Beyond Iteration Vectors: Instancewise Relational Abstract Domains

Pierre Amiranoff, Albert Cohen, Paul Feautrier

Abstract. We introduce a formalism to reason about program properties at an infinite number of runtime control points, called instances. Infinite sets of instances are represented by rational languages. This framework gives a formal foundation to the well known concept of iteration vectors, extending it to recursive programs with any structured control flow (nested loops and recursive calls). We also 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. Applications include dependence and region analysis for array and tree algorithms, array expansion, and automatic parallelization of recursive programs.

1 Introduction

Most compiler techniques reflect the natural inductive structure of programming languages semantics and machine models. This approach facilitates formal reasoning and proofs, enhances modularity (composition), and leads to fast and simple algorithms.

Yet more and more advanced program analysis and transformation problems benefit from decoupling the construction of static abstractions from the local, inductive semantics of the programs. These abstractions, sometimes called *constraint-based* [31], are custom to a specific class of programs or compilation problem. They are engineered to exhibit statically tractable algebraic properties, e.g., closed mathematical forms amenable to formal (symbolic) computations, typically based on Presburger arithmetic or decidable automata-theoretic classes.

To make this duality more concrete, consider the constant propagation optimization [1] which amounts to computing a property of a variable $\mathtt{v}$ at a statement s, asking whether $\mathtt{v}$ has some value v before s executes. It is quite natural to formalize constant propagation in an abstract data-flow or interpretation setting. But let us now consider another static analysis problem that may be seen as an extension of constant propagation: *induction variable recognition* [1, 22] captures the value of some variable $\mathtt{v}$ at a statement s as a function $f_{\mathtt{v}}$ of the number of times s has been executed. In other words, it captures $\mathtt{v}$ 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 the other induction variables using analyses of linear constraints [9], but such syntactically bound approaches will not easily cope with the calculation of function $f_{\mathtt{v}}$ itself.

*In the following, we will qualify as **instancewise** any compilation method operating on abstract, finitely presented, **relational domains** bewteen the infinite set of **runtime control points** and any semantical domain of interest. Other analyses will be called **statementwise**.*

1.1 Statementwise Analysis

We use the term *statementwise* to refer to the classical type systems, data-flow analysis and abstract interpretation frameworks, that aim for the computation of a *collecting semantics* of a *finite set of collection 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* [8] 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* [4] integrates partitioning into the analysis itself. Control points can be extended with *call strings* (abstract call stacks) and *timestamps*, but ultimately rely on *k-limiting* [42, 26] or *summarization* heuristics [37] to achieve convergence. Although unbounded lattices have long been used to capture abstract properties [9, 12]), few works considered the computation of data-flow facts using an *unbounded set of control points*. Even the most advanced non-uniform (a.k.a. elementwise) analyses of arrays or heap structures formalized in abstract interpretation, including the approaches by Deutsch [12, 11] and Venet [43, 44], although they compute individual properties over an infinite set of heap locations, they do map these properties to a *finite* set of program points only. The seminal paper by Esparza and Knoop [15] is closer to our work since it promises a non-intepretative family of static analysis (semi-)algorithms. Follow-up works building on model-checking of push-down systems or on various path-sensitive abstractions to extend precision and context sensitivity [16, 38, 25]. Yet all these methods ultimately result in the computation of data-flow properties attached to a *finite number of control points*.

1.2 Instancewise Analysis

On the other hand, ad-hoc approaches to static analysis are able to compute program properties as *functions defined on an infinite (or unbounded) number of run-time control points*. Historically, most works derived from loop-restructuring compiler efforts [45] aiming at a large spectrum of optimizations: vectorization, instruction-level, thread-level or data parallelism, scheduling and mapping for automatic parallelization, locality optimization, and many others [34, 45]. In the associated static analyses, the retrieval of abstract properties is often iterationless [45, 46, 35] and resort to operation research algorithms thoroughly alien to program interpretation [19]. The work of Creusillet [10] is one of the rare instancewise analyses to resort to abstract interpretation, the reason lying in the interprocedural nature of the analysis. The so-called *polyhedral model* encompasses most works on analysis and transformation of the (Turing-incomplete) class of *static-control programs* [17, 34], 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 loop counters. In general, the result of the analysis is a mapping from the infinite set of iteration vectors (the run-time control points) to an arbitrary (analysis-specific) vector space (e.g., dependence vector). 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 [17]. Iteration vectors are quite different from time-stamps in control point partitioning techniques [4, 44]: they are multidimensional, lexicographically ordered, *unbounded*, and constrained by Presburger formula [36].

First Contribution. We introduce a general abstraction of runtime control points that uncompasses most ad-hoc formalisms for the fine grain analysis of loop nests and arrays in sequential procedural languages. Within this framework, one may *define, abstract and compute* program properties as *functions* of an *infinite* set of runtime control points. The concept of *instance* allows to reason about these runtime points across multiple executions of the program; it is the generalization of an iteration vector to general recursion. Rational languages finitely represent infinite set of instances, and instancewise properties may be captured by rational relations [3]. This paper goes far beyond our previous attempts to extend iteration vectors to recursive programs, for the analysis of arrays [6, 5, 7, 2] or recursive data structures [20, 7, 5].

Second Contribution. Building on the instancewise framework, we extend the concept of *induction variables* to arbitrary recursive programs. This extension demonstrates the ability to characterize properties of programs as functions from an infinite set of runtime control points, beyond the restricted domain of Fortran loop nests. Technically, 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 an polynomial algorithm to compute each induction variable as 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* [3]. This binding function will give an *exact* result for valid traces.

Organization of the Paper. Section 2 defines the instancewise model. Section 3 extends induction variables to recursive control and data structures. Section 4 states the existence of rational binding functions. Section 5 addresses the computation of binding functions as rational transducers. Section 6 surveys theoretical results that derive from this work.

2 From Traces to Instances

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` (there is no simple way to remove the recursion).

```
int A[100];
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);
      }
  }
}
int main() {
  Toy(99, 0);
}
```

Fig. 1. Program `Toy` in C

```
   structure Monoid_int A;
A  void 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  int main() {
I    Toy(20, 0);
   }
```

Fig. 2. Program `Toy` in MoGuL

To focus on the core concepts, we introduce MoGuL, a toy 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 [27, 28, 21] or shape analyses [23, 41].

Figure 3 is a simplified version of the MoGuL C-like syntax, focusing on the control structures. Italic non-terminals are defined elsewhere: *`elementary_statement`* covers the usual atomic statements, including assignments, input/output statements, void statements,

```
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 ';' LABEL ':' predicate ';'
                          LABEL ':' translation_list ')' block       (S11)
                 |block                                              (S12)
```

Fig. 3. Simplified MoGuL syntax (control structures)

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; `block` collects a sequence of statements, possibly defining some induction variables. Every statement is labeled.

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 $\cdot$. Further explanations about MoGuL data structures and induction variables are deferred to Section 3.

Every statement is seen as a function application; therefore the execution stack alphabet holds every statement label. Moreover, each statement is provided with a second *return* label to identify the implicit termination of the statement: if ℓ is a statement label, ℓ corresponds to the beginning of the execution of a statement and $\overline{\ell}$ indicates its completion. Regarding the execution stack, ℓ pushes ℓ while $\overline{\ell}$ pops ℓ. An additional state, called *return state*, is associated to the completion of each statement. The result is called the *pushdown trace automaton* and the recognized words are the *execution traces*. A *trace prefix* is the trace of a partial execution, given by a prefix of a complete trace, it matches the intuitive notion of a runtime control point.

Figure 4 presents the trace pushdown automaton of the `Toy` program. We exhibit here a prefix of a valid trace: $IABCD\delta EF\overline{F}GABCD\delta EF\overline{F}GAB\overline{BAGE}\delta d\delta EF$.

For clarity of exposure and without loss of precision, the following examples will use a simplified representation of the trace pushdown automaton, omitting return states, except for calls to `Toy`, states associated to `block` statements and to loop predicates. Now, the previous trace prefix reduces to: $IBDF\overline{F}GBDF\overline{F}GB\overline{BG}dF$. We will use this simplified representation of traces in the following.

The purpose of this paper is to construct a finite-state representation for the (infinite) set of trace prefixes of `Toy`*, then compute a statically tractable characterization of the precise element of* `A` *accessed by every given trace prefix(again an infinite relational abstract domain).*

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

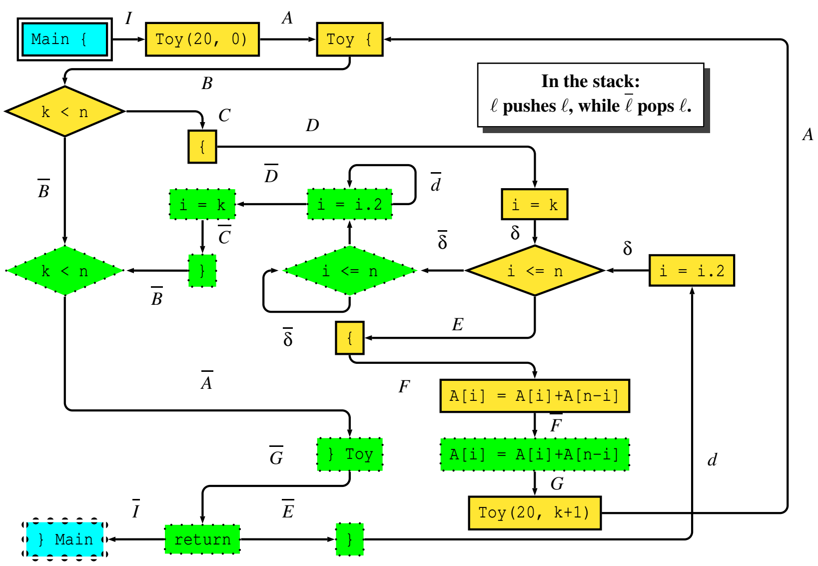


Fig. 4. Pushdown Trace Automaton

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 [39], the language of control words is rational. It is easy to build a finite-state automaton accepting the rational language of control words. We call the latter the *control automaton*.

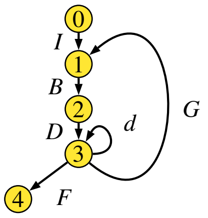
Figure 5 shows the control automaton for `Toy`; the control word language is $I+IB+IBD(d+GBD)^*(\varepsilon+F+G+GB)$.

The transformation from trace prefixes 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.

- In the pushdown trace automaton, a sequence of successive statements 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 6. 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.

The trace language is a Dyck language [3], i.e., a hierarchical parenthesis language. We define the *slimming congruence* generated by $\ell\overline{\ell} \equiv \varepsilon$, for all $\ell \in L_{\mathrm{ab}}$. This definition induces a rewriting rule over L_{ab}^*, obviously confluent. This rule is the direct transposition of the execution stack behavior. Applying it to any trace prefix p we can associate a minimal word w.

Definition 2 (Slimming Function). *The* slimming function *maps each trace prefix to the shortest element in the class of p modulo the slimming congruence.*

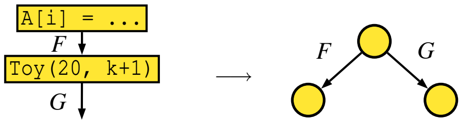


All states are final.

A few control words:

IBDdF,
IBDGBDF,
IBDGBDdG.

Fig. 5. Example Control Automaton



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

Fig. 6. Construction of the Control Automaton

Theorem 1. *The set of control words is the quotient set of all prefixes of all traces modulo the restricted Dyck congruence, and the slimming function is the canonical projection of trace prefixes over their control words.*

The following table illustrates the slimming function on a few trace prefixes.

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

The slimming function extends Harrison's NET function; we coined the term control word in favor of *procedure string* [42, 26, 30] to emphasize the unboundedness of the relation over stack configurations we ought to statically reason about.

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:[1] their control words are the same.

The demonstration of the following results and additional relations between trace prefixes and control words are provided in the appendix.

We now come to the formal definition of an instance:

Definition 3 (Instance). *Considering the set of all traces of a* MoGuL *program, an* instance *is a class of trace prefixes modulo the slimming congruence.*

Thanks to Theorem 1, instances are in bijection with control words. In the following, we will refer to instances or control words interchangeably, without naming a particular trace prefix representative.

From Theorem 1 again, it is fundamental to notice that an instance has a consistent definition across all possible executions; this allows to *collect the semantics* of all trace prefixes congruent to a given control word.

The following results states that an instance is also the finest possible way to collect the semantics over multiple executions.

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

Theorem 2 ensures that control words characterize runtime control points in a more compact way than trace prefixes. It even guarantees that computing a relational domain at each instance is no coarser than dynamically evaluating the concrete property at each runtime control point.

[1] I.e., after completing a hammock, the two sub-traces being associated to exclusive branches.

3 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:

- they do not support destructive updates (deletion of nodes and non-leaf insertions);[2]
- they are addressed 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* [32].

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.

3.1 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).

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 4 (Abstract Location). *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).

Below are listed some practical examples of monoid-based data structures.

3.2 Induction Variables

Traditionally, induction variables are scalar variables within loop nests with a tight relationship with the surrounding loop counters [1, 22]. 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 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 expressed as *an affine function of the surrounding loop counters*. MOGUL extensions are twofold:

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

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

Free group.
$G = \{$`right`$,$`left`$,$`right`$^{-1},$`left`$^{-1}\}$, $\equiv$ is the inversion of `left` and `right` (without commutation): Cayley graphs [14, 24].

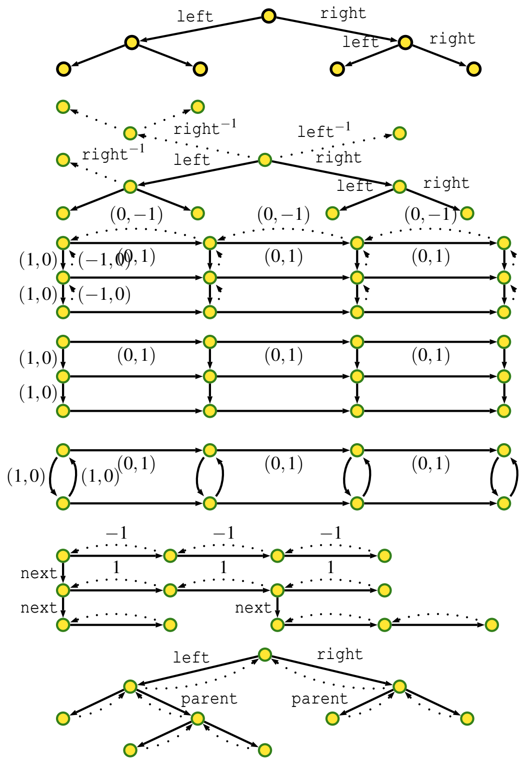
Free commutative group.
$G = \{(0,1),(1,0),(0,-1),(-1,0)\}$, $\equiv$ is the vector inversion and commutation, $\cdot$ is vector addition: a two-dimensional array.

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.
$G = \{$`next`$, 1, -1\}$, $\equiv$ is the inversion and commutation of 1: nested trees, lists and arrays.

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



- induction variables can be valued in any monoid;
- their evolution can be captured across loops and recursive calls.

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

Definition 5 (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 ·` 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 81/2 [24]. 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 5. 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 ·` 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.

4 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*.

4.1 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 3. *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 6 (Binding Function). *A* binding for `x` *is a pair* (w, v)*, where w is a control word and v the value of* `x` *at the instance w.*

Λ_{x} *denotes the* binding function for `x`*, mapping control words to the corresponding value of* `x`.

4.2 Bilabels

We now describe the mathematical framework to compute binding functions.

Definition 7 (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.

$B = L_{\text{ab}}^* \times M_{\text{loc}}$ 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 4.3.

Definition 8. *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 [3]. 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 9. *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 [3]. The set of rational subsets of M is a sub-semiring of the latter [3]; 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 $(s|c)$.

4.3 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. 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, $\Lambda_{\mathtt{x}}^n$, 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 the initial state 0 and for any induction variable `x`,

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

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.

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

2. $\Lambda_{\mathtt{z}}^{n}$ denotes the set defined by
$$\Lambda_{\mathtt{z}}^{n} = \bigcup_{w \in \mathcal{L}_n} (w|\varepsilon_m). \tag{3}$$
$\Lambda_{\mathtt{z}}^{n}$ 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 $c_{s\mathtt{x}}$ (case **c**.1 of Definition 5), and calling d and a the respective departure and arrival states of s in the control automaton,
$$\Lambda_{\mathtt{x}}^{a} \supseteq \Lambda_{\mathtt{z}}^{d} \bullet (s|c_{s\mathtt{x}}). \tag{4}$$
Since $\Lambda_{z}^{d} \bullet (s|c_{s\mathtt{x}}) = \bigcup_{w \in \mathcal{L}_d} (ws|c_{s\mathtt{x}})$, (4) means: if $w \in \mathcal{L}_d$ is a control word, ws is also a control word and its binding for x is $(ws|c_{s\mathtt{x}})$.
The control automaton automaton of `Toy` has 5 states. For the case **c**.1 of Definition 5,
statement I : `k = 0`, and (4) yields $\Lambda_{\mathtt{k}}^{1} \supseteq \Lambda_{\mathtt{z}}^{0} \bullet (I|0)$.
3. For each statement s defining an induction variable x to $\mathtt{y} \cdot c$ (case **c**.2 of Definition 5), and d and a the respective departure and arrival states of s,
$$\Lambda_{\mathtt{x}}^{a} \supseteq \Lambda_{\mathtt{x}}^{d} \bullet (s|c_{s\mathtt{x}}). \tag{5}$$
To complete the system, we add for every induction variable x unchanged by s a set of equations in the form (5), where $c_{s\mathtt{x}} = \varepsilon_m$.
E.g., for case **c**.2 of Definition 5,
statement G : `k = k · 1` statement d : `i = i · 2` statement D : `i = k`
and (5) yields

$\Lambda_{\mathtt{i}}^{1} \supseteq \Lambda_{\mathtt{i}}^{3} \bullet (G\|0)$	$\Lambda_{\mathtt{i}}^{3} \supseteq \Lambda_{\mathtt{k}}^{2} \bullet (D\|0)$	$\Lambda_{\mathtt{i}}^{4} \supseteq \Lambda_{\mathtt{i}}^{3} \bullet (F\|0)$	$\Lambda_{\mathtt{z}}^{2} \supseteq \Lambda_{\mathtt{z}}^{1} \bullet (B\|0)$
$\Lambda_{\mathtt{k}}^{1} \supseteq \Lambda_{\mathtt{k}}^{3} \bullet (G\|1)$	$\Lambda_{\mathtt{i}}^{3} \supseteq \Lambda_{\mathtt{i}}^{3} \bullet (d\|2)$	$\Lambda_{\mathtt{k}}^{4} \supseteq \Lambda_{\mathtt{k}}^{3} \bullet (F\|0)$	$\Lambda_{\mathtt{z}}^{3} \supseteq \Lambda_{\mathtt{z}}^{2} \bullet (D\|0)$
$\Lambda_{\mathtt{i}}^{2} \supseteq \Lambda_{\mathtt{i}}^{1} \bullet (B\|0)$	$\Lambda_{\mathtt{k}}^{3} \supseteq \Lambda_{\mathtt{k}}^{2} \bullet (D\|0)$	$\Lambda_{\mathtt{z}}^{1} \supseteq \Lambda_{\mathtt{z}}^{0} \bullet (I\|0)$	$\Lambda_{\mathtt{z}}^{3} \supseteq \Lambda_{\mathtt{z}}^{2} \bullet (d\|0)$
$\Lambda_{\mathtt{k}}^{2} \supseteq \Lambda_{\mathtt{k}}^{1} \bullet (B\|0)$	$\Lambda_{\mathtt{k}}^{3} \supseteq \Lambda_{\mathtt{k}}^{2} \bullet (d\|0)$	$\Lambda_{\mathtt{z}}^{1} \supseteq \Lambda_{\mathtt{z}}^{3} \bullet (G\|0)$	$\Lambda_{\mathtt{z}}^{4} \supseteq \Lambda_{\mathtt{z}}^{3} \bullet (F\|0)$

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.[3]

`Toy` yields the system

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

Let Λ be the set of unknowns for $(\mathcal{S})$, i.e., the set of $\Lambda_{\mathtt{x}}^{n}$ 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) [39]. 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
$$\forall i \in \{1, \ldots, m\}, X_i = \sum_{j=1}^{m} X_j \bullet C_{i,j} + R_i, \tag{6}$$

[3] Some unknown sets correspond to unbound variables at the node of interest, they are useless.

where R_i results from the terms $\Lambda_{\mathtt{x}}^{0} = \mathcal{E}$ 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 [39]; moreover, this solution can be characterized by a *rational expression* for each unknown set in Λ.

Definition 10 (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.*

We may conclude that the solution of $(\mathcal{S})$ is a characterization of each unknown set X_i in Λ as a rational function.

Lemma 1. *For any induction variable* $\mathtt{x}$ *and node n in the control automaton, the binding function for* $\mathtt{x}$ *restricted to $\mathcal{L}_n$ $\Lambda_{\mathtt{x}}^{n}$ is a rational function.*

Theorem 4. *For any induction variable* $\mathtt{x}$*, the binding function* $\Lambda_{\mathtt{x}}$ *is a rational function.*

This is a corollary of Lemma 1, since the functions $\Lambda_{\mathtt{x}}^{n}$ are defined on disjoint subsets of control words, partitioned according to the suffix n.

Properties of rational relations and functions are similar to those of rational languages [3]: 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 [33] and [5]. Since most of these properties are associated with polynomial algorithms, binding functions can be used in many analyses, see [6, 20, 5, 2] for our previous and ongoing applications to the automatic parallelization of recursive programs.

5 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 [5]; but Gaussian elimination on non-commutative semirings is an exponential process. We present a polynomial algorithm to compute the binding function of each induction variable as a rational transducer.

A *rational transducer* is a finite-state automaton where each transition is labeled by a pair of *input* and *output* symbols (borrowing from Definition 7), a symbol being a letter of the alphabet or the empty word [3].[4]

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.[5]

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 representation for the binding function of an induction variable is called the *binding transducer*.

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

[5] 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 $(xy|z)$.

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 (5.**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.*

The binding transducer for `Toy` is shown in Figure 7. 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.

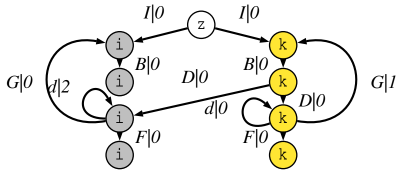


Fig. 7. Binding Transducer for `Toy`

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

This result is a corollary of Theorem 4.

From this result, it is possible (yet not mandatory for most analysis purposes) to capture all induction variables and their relations into a single object; this may translate into computing product-states of many transducers, which in the worst case is exponential in the number of induction variables [2].

The construction of the binding transducer is fully implemented in OCaml and has been experimented on a dozen of recursive kernels. 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].

6 Applications of Instancewise Analysis

Instancewise analyses are pervasive when dealing with loop nest programs [45, 7]. This section is a short overview of the known applications of instancewise binding functions to the analysis of *recursive* programs. Most applications derive from standard algorithms and decision procedures on rational and push-down transducters [3] which are similar to those of finite-state and push-down automata — with the notable exception of intersection and complementation. In some cases, one must resort to approximation schemes [5] and semi-algorithms [20].

- Instancewise dependence analysis for arrays [6, 5]. 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, only approximate may/must dependences are available. This technique applies to the `Toy` example.
- Instancewise reaching-definition analysis for arrays [6, 5] (a.k.a. array data-flow analysis [18, 29]). 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 [5] and the `Toy` program.
- Instancewise dependence test for arrays [2]. Amiranoff's thesis 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 available, provided the guards can be expressed as affine functions of some inductive variables lying in free-commutative monoids (unpublished result). This extension defines conditions for the exactness of the dependence test (i.e., the absence of approximation) that strictly generalize the case of static-control loop nests; e.g., the exact dependence test can be applied to a complex control-flow of the `Toy` example.
- Instancewise dependence and reaching-definition analysis for trees [5]. The relation between conflicting instances is a rational transducer, from the Elgot and Mezei theorem [13, 3]; 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 [33, 5]. The array and tree cases can be unified: [5] describes a technique to analyze nested trees and arrays in free partially-commutative monoids [40].
- Instancewise dependence test for trees [20]. Instead of a relation between instances, this test leverages on instancewise analysis to compute precise statementwise dependence information with unlimited context-sensitivity (not k-limited). This technique features a semi-algorithm to solve the undecidable dependence problem, and the semi-algorithm is proven to terminate provided the approximation scheme of the previous technique is used (unpublished result).

7 Conclusion and Perspectives

Instancewise compilation decouples the static properties and analyses from the inductive presentation of program semantics, departing from interpretative algorithms to evaluate semantic program properties as relational abstract domains over infinite sets 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 [2]. 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. *An Automata-Theoretic Modelization of Instancewise Program Analysis: Transducers as mappings from Instances to Memory Locations*. PhD thesis, CNAM, Paris, Dec. 2004.
3. J. Berstel. *Transductions and Context-Free Languages*. Teubner, Stuttgart, Germany, 1979.
4. F. Bourdoncle. Abstract interpretation by dynamic partitioning. *J. of Functional Programming*, 2(4):407–423, 1992.
5. A. Cohen. *Program Analysis and Transformation: from the Polytope Model to Formal Languages*. PhD Thesis, Université de Versailles, France, Dec. 1999.
6. 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.
7. J.-F. Collard. *Reasoning About Program Transformations*. Springer-Verlag, 2002.
8. P. Cousot. *Semantic foundations of programs analysis*. Prentice-Hall, 1981.
9. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5thACM Symp. on Principles of Programming Languages*, pages 84–96, Jan. 1978.
10. B. Creusillet. *Array Region Analyses and Applications*. PhD thesis, École Nationale Supérieure des Mines de Paris (ENSMP), France, Dec. 1996.
11. A. Deutsch. *Operational Models of Programming Languages and Representations of Relations on Regular Languages with Application to the Static Determination of Dynamic Aliasing Properties of Data*. PhD thesis, École Polytechnique, France, Apr. 1992.
12. 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.
13. C. C. Elgot and J. E. Mezei. On relations defined by generalized finite automata. *IBM J. of Research and Development*, pages 45–68, 1965.
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. J. Esparza and J. Knoop. An automata-theoretic approach to interprocedural data-flow analysis. In *FOSSACS'99*, 1999.
16. 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.
17. P. Feautrier. Array expansion. In *ACM Intl. Conf. on Supercomputing*, pages 429–441, St. Malo, France, July 1988.
18. P. Feautrier. Dataflow analysis of scalar and array references. *Intl. J. of Parallel Programming*, 20(1):23–53, Feb. 1991.
19. P. Feautrier. Some efficient solutions to the affine scheduling problem, part II, multidimensional time. *Intl. J. of Parallel Programming*, 21(6):389–420, Dec. 1992. See also Part I, one dimensional time, 21(5):315–348.
20. P. Feautrier. A parallelization framework for recursive tree programs. In *EuroPar'98*, LNCS, Southampton, UK, Sept. 1998. Springer-Verlag.
21. 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.
22. 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.
23. 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.
24. 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 81/2, a Declarative Data-Parallel Language, RR 1012, Laboratoire de Recherche en Informatique, Université Paris Sud 11, France, 1995".
25. H. Hampapuram, Y. Yang, and M. Das. Symbolic path simulation in path-sensitive dataflow analysis. In *Proc. of the ACM Workshop on Program Analysis for Software Tools and Engineering (PASTE'05)*, pages 52–58, Lisbon, Portugal, Sept. 2005.
26. W. L. Harrison. The interprocedural analysis and automatic parallelisation of Scheme programs. *Lisp and Symbolic Computation*, 2(3):176–396, Oct. 1989.
27. 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, June 1992.
28. 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.
29. D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Array dataflow analysis and its use in array privatization. In *20^{th}ACM Symp. on Principles of Programming Languages*, pages 2–15, Charleston, South Carolina, Jan. 1993.
30. M. Might and O. Shivers. Environment analysis via delta cfa. In *ACM Symp. on Principles of Programming Languages (PoPL'06)*, pages 127–140, Charleston, South Carolina, Jan. 2006.
31. F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.
32. C. Okasaki. Functional data structures. *Advanced Functional Programming*, pages 131–158, 1996.

33. M. Pelletier and J. Sakarovitch. On the representation of finite deterministic 2-tape automata. *Theoretical Computer Science*, 225(1-2):1–63, 1999.
34. G.-R. Perrin and A. Darte, editors. *The Data Parallel Programming Model*. Number 1132 in LNCS. Springer-Verlag, 1996.
35. S. Pop, A. Cohen, and G.-A. Silber. Induction variable analysis with delayed abstractions. In *Intl. Conf. on High Performance Embedded Architectures and Compilers (HiPEAC'05)*, number 3793 in LNCS, pages 218–232, Barcelona, Spain, Nov. 2005. Springer-Verlag.
36. W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):27–47, Aug. 1992.
37. 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.
38. 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.
39. G. Rozenberg and A. Salomaa, editors. *Handbook of Formal Languages*, volume 1: Word Language Grammar. Springer-Verlag, 1997.
40. G. Rozenberg and A. Salomaa, editors. *Handbook of Formal Languages*, volume 3: Beyond Words. Springer-Verlag, 1997.
41. 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.
42. M. Sharir and A. Pnueli. *Program Flow Analysis: Theory and Applications*, chapter Two Approaches to Interprocedural Data Flow Analysis. Prenticce Hall, 1981.
43. 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.
44. A. Venet. A scalable nonuniform pointer analysis for embedded programs. In *Int. Symp. on Static Analysis (SAS'04)*, volume 3148 of *LNCS*, pages 149–164, Verona, Italy, 2004. Springer-Verlag.
45. M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.
46. P. Wu, A. Cohen, D. Padua, and J. Hoeflinger. Monotonic evolution: an alternative to induction variable substitution for dependence analysis. In *ACM Intl. Conf. on Supercomputing (ICS'01)*, Sorrento, Italy, June 2001.