Instance-wise Reaching Definition Analysis for Recursive Programs using Context-free Transductions

Albert Cohen and Jean-François Collard
PR*i*SM, University of Versailles
45 avenue des États-Unis, 78035 Versailles, France
Albert.Cohen@prism.uvsq.fr, Jean-François.Collard@prism.uvsq.fr

Abstract

Automatic parallelization of recursive programs is still an open problem today, lacking suitable and precise static analyses. We present a novel reaching definition framework based on context-free transductions. The technique achieves a global and precise description of the data flow and discovers important semantic properties of programs. Taking the example of a real-world non-derecursivable program, we show the need for a reaching definition analysis able to handle run-time instances of statements separately. A running example sketches our parallelization scheme, and presents our reaching definition analysis. Future fruitful research, at the crossroad of program analysis and formal language theory, is also hinted to.

Keywords: Reaching Definition Analysis, Recursive Programs, Context-free Languages, Push-down Transducers.

1. Introduction

This paper tackles the reaching definition problem for recursive programs. There are several, related but different, analyses of memory accesses. For instance, *alias analyses* compute pairs of conflicting memory accesses, and *dependence analyses* express conflicts with respect to program statements (syntactic lines). Our purpose goes beyond: For a given use (a read) of a memory cell, we are looking for the identity of the write that defined the value currently associated to the memory cell. Reaching definition analysis (RDA) is thus a special case of *data-flow analysis* [2].

Let us immediately stress a very important point. Classical RDAs tell, for a given read statement, whether a write statement may be the reaching definition. In other words, they tell whether there is a definition-free path connecting a definition site to a use site. Our RDA, however, identifies which *run-time instance* of some write statement defined the value used by a given *run-time instance*. We stress this difference in calling our RDA: "*Instance-wise* reaching definition analysis" (IRDA) [8]. To describe the *relation* between write instances and read instances, we first *name* all these instances unambiguously, then find a mapping from names of reads to names of writes.

Array data flow analyses [12, 18] are IRDAs for loop nests with arrays. Applications are numerous, especially in automatic parallelization of imperative programs: Array expansion, conversion to single assignment form, etc. However, most reaching definition analyses focuses on loop nests, and *recursive* programs have received little interest. The contributions of this article are:

- Formalizing instance-wise reaching definitions in recursive programs. A suitable labeling scheme for run-time instances of statements is introduced.
- Showing the use of context-free transductions to adequately describe reaching definitions in recursive programs, at the run-time instance level.
- Advocating for the development of complex transformations techniques for recursive programs, and for interesting applications of formal language theory to program analysis.

Of course, the idea of using formal languages for program analysis has been around for years. However, to the best of our knowledge, this paper is the first to use formal languages in an *instance-wise* reaching definition analysis.

The paper is organized as follows: Section 2 discusses related work. Section 3 presents our running example and show the use of reaching definitions in parallelization frameworks. Then, still studying the motivating example, we describe in Section 4 the IRDA we want to design. Section 5 formalizes IRDA in a transduction-based model. To use this transduction in a compiler we also need to construct the corresponding push-down transducer. This is done in Section 6. The method is applied in Section 7 to build the transducer for the motivating example. Section 8 comments on our analysis and describes future work.

2. Related Work

Many types of static analyses on data have been proposed. *Points-to analyses* compute a store model using abstract locations [5]. Jensen et al. [16] also developed an access analysis where a first-order logic serves as a store model (or, alternatively, a second-order one for sets of cells). Adapting the respective store models—tailored for pointer accesses—to array accesses with affine subscripts seems tedious. *Alias analyses* compute pairs of memory access paths that are, or may be, aliased [11, 17]. *Data dependences* describe conflicts between memory accesses in computing pairs of *dependent* statements (or run-time instances). Some of these analyses do handle arrays. They are yet basically different from our reaching definition analysis since they do not give the *identity of the last defining write*.

Most RDA abstract the effects of a program by sets of flow equations; These equations are then solved in general by iterative schemes [2, 9]. They often require that a fixpoint can be reached in a finite (and hopefully small) number of iterations. These methods proved to be robust in the case of intricate control flow. Some analyses explicitly handle recursive programs [3]. However, they do not identify the run-time instance of a definition site, i.e. the last write in a given memory cell.

Analyses closer to our work were proposed in [7, 12, 18, 20], because they give the identity of definition instances and because they precisely handle array accesses thanks to symbolic simplifiers [12, 19]. However, none of these analyses was developed for recursive programs. Our framework subsumes these analyses because loop nests can be considered as nested recursive procedures. Notice that the opposite does not hold: Recursive programs *cannot* in general be written as a nest of loops (without explicit stack handling).

3. The Motivation: Automatic Parallelization

Let us consider a motivating example, as shown in Figure 1, accessing an array `A`. (Programs are written in C but the method applies to any imperative language.) This is an example of recursive program for which no simple non-recursive version exists: Classical derecursivation algorithms will not work since recursive calls and loop iterations are mixed [1]. This program computes all possible RAM configurations for a computer, given the size of the memory chips and the number of slots available. This program originally appeared in Java on Sun's Web-site.

[1] Writing the outer loop recursively would make appear two mutually recursive procedures.

```
        int A[N+1];

        void FillSlots (int n) {
A/a       for (i = 0; i <= M; i++) {
s           A[n] = ···;
I           if (n == 0)
T           { /* then */
B/b           for (j = 0; j <= N; j++)
r               ··· = ··· A[j] ···;
E           } else
p             FillSlots (n-1);
          }
        }

        int main () {
          FillSlots (N);
        }
```

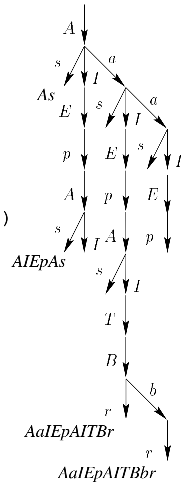

Figure 1. Procedure `FillSlots` and call tree

To expose as much parallelism as possible, our intent is to expand the data structures (here, arrays) so as to get rid of as many output, anti- and true dependences as possible. In the extreme expansion case, we would like to convert the program into *single assignment form*. This program transformation is very similar to *static single assignment* form (SSA) [10] or array expansion [12]. In single assignment form, all output dependences, anti-dependences, and true dependences due to memory reuse, are removed.

Let us look at the execution of `FillSlots`. Figure 1 displays the call tree, with some kind of "call string" associated to each node. Each call string is actually a special kind of word over the statement labels, to be formally defined later. *As* and *AIEpAs* are instances of statement s, *AaIEpAITBr* and *AaIEpAITBbr* are instances of statement r.

Now, let us consider a word w pinpointing to one run-time instance of statement r, and let us assume that an IRDA gives a mapping σ, from w (the read of `A[j]` for the appropriate loop-counter j) to the run-time instance $\sigma(w)$ of statement s that defined the value. Then, we could expand array `A` into a tree `T` whose nodes are labeled by call strings. Since this paper focuses on IRDA, we neither discuss construction of `T` nor implementation of the transformed program. Therefore, we simply use the abstract syntax `T[`w`]` for the node associated with call string w. The resulting program in single assignment form appears in Figure 2.

Now every instance of statement s writes in a distinct memory location, all these instances can be executed in parallel. Two instances of the inner loop—i.e. the block introduced by B—can also be executed in parallel. Loop B itself may also be parallelized, but we need more information on statement r. We can sketch an abstract parallelization scheme for `FillSlots`:

- First, execute all instances of s—i.e. all writes—in parallel and wait for execution completion;
- Then, run all B loops in parallel, executing each one

```
      void FillSlots (int n) {
A/a     for (i = 0; i <= M; i++) {
s         T[w] = ···;
I         if (n == 0)
T         { /* then */
B/b         for (j = 0; j <= N; j++)
r             ··· = ··· T[σ(w)] ···;
E         } else
p           FillSlots (n-1);
        }
      }
```

Figure 2. Transforming `FillSlots`.

sequentially.

4. An Intuitive Flavor of the Analysis

We have shown that IRDA is a critical point in automatic parallelization of recursive programs like `FillSlots`. Let us now focus on the IRDA itself.

4.1. A Few Definitions

An Alphabet of Labels Program statements have been labeled for easier reference. There are two *operative* statements: s writes into array `A` and r performs some read access in `A`. Statement I is a conditional whose branches are T (`then`) and E (`else`), and statement p a recursive call to procedure `FillSlots`. Loop statements are decomposed into two sub-statements which are given distinct labels: The first one denotes the loop *entry*—e.g. A or B—and the second one denotes the loop *iteration*—e.g. a or b.

We get an alphabet $\Sigma = \{A, a, s, I, T, E, B, b, r, p\}$. In English, words are ordered by the *lexicographical* order generated by the alphabet order $a \lhd b \lhd c \lhd \cdots$. Similarly, in any program one can define a partial "textual" order $\lhd$ over statements. Statements in the *same basic block* are sorted in *apparition order*, and statements appearing in *different blocs* are *mutually incomparable*. In our example: $r \lhd b$ in the block introduced by the inner loop, $s \lhd I \lhd a$ in the outer loop. And that's all: T and E are branches of a conditional, B and p appear in different blocks . . .

Using $\lhd$ we see that instance w' executes before instance w iff the first differing label in w' and w are ordered by $\lhd$: I.e. if α' (resp. α') is the first label in w (resp. w') after the greatest common prefix of w and w', one must have $\alpha' \lhd \alpha$.

Call Strings and Control Words In the previous section, we introduced an intuitive labeling scheme for nodes in call trees by so-called "call strings". However, to be consistent with formal language theory, we will talk about "words" instead of "strings". Second, call strings have a somewhat different meaning in the literature: They usually capture the matching of call- and return-sites, and therefore build context-free languages. In this paper, *control words* capture the flow of control: Each control word corresponds to a possible run-time instance of a program statement. (Whether this instance is actually executed or not depends on conditionals and loop bounds, which have not been taken into account.) They build a rational (a.k.a. regular) control word language: In the case of `FillSlots`, we show Section 5.1 and Figure 4 that instances of s build language *A(IEpA+a)* s* and instances of r build language *A(IEpA+a)* ITBb* r*.

Execution Order is Lexicographical Order on Control Words As a result, the sequential execution order over instances is exactly the *lexicographical order*, over control words, generated by $\lhd$. We write $w' \ll w$ the fact that instance w' executes before w. In the following, trees of control words are sorted from left to right, according to $\ll$.

4.2. Reaching Definitions for `FillSlots`

Let us compute "by hand" the instance-wise reaching definitions for `FillSlots`. We are concerned in read accesses to `A[j]` performed by statement r. First, notice that statement r can only be executed when $n = 0$, i.e. when array `A` has been fully initialized by the successive calls to `FillSlots`. Moreover, the j loop-counter indexing array `A` in statement r ranges between 0 and N. As a result, values read by r have always been *defined*.

We want to compute, given the control word w of an instance of statement r, the control word w' of the reaching definition, i.e. the control word of the last instance of s which executes before w and writes into the array element read by w. This reaching definition is such that the value of n at instance w' is the same as the value of j at instance w. Since j is exactly the number of b in w and n is exactly N minus the number of p in w', we have $|w|_b = N - |w'|_p$ where $|w|_\alpha$ is the *occurrence number* of α in word w.

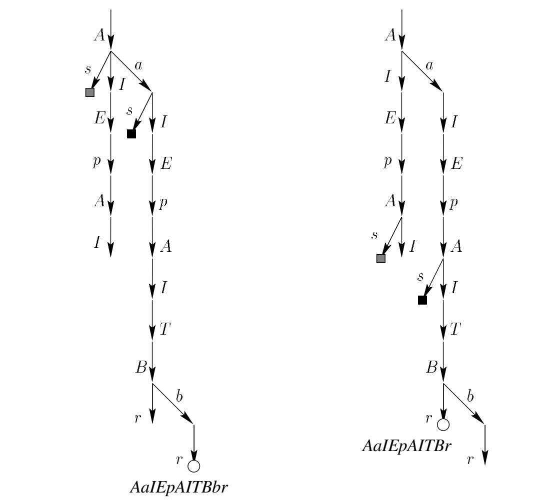

Figure 3.a. Studying *AaIEpAITBbr*. Figure 3.b. Studying *AaIEpAITBr*.

Figure 3. Source examples for $N = 1$.

Suppose $N = 1$, and let us study instance *AaIEpAITBbr* as shown by a circle in Figure 3.a. *AaIEpAITBbr* reads `A[1]`. *Aas*—the black square—also writes in `A[1]`. No instance executing between *Aas* and *AaIEpAITBbr* can assign `A[1]` since its control word cannot hold any occurrence of p. Moreover, we are sure that *Aas* is actually executed; Therefore other instances writing in the same array element such as definition *As*—the gray square in Figure 3.a—cannot reach the read, since they are always "killed" by *Aas*. *Aas* is thus the *reaching definition* of *AaIEpAITBbr*. Similarly, the definition reaching *AaIEpAITBr* in Figure 3.b is *AaIEpAs*—the black square—since all other write instances—like the gray square *AIEpAs*—are killed by *AaIEpAs*.

The reaching definition computation for these two examples may actually be generalized. It can be proven—and we will do this automatically in Section 7—that the definition reaching some instance w of r is an instance of s of the form us where u is the greatest prefix of w s.t. $|w|_b = N - |u|_p$. Let us call $\sigma(w)$ the definition instance reaching w. We have been able here to compute the unique reaching definition for each instance of r. But in general, σ maps instances to *sets of possible reaching definitions*, since extricate conditional expressions may cause approximative results.

5. Formalization of the Analysis

We start with some vocabulary in formal language theory, see [4] for details. A *transduction* is a relation over formal languages. It is also seen as a function from one language to subsets of an other. A *transducer* is the "engine" to "code" this relation. A *rational* (resp. *push-down*) transducer is a generalized finite (resp. push-down) automaton with *output*. It implements a *rational* (resp. *push-down*) transduction. Notice that a rational transduction is also context-free. See Figure 5 for simple transducer examples. In the following, we use the shorter "transducer τ" for "transducer implementing transduction τ".

5.1. Program Model

We handle programs with recursive calls, any loop (with unrestricted bounds), and any conditional statement (with unrestricted predicates); `goto`s are removed by classical processing. We only consider scalars and one-dimensional arrays. We call *induction variables* the integer arguments of a procedure that are initialized, incremented or decremented by a constant value at each recursive call. Loop counters are also considered as induction variables since loops can be rewritten in terms of recursive calls. Moreover, we require that all array subscripts be affine functions of induction variables and symbolic constants. For expository reasons, we make the following simplifications: We consider a single array `A`, any number of read accesses, and *only one assignment* s writing into array `A`.

For instance, procedure `FillSlots` has two induction variables appearing in array subscripts: n and j; j is incremented at each inner loop iteration, and n is decremented at each recursive call. (Notice that these conditions subsume "traditional" affine assumptions on loop nests.)

The alphabet of statement labels (see Section 4.1) is denoted with Σ. For a word w in Σ^* and a letter α in Σ, $|w|_\alpha$ denotes the number of occurrences of α in w.

Control words are formally defined as words recognized by the *control automaton* of the program. It is a *finite automaton* whose states are blocks and statements in the program. An *edge* from a state $\mathbf{q}_1$ to a state $\mathbf{q}_2$ express that statement (or block) $\mathbf{q}_2$ occurs in block $\mathbf{q}_1$; It is labeled by the label of $\mathbf{q}_2$. Operative statements yield termination states (see Figure 4). L denotes the control word language.

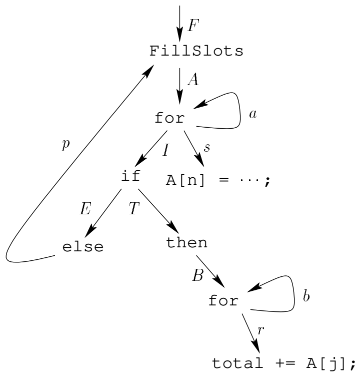

Figure 4. Control automaton for `FillSlots`.

5.2. Reaching Definitions as Transductions

The applicability of our transduction model relies on the two following properties:

- Without considering loop bounds and predicates, L is a *rational* (a.k.a. *regular*) language. This result is a straightforward application of L being the language recognized by the *control automaton*.
- Now consider array subscripts. Due to our syntactical restrictions, a subscripting function f is affine w.r.t. the occurrence numbers of labels in control words and w.r.t. symbolic constants. I.e., if $\Sigma = \{\alpha_1, \ldots, \alpha_n\}$ and if $N_1, \ldots, N_m$ are symbolic constants, then there exist $n + m$ numerical constants $a_1 \ldots a_{n+m}$ s.t.:

$$f(w) = a_1 |w|_{\alpha_1} + \ldots + a_n |w|_{\alpha_n} + a_{n+1} N_1 + \ldots + a_{n+m} N_m. \quad (1)$$

The formal proof of this result is not given here, for lack of space (A restricted case is studied in [6]).

Let g be the subscript function to array `A` in the right-hand side (the read) and f the function subscripting array `A` in the left-hand side. Let us consider the read in `A` by an instance

w of s. A read w is *dependent* on instance w' of s if the three conditions below hold.

Conflicting accesses: w and w' hit the same memory cell:

$$g(w) = f(w'). \tag{2}$$

Ordering: The write w' has to execute before the read w, denoted by $w' \ll w$. Actually, $\ll$ is the lexicographical order on control words, defined by:

$$w' \ll w \iff w = u\alpha v \land w' = u\alpha' v' \land \alpha' \lhd \alpha. \tag{3}$$

Execution: Obviously, write w' has to execute. This is denoted by $e(w')$.

Then, the reaching definition of a read w is *the last write instance*, with respect to order $\ll$, satisfying these three conditions: $\max_{\ll}\{w' : g(w) = f(w') \land w' \ll w \land e(w')\}$.

Clearly, this maximum is unique in the course of program execution. However, since we make no assumption on recursion guards, we cannot prove nor disprove, *a priori*, that an instance executes. So, we have to approximate this maximum as a *set* $\sigma(w)$ *of possible reaching definitions*. This approximation may be more or less precise, depending on the knowledge on $e(w')$ and on how well this knowledge fits the context-free transduction framework. This (compile-time) knowledge may come from structural properties of the program, analysis of conditional expressions, etc. At first, loop bounds and conditional predicates (such as recursion guards) are *not taken into account*. We thus restrict ourselves to the transduction built from (2) and (3):

$$\delta(w) = \{w' \in \Sigma^* s : f(w') = g(w) \land w' \ll w\} \tag{4}$$

Transduction δ captures the true dependences of the program. We will prove in the next section that δ is a context-free transduction. But our aim is still to perform a reaching definition analysis; And true dependences are no good approximation, in general. Our IRDA scheme applies various rewriting rules to δ, using compile-time knowledge on the existence condition $e(w')$:

- If we can prove that some statement instance *does not* execute, and if adding this information keeps the transduction context-free, some true dependences can be removed. The remaining instances are described by predicate $e_{\text{MAY}}(w')$ (instances that *may* execute).
- On the opposite, suppose we can prove that some statement instance w'' *does* execute, and that adding this information keeps the transduction context-free. Then, writes executing before w'' are "killed": They cannot reach an instance w such that $w'' \in \delta(w)$. Instances that are effectively executed are described by predicate $e_{\text{MUST}}(w')$ (instances that *must* execute).

The number and complexity of these rules depends on the program and on the precision required. This issue is studied in more detail in the next sections.

At last, the result of our IRDA is a new transduction, call it τ, which is an over-approximation of σ: $\sigma(w) \subseteq \tau(w)$. Formally, an instance w' is in $\tau(w)$ iff it is in dependence with w, It executes (according to the static knowledge we have), and it is not killed by an other dependence:

$$\tau(w) = \{w' : w' \in \delta(w) \land e_{\text{MAY}}(w') \land (\nexists w'' \in \delta(w) : w' \ll w'' \land e_{\text{MUST}}(w''))\} \tag{5}$$

Let us now give a constructive proof that τ is context-free.

6. Building the Context-Free Transducer

We first present a construction of the dependence transducer δ, then show how to refine its output, in order to get a precise description of reaching definition instances.

6.1. The Dependence Transducer

Implementing the lexicographical order with a transducer (3) is easy, it is a well known example of rational transducer. Suppose we have just read a prefix u of w: Let $w = u\alpha v \in L$, $u \in \Sigma^*$, $\alpha \in \Sigma$ and $v \in \Sigma^*$. To satisfy the lexicographical order, we have to output any word $w' = u\alpha' v'$ s.t. $\alpha' \lhd \alpha$. See example in Figure 5.a.

Satisfying the conflicting access condition (2) is done using a counter. The counter, therefore, checks whether $g(w) - f(w')$ is equal to zero. To comply with the standard definition of push-down transducers, the counter is implemented as a stack in a natural way [4, 15]. Such a push-down transducer allows the following counter arithmetic:

- Initialization to a constant, noted CNT := cst;
- Adding a constant, noted CNT += cst or CNT −= cst;
- Comparing with zero, noted CNT $= 0$ or CNT $\neq 0$.

Figure 5.b is an example of conflict equation transducer.

Intersecting the lexicographical order and conflict equation transducers is easy: The first one is essentially "state-based" and the second one is "stack-based"[2]. It yields a three-state transducer checking the counter in three stages.

1. Read and write a common prefix: Transitions are of the form $\alpha v/\alpha v$, $\alpha \in \Sigma$, $v \in \Sigma^*$.
2. Read the input suffix: Transitions are of the form $\alpha v/\varepsilon$.
3. Write an output suffix that zeroes the counter: Transitions are of the form $\varepsilon/\alpha v$.

Therefore the result is a push-down transducer.

The last stage consists in restricting input words to instances of read statements, and output words to instances of s. We need the following notations: Let INPUT (resp. OUTPUT) be the transducer build from the finite automaton recognizing instances of read statements (resp. statement s)

[2] The intersection of a rational transduction and an context-free one is in general not context-free.

in copying its input to its output—i.e., when an edge in the finite automaton is labeled by α, the corresponding edge in INPUT is labeled by α/α. Composing the input of our transducer with INPUT and the output by OUTPUT, we get the dependence transducer δ. We can apply Elgot and Mezei's theorem [3] [4] since we compose two rational transducers—INPUT and OUTPUT—with a context-free one. It proves that δ is a context-free transducer. Notice that Elgot and Mezei's theorem is constructive: Building δ is done automatically from INPUT, OUTPUT, $\ll$, g and f.

6.2. The Reaching Definitions Transducer

Here comes a few examples of rewriting rules that can be applied to refine the set of possible reaching definition instances. Any statement instance us where u is a *strict prefix* of some control word w is called an *ancestor* of w. E.g., *AaIEpAs* is an ancestor of *AaIEpAIBr* (see Figure 3.b).

The LAD **property** (Left of Ancestor in Dependence). Let w be some read instance and w' in dependence with w. Suppose an ancestor $u's$ of w' is also in dependence and executes before w'. If w' executes, u' executes too, and then $u's$ does. Therefore $u's$ kills every instance executing before it, including w'. (w' may execute or not, but when it does, it is necessarily killed.) See [6] for details.

$$\text{LAD} \iff (\exists \alpha \in \Sigma : \alpha \lhd s).$$

Rule LAD then applies: If some word $u's$ is in dependence with w, no words prefixed by $u'\alpha$ with $\alpha \lhd s$ can be reaching definitions of w. Formally, transitions $\varepsilon/\alpha v$ ($v \in \Sigma^*$) with $\alpha \lhd s$ are transformed into $\varepsilon/\alpha v$: $\text{CNT} \neq 0$.

The MSF **property** (Monotonic Subscript Function). A non-decreasing function f is s.t. $\forall uv \in L, u, v \in \Sigma^* : f(uv) \geq f(u)$; Non-increasing functions are defined similarly. A function is *monotonic* when it is either non-decreasing or non-increasing.

Suppose the *write* access function f to array A (in s) is monotonic, $s \lhd \alpha$ and $\alpha v/\alpha v$ is a transition that does not modify the counter. Let w be some read instance and us an ancestor of w, in dependence with w. We know that $u\alpha vs$ is executed—being an ancestor of w. And $u\alpha vs$ is in dependence with w, since transition $\alpha v/\alpha v$ does not modify the counter; Thus $u\alpha vs$ kills us.

$$\text{MSF} \iff f \text{ is monotonic}.$$

For all $\alpha \in \Sigma$ and $v \in \Sigma^*$ s.t. $s \lhd \alpha$ and $\alpha v/\alpha v$ does not modify the counter: Rule MSF consists in following transition $\alpha v/\alpha v$ as much as possible before outputting s (to get the biggest ancestor). Formally, transition α/s is removed, if not the only transition that outputs s. Application of this rule is shown at Section 7.

[3] More precisely, a generalization to context-free transductions.

The VPA **property** (Values are Produced by Ancestors). This property comes from the common observation about recursive programs that "values are produced by ancestors." Indeed, a lot of sort, tree, or graph-based algorithms perform in depth explorations where values are produced by ancestors. This behavior is also strongly assessed by scope rules of local variables. Formally,

$$\text{VPA} \iff (w' = \sigma(w) \Rightarrow (\exists u, v : w = uv \wedge w' = us)).$$

Rule VPA then consists in removing all transitions producing non-ancestors. Formally, all transitions α/α' s.t. $\alpha' \lhd \alpha$ and $\alpha' \neq s$ are removed.

The OKA **property** (One Killing Ancestor).
Let w be a read instance. If it can be proven that at least one ancestor us of w is in dependence, it kills all writes executing before (since it does executes when w does).

$$\text{OKA} \iff (\delta(w) \neq \emptyset \Rightarrow (\exists u, v : w = uv \wedge us \in \delta(w))).$$

Property checking Satisfaction of property LAD is easily obtained from $\lhd$. Checking property MSF is done by examinating transitions of the form $\alpha v/\alpha v$ in δ and coefficients in f (their sign must be the same). Property OKA can be discovered using invariant properties on induction variables (see Section 7). Checking for property VPA is difficult (automatically or not), but we have the following result:

When both OKA *and* MSF *hold,* VPA *also holds.*
Suppose δ' is the transducer after application of the MSF rule, and—by application of OKA—suppose $us \in \delta'(w)$ is an ancestor of w. Any word $u\alpha vs$ with $s \lhd \alpha$ and $v \in \Sigma^*$ is s.t. $f(u\alpha vs) \neq f(us)$, by definition of us. Therefore, no instances in dependence execute after us: Property VPA is satisfied. This result allows to apply rule VPA from properties MSF and OKA.

To summary this section: Our IRDA checks for properties LAD, MSF and OKA—possibly using external classic analyses; Then applies rules LAD, MSF and VPA accordingly.

More properties can probably be useful that can be obtained by more involved analyses [9]. The problem is to find a relevant and efficient rewritting rule for each one.

6.3. Algorithm for IRDA

In this section, we integrate the precedent techniques in a general IRDA algorithm.

Input: Σ, f, g, L (from the control automaton), and $\lhd$.
Step 1: Build the lexicographical order transducer $\ll$.
Step 2: Build the transducer checking $g(w) = f(w')$.
Step 3: Intersect the two, yielding a three-state transducer.
Step 4: Compute transducers INPUT and OUTPUT from L.
Step 5: Compose the input by INPUT and the output by OUTPUT. Here is transducer δ.

Step 6: Check properties LAD, MSF and OKA using external static analyses. Asking the user for help may be useful for VPA or other properties not described here.

Step 7: Apply rewritting rules accordingly. The result is the reaching definition transducer.

Output: Return dependence and reaching definition transducers δ and τ.

Notice τ and δ are both useful to compilers. Section 8 discusses decidability results on describing dependences and reaching definitions with push-down transducers.

7. Back to procedure `FillSlots`

Let us show how to compute the transducer τ describing the data flow of `FillSlots`. Here we have $g(w) = |w|_b$ and $f(w') = N - |w'|_p$: The conflict equation is

$$g(w) - f(w') = |w|_b + |w'|_p - N.$$

Intersecting the lexicographical order and the conflict equation, then applying composition with INPUT and OUTPUT, we get the dependence transducer in Figure 5.c.

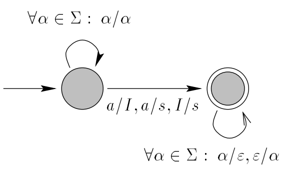

Figure 5.a: Transducer for lexicographical order.

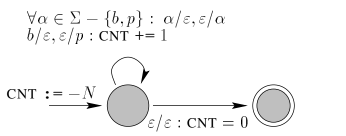

Figure 5.b: Transducer for conflict equation.

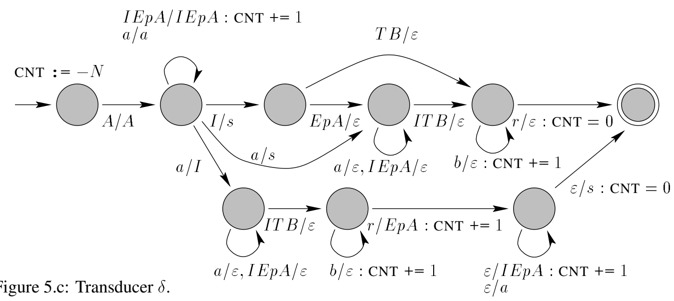

Figure 5.c: Transducer δ.

Figure 5. Computing the dependence transducer for `FillSlots`.

Rule LAD does not apply since no label precedes s for $\triangleleft$. Rule MSF applies since $f(w')$ is a non-increasing function (as the length of w' increases). Transition a/a does not modify the counter, therefore edge a/s is removed; But transition $IEpA/IEpA$ increments the counter thus I/s is kept.

We have seen in Section 4.2 that $0 \leq j \leq N$ and $n = 0$ holds for all instances of statement r. These properties can be easily discovered using classical techniques [2, 9]. As a consequence, if w is an instance of r there exists a prefix u of w s.t. $|us|_p = N - |w'|_b$. Property OKA is satisfied. We have seen that combined properties OKA and MSF forces reaching definitions to be ancestors: Rule VPA thus applies. It consists in stripping out a/I (and subsequent unreachable states and edges). The result is shown in Figure 6.

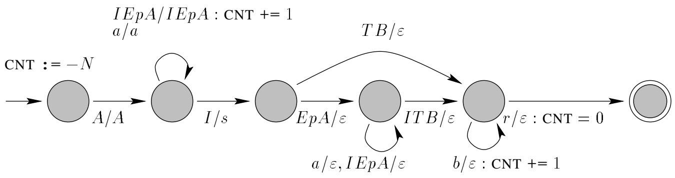

Figure 6. Reaching definitions for `FillSlots` after application of VPA and MSF.

This is our result for τ. It computes in fact a unique reaching definition for every instance: We achieved—in an automated way—the best precision possible. We also gave a formal description of the result of Section 4.2.

8. Discussion on This Approach

The present framework can be extended to programs with several arrays and assignments: The weakest solution to handle several assignments to the same array consists in taking the union of context-free languages, which is context-free. We also believe that this transduction-based framework can be extended to support IRDA over more general data structures, such as trees [13]. Formal languages seem very natural for compile-time analysis of recursive programs. Moreover, using context-free transducers has the following benefits:

- It can be decided whether a word is a reaching definition of another one.
- It is decidable whether a language of reaching definitions is empty or not, and therefore, whether an instance reads an uninitialized value. This makes our IRDA useful to precise program checking.
- Computing the cardinality of a set of reaching definitions is also possible. This is useful to detect when IRDA gives exact results (sets are singletons).
- Context-free languages are closed under intersection with rational languages. Any decidable property on the reaching definitions of a given read is thus decidable for a rational set of read instances.

We have seen on example `FillSlots` that instancewise reaching definitions can be of great help in parallelizing recursive programs. Indeed, benefits of IRDA have already been studied in the case of Fortran loops [12, 18], but we advocate for an extension to recursive programs. First, IRDA allows to remove dependences due to memory reuse—via single assignment form transformation; Second, since it works on a per-instance basis, both fine-grain and coarse-grain parallelism can be discovered. Several problems obviously remain. Two of them are studied below.

1. The exact language of possible reaching definitions is not context-free: Obviously, many properties on the existence condition $e(w')$ cannot be taken into account in our framework. More precise IRDAs require more powerful classes of languages. Indexed grammars [1] are an extension of context-free grammars useful to our context. This research is left for future work.
2. Multidimensional arrays could be handled by intersecting reaching definitions languages. Unfortunately, neither context-free languages nor indexed grammars are closed under intersection. Scattered context grammars [14] are an extension of context-free languages closed under intersection; Studying the applicability of this class is left for future work too.

9. Conclusion

This article presents a novel application of formal languages to the automatic discovery of some semantic properties of programs: Reaching definitions. When programs are recursive and no property is known at compile-time on recursion guards, only a conservative approximation can be hoped for. In our case, we approximate the set of possible reaching definitions of a given read by a context-free language. The relation between reads and their respective languages of possible reaching definitions is context-free, by construction. The result of the analysis is thus a push-down transducer mapping control words of read instances to control words of write instances. Decidability properties on context-free languages and transductions allow several applications of our framework, especially in automatic parallelization of recursive programs. These applications include array expansion, parallelism extraction, and aggressive optimization and verification techniques.

Acknowledgments The authors are supported by the French *Ministère de l'Enseignement Supérieur et de la Recherche* (MESR), the *Centre National de la Recherche Scientifique* (CNRS), INRIA project AAA, and the German-French ProCoPe program.

We would like to thank Yvan Djelic, Paul Feautrier, and Martin Griebl for their help and fruitful comments about this topic.

References

[1] A. V. Aho. Indexed grammars – an extension to context free grammars. *J. ACM*, 15:647–671, 1968.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, Mass, 1986.

[3] M. Alt and F. Martin. Generation of efficient interprocedural analyzers with PAG. In A. Mycroft, editor, *Int. Symp. on Static Analysis, SAS'95*, volume 983 of *LNCS*, pages 33–49, Glasgow, UK, Sept. 1995. Springer Verlag.

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

[5] D. R. Chase, M. N. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *SIGPLAN Conf on Programming Language Design and Implementation*, volume 25, pages 296–310, June 1990.

[6] A. Cohen. Analyse de flot de données pour programmes récursifs à l'aide de langages algébriques. *TSI*, 1998. To appear.

[7] J.-F. Collard, D. Barthou, and P. Feautrier. Fuzzy array dataflow analysis. In *Proc. of 5th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 92–102, Santa Barbara, CA, July 1995.

[8] J.-F. Collard and J. Knoop. A comparative study of reaching definitions analyses. Technical report, PRiSM, U. of Versailles, 1998.

[9] P. Cousot. *Program Flow analysis: theory and applications*, chapter Semantic foundations of programs analysis, pages 303–342. Prentice-Hall, 1981.

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

[11] A. Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 230–241, Orlando, June 1994.

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

[13] P. Feautrier. A parallelization framework for recursive tree programs. In *EuroPar*. LNCS, 1998. To appear.

[14] S. Greibach and J. Hopcroft. Scattered context grammars. *J. of Computer and System Sciences*, 3:233–247, 1969.

[15] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[16] J. L. Jensen, M. E. Jorgensen, M. I. Schwartzbach, and N. Klarlund. Automatic verification of pointer porgrams using monadic second-order logic. In *ACM SIGPLAN Conf on Prog. Lang. Design and Implem. (PLDI)*, 1997.

[17] W. A. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *ACM SIGPLAN Conf on Prog. Lang. Design and Implem. (PLDI)*, pages 56–67, June 1993.

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

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

[20] W. Pugh and D. Wonnacott. Eliminating false data dependences using the omega test. In *ACM SIGPLAN PLDI*, 1992.