kernel to evaluate the binomial coefficients. `n-Queens` solves the classical problem to place n Queens on a $n \times n$ chessboard. `To_&_fro` is the recursive merge-sort alternating between two arrays. It is optimized in `To_&_fro+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 Applications of Instancewise Analysis

To illustrate the practical applications of the binding transducer, we revisit a simple program optimization that benefit from the computation of instancewise binding functions, then we outline the known applications and results.

[6] Pair of words leads to an equivalent definition.

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

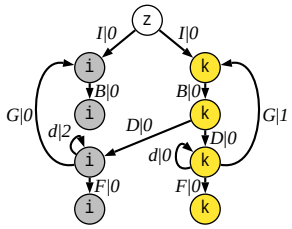


Figure 13: Binding transducer

Code name	Data	Lines	Refs	Loops	Calls	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
`Merge_sort_tree`	ternary tree	75	8	0	8	80
`To_&_fro`	1D array	115	12	0	19	164
`To_&_fro+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 14: Application to sample programs

8.1 Instancewise Dead-Code Elimination

Going back to the motivating examples in Section 1.1 (either code version), we call s the assignment to array `A` in the loop nest and t the read reference in procedure `line`. We assume the codes have been rewritten in MOGUL (the first version has a single induction variable addressing $\mathbb{Z}^2$ and the second one has two recursively swapped induction variables).

Let B_s and B_t denote the binding functions for the array references in s and t, respectively, and L^*_{ab} denote the set of all control words. The "chessboard" footprint, very hard to compute by statementwise means, corresponds to the rational set $B_t(L^*_{\mathrm{ab}})$. An intensional representation for this rational set can be computed, either as a finite-state automaton (in a straightforward transducer projection [7]), or as a $\mathbb{Z}$-polyhedron (e.g., through a Parikh mapping [43, 49]).

From this first result, one may automatically characterize the iterations of the loop nest which correspond to useless assignments to `A`: the (conservative) set of dead iterations is $B_s^{-1}(B_t(L^*_{\mathrm{ab}}))$. Once again, this turns out to be a classical operation on transducers and finite-state automata. To implement the actual optimization on the bounds and strides, a polyhedral characterization of the iteration domain can be deduced from the resulting automaton (because s is surrounded by a loop nest, not arbitrary recursive control) [45, 49].

8.2 State of the Art

Aggressive dead-code elimination is a very simple application of the instancewise framework. One may imagine many other extensions of scalar, loop and interprocedural optimizations, working natively on recursive programs. However, published results are still preliminary [12, 26, 5, 11, 3]: here is a short overview of the known applications of binding functions to the analysis of recursive programs.

- Instancewise dependence analysis for arrays [12, 11]. The relation between dependent instances is computed as a one-counter (context-free) transducer, or by a multi-counter transducer in the case of multi-dimensional arrays. In the multi-counter case, the characterization of dependences is undecidable in general, but approximations are possible.
- Instancewise reaching-definition analysis for arrays [12, 11] (a.k.a. array data-flow analysis [25, 39]). Compared to dependence analysis, kills of previous array assignments are taken into account. Due to the conservative assumptions about conditional guards (ignored in this paper), one may only exploit kill information based on structural properties of the program, i.e., exclusive branches and ancestry of control words in the call tree (whether an instance forcibly precedes another in the execution). This limitation seems rather strong, but it already subsumes the loop-nest case [11].
- Instancewise dependence and reaching-definition analysis for trees [11]. The relation between conflicting instances is a rational transducer, from the Elgot and Mezei theorem [20, 7]; the dependence relation requires an additional sequentiality constraint, which makes its characterization undecidable in general, but an approximation scheme based on synchronous transducers is available [44, 11]. The array and tree cases can be unified: [11] describes a technique to analyze nested trees and arrays in free partially-commutative monoids [50].
- Instancewise dependence test for trees [26, 5]. Instead of a relation between instances, these tests leverage on instancewise analysis to compute precise statementwise dependence information with unlimited context-sensitivity (not k-limited). Both [26] and [5] feature a semi-algorithm to solve the undecidable dependence problem; it is proven to terminate provided the approximation scheme of the previous technique is used (unpublished result).
- Instancewise dependence test for arrays [3]. This paper proves the decidability and NP-completeness of dependence testing based on binding transducers, in the case of arrays. An extension taking conditional guards into account is possible, provided the guards can be expressed as affine functions of some inductive variables lying in free-commutative monoids (unpublished result).

9 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 [3]. 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.

Acknowledgments. Jean-François Collard and Martin Griebl pioneered the field of instancewise analysis of recursive programs; many results presented in this paper are natural extensions of our previous work with Jean-François Collard. We are very grateful to Véronique Donzeau-Gouge, Catherine Dubois, Christine Eisenbeis and Matthieu Martel for their numerous contributions and comments.

References

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

[2] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan and Kaufman, 2002.

[3] 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.

[4] P. Amiranoff, A. Cohen, and P. Feautrier. Instancewise analysis. Technical Report 5117, INRIA Futurs, France, Feb. 2004. Research report.

[5] D. K. Arvind and T. A. Lewis. Dependency analysis of recursive data structures using automatic groups. In *11th Workshop on Languages and Compilers for Parallel Computing*, number 1656 in LNCS, pages 353–366, Chapel Hill, North Carolina, Aug. 1998. Springer-Verlag.

[6] R. Bagnara, E. Ricci, and E. Zaffanella. Precise widening operators for convex polyhedra. In *Int. Symp. on Static Analysis (SAS'03)*, LNCS, San Diego, CA, June 2003. Springer-Verlag.

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

[8] A. Bouajjani, J. Esparza, and T. Touilli. A generic approach to the static analysis of concurrent programs with procedures. In *ACM Symp. on Principles of Programming Languages (PoPL'03)*, Jan. 2003.

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

[10] S. Chong and R. Rugina. Static analysis of accessed regions in recursive data structures. In *Int. Symp. on Static Analysis (SAS'03)*, pages 463–482, San Diego, CA, June 2003.

[11] 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/%\home{acohen}/publications/thesis.ps.gz`.

[12] A. Cohen and J.-F. Collard. Instancewise reaching definition analysis for recursive programs using context-free transductions. In *Parallel Architectures and Compilation Techniques (PACT'98)*, pages 332–340, Paris, France, Oct. 1998. IEEE Computer Society Press.

[13] J.-F. Collard. *Reasoning About Program Transformations*. Springer-Verlag, 2002.

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

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

[16] 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, CA, Jan. 1977.

[17] 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.

[18] B. Creusillet. *Array Region Analyses and Applications*. PhD thesis, École Nationale Supérieure des Mines de Paris (ENSMP), Paris, France, Dec. 1996.

[19] 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, June 1994.

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

[21] 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.

[22] J. Esparza and J. Knoop. An automata-theoretic approach to interprocedural data-flow analysis. In *FOSSACS'99*, 1999.

[23] J. Esparza and A. Podelski. Efficient algorithms for pre* and post* on interprocedural parallel flow graphs. In *ACM Symp. on Principles of Programming Languages (PoPL'00)*, pages 1–11, 2000.

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

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

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

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

[28] M. P. Gerlek, E. Stoltz, and M. J. 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.

[29] 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 *ACM Symp. on Principles of Programming Languages (PoPL'96)*, pages 1–15, St. Petersburg Beach, Florida, Jan. 1996.

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

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

[32] 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, California, June 1992.

[33] N. D. Jones and S. S. Muchnick. A flexible approach to interprocedural data flow analysis of programs with recursive data structures. *ACM Press*, 1982.

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

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

[36] V. Loechner and D. Wilde. Parameterized polyhedra and their vertices. *Int. J. of Parallel Programming*, 25(6), Dec. 1997. `http://icps.u-strasbg.fr/PolyLib`.

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

[38] M. Martel. Improving the static analysis of loops by dynamic partitioning techniques. In *IEEE Workshop on Source Code Analysis and Manipulation*. IEEE Press, 2003.

[39] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Array dataflow analysis and its use in array privatization. In *20th ACM Symp. on Principles of Programming Languages*, pages 2–15, Charleston, South Carolina, Jan. 1993.

[40] M. Müller-Olm. Precise interprocedural analysis through linear algebra. In *ACM Symp. on Principles of Programming Languages (PoPL'04)*, Venice, Italy, Jan. 2004.

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

[42] C. Okasaki. Functional data structures. *Advanced Functional Programming*, pages 131–158, 1996.

[43] R. J. Parikh. On context-free languages. *J. of the ACM*, 13(4):570–581, 1966.

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

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

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

[47] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *ACM Symp. on Principles of Programming Languages (PoPL'95)*, San Francisco, CA, Jan. 1995.

[48] T. W. Reps, S. Schwoon, and S. Jha. Weighted pushdown systems and their application to interprocedural dataflow analysis. In *Int. Symp. on Static Analysis (SAS'03)*, pages 189–213, San Diego, CA, June 2003.

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

[50] G. Rozenberg and A. Salomaa, editors. *Handbook of Formal Languages*, volume 3: Beyond Words. Springer-Verlag, 1997.

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

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

[53] M. Sharir and A. Pnueli. *Program Flow Analysis: Theory and Applications*, chapter Two Approaches to Interprocedural Data Flow Analysis. Prenticce Hall, 1981.

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

[55] A. Venet. Nonuniform alias analysis of recursive data structures and arrays. In *Int. Symp. on Static Analysis (SAS'02)*, volume 2477 of *LNCS*, pages 36–51. Springer-Verlag, 2002.

[56] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.

A Appendix

A.1 The Trace Grammar

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 (*S*9), there is a production schema

$$C_{\texttt{id}} \quad ::= \quad \texttt{Label}\, B_{\texttt{id}}\, \overline{\texttt{Label}} \tag{13}$$

   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 (*S*11), there are four production schemas

$$L_s \quad ::= \quad \varepsilon \quad | \quad \texttt{Label}_{\texttt{e}}\, \texttt{Label}_{\texttt{p}}\, B_s\, O_s\, \overline{\texttt{Label}_{\texttt{p}}}\, \overline{\texttt{Label}_{\texttt{e}}} \tag{14}$$

$$O_s \quad ::= \quad \varepsilon \quad | \quad \texttt{Label}_{\texttt{i}}\, \texttt{Label}_{\texttt{p}}\, B_s\, O_s\, \overline{\texttt{Label}_{\texttt{p}}}\, \overline{\texttt{Label}_{\texttt{i}}} \tag{15}$$

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

3. For each conditional *s*, i.e., each derivation of production (*S*10), there are two productions schemas

$$I_s \quad ::= \quad \texttt{Label}\, T_s\, \overline{\texttt{Label}} \quad | \quad \texttt{Label}\, F_s\, \overline{\texttt{Label}} \tag{16}$$

   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 (*S*4) or (*S*5), there is a production schema

$$B_s \quad ::= \quad \texttt{Label}\, S_1\, \ldots\, S_n\, \overline{\texttt{Label}} \tag{17}$$

   where 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

$$S_s \quad ::= \quad \texttt{Label}\, \overline{\texttt{Label}} \tag{18}$$

   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. The set of traces of a program P —called the *trace language* of P —is the set of terminal sentences of G_P. 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 (13) to (18). 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* [37]. 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 Presburger arithmetic [45]. In this paper, we will interpret address computations in the theory of finitely-presented monoids; everything else will remain uninterpreted.

A.2 Trace Grammar and 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 (*S*9), there are two productions

$$C_{\texttt{id}} \quad ::= \quad \texttt{Label}\, B_{\texttt{id}} \quad | \quad \varepsilon$$

2. For each loop statement *s*, i.e., each derivation of production (*S*11), there are six productions

$$L_s \quad ::= \quad \texttt{Label}_{\texttt{e}}\, \texttt{Label}_{\texttt{p}}\, B_s \quad | \quad \texttt{Label}_{\texttt{e}}\, O_s \quad | \quad \varepsilon$$
$$O_s \quad ::= \quad \texttt{Label}_{\texttt{i}}\, \texttt{Label}_{\texttt{p}}\, B_s \quad | \quad \texttt{Label}_{\texttt{i}}\, O_s \quad | \quad \varepsilon$$

3. For each conditional *s*, i.e., each derivation of production (*S*10), there are three productions

$$I_s \quad ::= \quad \texttt{Label}\, T_s \quad | \quad \texttt{Label}\, F_s \quad | \quad \varepsilon$$

4. For each block *s* enclosing *n* statements, i.e., each derivation of (*S*4) or (*S*5), there are $n+1$ productions

$$B_s \quad ::= \quad \texttt{Label}\, S_1 \quad | \quad \cdots \quad | \quad \texttt{Label}\, S_n \quad | \quad \varepsilon$$

5. For each elementary statement *s*,

$$S_s \quad ::= \quad \texttt{Label} \quad | \quad \varepsilon$$

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

The control words grammar grammar above is right linear,[8] hence its generated language is rational.

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

Free monoid.
$G = \{\texttt{right}, \texttt{left}\}$, $\equiv$ is the identity, $\cdot$ 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 [21, 30].

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.

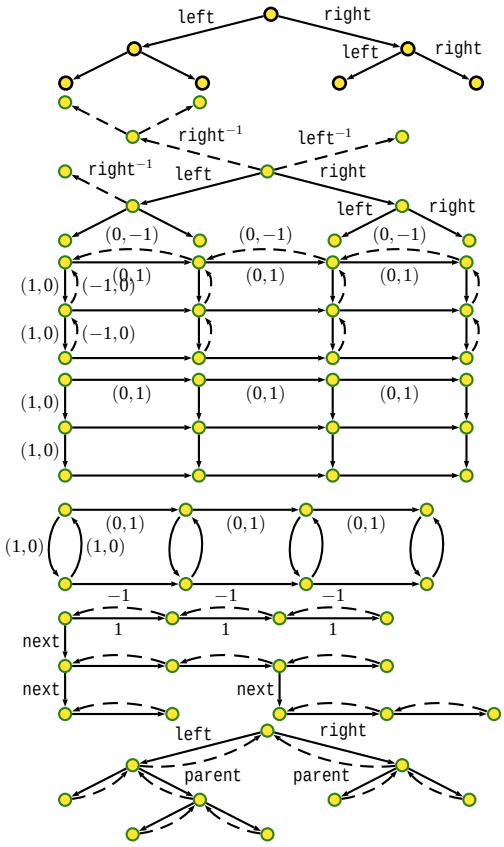


Figure 15: Monoid-addressable structures

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.

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.

Conversely, we can show that the control words grammar only generates control words, assuming the trace grammar satisfies a termination criterion, defined through the concept of *unlooping grammar* [54] (see the section on *reduced grammars*). More details and a simple decision algorithm can be found in [4].

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

Theorem 8 *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 G_P is unlooping.*

We assume the program satisfies Theorem 8.

A.3 Proof of Lemma 1

For completeness, we formally define $\lhd_{\text{lab}}$ from the MoGuL grammar. Given s_1 and s_2 two labels in L_{ab}, $s_1 <_{\text{lab}} s_2$ if and only if

- there is a production generated by (17) in the trace grammar, such as s_1 is the label of S_i and s_2 is the label of S_j, with $1 \le i < j \le n$;
- or there is a production generated by (14) or (15) such as s_1 is the label of B_s and s_2 is the label of O_s.

The proof of Lemma 1 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, $q \le n$. 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

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

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

A.4 Free Partially-Commutative Monoids

Figure 15 surveys representative monoid-addressable data structures fitting into our framework.