A Precise Fixpoint Reaching Definition Analysis for Arrays

Jean-François Collard
CNRS - PRiSM. University of Versailles.
45 avenue des Etats-Unis. 78035 Versailles. France.
jfc@prism.uvsq.fr

Martin Griebl
FMI. University of Passau.
Innstr 32. 94032 Passau. Germany.
griebl@fmi.uni-passau.de

Abstract

This paper describes a precise reaching definition (RD) analysis tuned for arrays.

RD analyses are of two kinds. The first group, Maximal Fixed Point (MFP) analyses, considers arrays as indivisible objects and does not contrast the side-effects of separate instances of writes. Its main benefit, however, is its wide applicability (e.g. to any unstructured program). On the other hand, analyses based on integer linear programming are able to pinpoint, for a given read instance, which instance of which write reference actually defined the read value. They are, however, restricted to limited classes of programs.

Our analysis tries to take the best of both worlds by computing, in an iterated MFP framework, instancewise RDs of array elements.

1 Introduction

Over the past decade, two different frameworks have been succesful in reaching definition analysis.

The first one is based on abstract interpretation, with one main feature being an iterative propagation of data-flow information until a maximal fixed point (MFP) is reached [1]. Its main benefit is its wide applicability, e.g., to any unstructured program. Its drawback is that the granularity of control and data units is pretty coarse: statements as well as arrays are usually considered as atomic units, i.e., these analyses do not distinguish between different array *cells*, or between different run-time *instances* of a statement, e.g., caused by loops.

The second framework is tailored for arrays and is able to pinpoint, for a given read instance, which instance of which write reference actually defined the read value. Methods in this framework are typically based on integer linear programming (ILP) and may be caracterized by their use of symbolic solvers to precisely analyze arrays elementwise. However, they are usually restricted to very limited classes of programs.

Of course, this classication is rough, and several works are closely related (for related work of the two frameworks cf. Section 3 and 4). But none of them, to the best of our knowledge, answers (if possible) the following question: given a run-time instance of a read reference in a possibly unstructured program (e.g., containing `goto`s), which instance of which write statement defines the read value?

To answer this question, this paper tries to take the best of both worlds. This is not simply a philosophical issue, but it has also practical relevance: the instancewise analysis described in this paper combines the applicability of the first framework with the power of the second framework. Furthermore, since the analysis is based on iterative propagation on the program's control-flow graph, it makes instancewise analysis closer to abstract interpretation and, hence, may allow in the future, to extend to arrays other well-known data-flow analyses, not just reaching definitions.

2 A Motivating Example

Consider program `piter` in Figure 1. (Its Control-Flow Graph (CFG) appears in the same figure but will be discussed later on. Note that we model statements as *edges* instead of nodes, following

```
    real A[0:2*N+1]
    for i = 0 to N
       for j = 0 to N
3         A[i+j+1] := ..
          if (P) then
6            A[i+j] := ..
          end if
7         ..  := A[i+j]
       end for
    end for
```

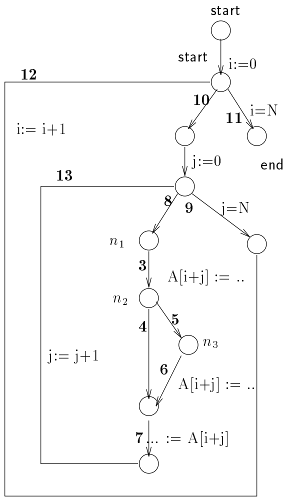

Figure 1: Program `piter` and its Control-Flow Graph.

the findings of Knoop et al. [11]. Our analysis, however, applies to the traditional definition of CFGs as well.)

Since reaching definition analyses of the MFP framework consider array `A` as a one indivisible object and statements **3**, **6** and **7** as atomic units, the result of the analysis will be: both, **3** or **6**, can be reaching definitions for the read of `A` in **7**.[1]

However, the information we want to achieve on the example is:

- At which iterations of `i` and `j` may **7** read an *uninitialized* value? In other words, which instances of **7** may be reached by no definition? (The answer is none of them, except perhaps for the very first.)
- Which instances of **3** and/or **6** reach a given, parametric instance (i, j) of **7**?

After having revisited the two frameworks in more detail in the next two sections, we shall derive our new method in Section 5, and afterwards come back to this example in more detail.

3 The MFP framework

Traditional RD analyses are based on iterative resolution of dataflow equations. More precisely, they rely on a global (i.e., with respect to all possible paths) specification of a dataflow problem, called a Meet-Over-all-Paths (MOP) specification, and on a practical local specification, called Maximal Fixed Point (MFP) whose resolution is iterative. The equality of the MOP specification and the MFP solution has been proven in the Coincidence Theorem [10], extended to the interprocedural case in [12].

We detail below the current drwabacks associated with MFP techniques or with MOP specifications.

Array Cells No MFP technique we know of distinguishes between different array elements, except for the work in [8] detailed below. However, this is a big issue in, e.g., automatic parallelization or program verification.

[1] Things are even worse for dataflow analyses based on lexical names of variables. Then, `A[i+j]` is considered as a variable name (a string with 6 characters), and `A[i+j+1]` is considered as a separate variable (another string, with 8 characters). Definitions of the latter can therefore never reach use of the former. See [6] for details.

Instances of Statements Typically, different elements of arrays are referenced at different instances of the same statement, e.g., inside a loop. Hence, the desire for elementwise array analysis enforces an *instancewise* analysis. Or, the other way round: as long as the analysis only considers scalar variables, all the instances of a scalar assignment write into the same memory location. However, this does not hold for arrays, as the example in Figure 1 shows: the first instance of **3** writes into `A[1]`, the second into `A[2]`, etc. Hence the need for an instancewise RD analysis.

A paper by Duesterwald et al. [8] is close to our goal. That paper extends the classical framework so as to express, in limited cases, the instances of reaching definitions. Array subscripts must be affine functions of one variable only (the counter of the innermost loop). Moreover, multidimensional arrays are linearized, which requires that array bounds are known numerically at analysis time.

They consider nests with a single loop (or the innermost loop only), and derive, for the read in a given iteration i, the distance d such that the defining write instance occurred at iteration $i - d$. The set of possible values for d is limited to an integer lattice (a range of integer values, plus a bottom *bot* element and a top $\top$, to denote the iteration count of the loop). Unfortunately, this set of values may not be expressive enough in some cases.

Precision on the Control-flow Another issue is how precisely control-flow is taken into account. In MFP, all possible paths in the control-flow graph are considered to be valid (in the intraprocedural case), independently of the possible outcomes of predicates in `if`s, `while`s, etc. However, this may lead to a big loss in precision.

```
1  if P(..) then x := 0
3  if P(..) then x := 1
5  y := x
```

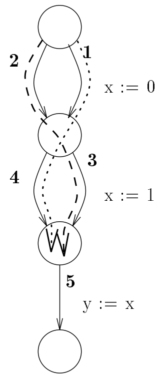

Figure 2: Impossible execution paths.

To see why, consider the example in Figure 2, and assume predicate P does not depend on x. Then, depending on the outcomes, the execution flows either through both **1** and **3**, or through none of them. In both cases, the definition of `x` in **1** never can reach **5**.

In the MOP framework, all possible paths are considered. For instance, the two paths denoted with dashed lines in Figure 2 (i.e., paths **1**.**4** and **2**.**3**). These paths, however, never correspond to an actual execution. Our analysis takes advantage of a symbolic solver, Omega [14], to avoid this loss of precision. In particular, Omega provides *uninterpreted functions* that allow to manipulate functions on which we have little or no knowledge. On an uninterpreted function F, Omega is able to perform logic operations such as simplifying $F(x) \geq 0 \vee F(x) \leq 0$ to true, or simplifying $F(x) \geq 0 \wedge F(x) < 0$ to false. Such operations are exactly what we need to derive precise reaching definitions on the program in Figure 2 (coding for example $P(..)$ with $F(..) \geq 0$).

4 The ILP Framework

Originally, the instancewise array dataflow analysis based on integer linear programming (ILP) was designed for nested `for` loops whose bounds are affine expressions w.r.t. surrounding loop counters and symbolic constants. The loop body had to consist of assignments to arrays whose indices also had to be affine expressions in outer loop indices and symbolic constants [9]. In this setting, an *iteration vector* of a statement S is the vector which consists of the indices of all loops surrounding S, ordered outside-in. Every run-time "incarnation" of S is a separate instance of S which we precisely identify by S and the iteration vector i. We denote such an instance with $\langle S, i\rangle$. The *iteration domain* $D(\mathbf{s})$ of a statement S is the set of all iteration vectors of S. The iteration domain turns out to be convex (a polytope), and the order in which the iteration vectors are generated by the loop nest is the lexicographic order on the iteration vectors.

In the meantime, the analysis has been extended to more general but structured loops, such as `while` loops. Then, the iteration domain is not a polytope anymore but an unbounded polyhedron.

In other works, the flexibility of the basic idea has also been proved by extensions to explicitliy parallel programs [5] or even to recursive programs [4]. However, it turns out that the effort for solving the integer linear programs increases dramatically with the size of the input program.

In addition, since the basic approach relies on the structure of the input program, an extension to programs without any structure, e.g., containing arbitrary `goto`s, is not straight forward. Current methods in this framework [9, 3, 15] would just fail. A simple way out would be to use a preprocessing phase for structuring the program. However, this leads, in general, to significant code duplication. Furthermore, in an interactive compiler environment, the user will not be able to recognize his or her program.

5 Our MFP Technique for Instancewise Reaching Definition Analyses

Section 5.1 defines what an instance-based analysis means when arbitrary `goto`s appear. We then formally define the dataflow equations we solve. An intuitive view of the resolution process is given in Section 5.3, and the algorithm is formally given in Section 5.4. Convergence properies of the algorithm are discussed in Section 5.5.

5.1 How to Label Instances and Define the Execution Order in the Presence of `goto`s?

5.1.1 Structured Programs

An example of such a structured program is when `goto`s build nested loops, as in Figure 3.

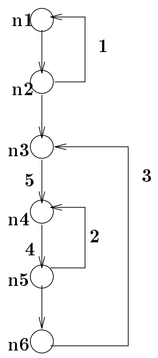

Figure 3: `goto`s building a structured program.

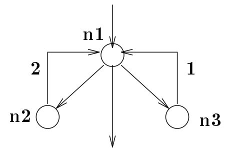

Figure 4: A graph were loops are ambiguous

Edge **4** (between n4 and n5) should be labeled by the iteration vector built from the number of **3**s and the number of **2**s, in this order, and assuming the count of **2** is re-initialized to 0 each time **5** is taken.

5.1.2 How Can We Tell a Program is "Structured"?

Natural loops do not correspond to the loops we want to detect. Consider for instance Figure 5. The natural loop [1] for **3** includes nodes n1, n2, n3 and n4, and edges **1**, **2**, **3**, **4** and **5**, and therefore surrounds the loop on **2**. However, we should not consider the two loops here as nested.

Structural analysis [13] does not provide the information we need either. Consider the example in Figure 4, taken from [13]. Structural analysis detects that this flow graph corresponds to two nested `while` loops. Which loop is innermost, however, depends on the order edges are visited. For our purpose, it would not make much sense in such cases to enforce a nest structure and to derive information about the lexicographic order of an index vector. Therefore, when `goto`s appear in a program segment, and when no separate analysis states otherwise, we handle this segment according to the following section.

5.1.3 Unstructured Program Segments

To precisely identify statement instances in the presence of backward edges, we would need to capture the exact sequence of traversed edges. One way to do this would be to reason on execution traces, or on words built on the alphabet of backward edges. For example, an instance of **4** in Figure 5 could be labeled with a word in $\{\mathbf{2},\mathbf{3}\}^*$. Previous papers of ours explored this way [4], and our experience is that such a labeling scheme is too complex to be usefulwith current compilation techniques.

Therefore, we pick the following trade-off: statements are labeled by the (unordered) list of edge counts, that is, by the *number of times* each edge was traversed *since the unstructured part was entered*. For instance, an instance of **4** in Figure 5 is labeled by (#**2**,#**3**), where #**2** and #**3** denote the number of times edges **2** and **3** are traversed, respectively. Clearly, the information we lose with this labeling is the order in which these edges are traversed.

One obvious question is how to choose the edges we should consider for a given instance. Let us remark that we may label a statement instance by an additional edge counter, even if traversing this edge does not lead to distinct instances. Inversely, if we "forget" such an edge, we will just have a semi-instancewise analysis, where several instances are collapsed into "super-instances". The simplest and most general solution, however, is to label a statement **s** with the edge counts of all back-edges whose natural loops include statement **s**.

5.1.4 What is then the Execution Order on Instances?

When programs are structured, the execution order is the same as if the loops had been made explicit, i.e., the order is the lexicographical order on vectors built from the edge counts. For example, the order between two instances of **4** from Figure 3 is:

$$\langle \mathbf{4}, \#\mathbf{2'}, \#\mathbf{3'} \rangle \prec \langle \mathbf{4}, \#\mathbf{2}, \#\mathbf{3} \rangle \iff \#\mathbf{2'} < \#\mathbf{2} \ \vee\ (\#\mathbf{2'} = \#\mathbf{2} \wedge \#\mathbf{3'} < \#\mathbf{3})$$

However, now consider Figure 5. Obviouly, the lexicographical order on the numbers of edge traversals does not correctly capture the order anymore. What we can say, however, is that a given

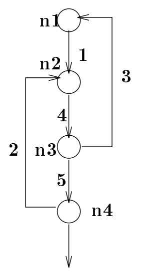

Figure 5: Control-flow graph of an unstructured programs.

execution will have one single trace, i.e., one single path along edges **3** and **2**. To see this, consider an instance of **4** reached after traversing **3**, **3** again, then **2**, which we denote by a word built by concatenating edge labels: **3**.**3**.**2**.**4**. Then, previous incarnations of **4** occured just after **1**, denoted ϵ.**4** (ϵ is the empty label), then at **3**.**4**, and then at **3**.**3**.**4**.

What we see is that the execution enforces that the numbers of occurences of all shared back edges must be ordered by $\leq$. That is:

$$\langle \mathbf{4}, \mathbf{\#2'}, \mathbf{\#3'} \rangle \prec \langle \mathbf{4}, \mathbf{\#2}, \mathbf{\#3} \rangle \Leftrightarrow \mathbf{\#2'} \leq \mathbf{\#2} \wedge \mathbf{\#3'} \leq \mathbf{\#3}$$

5.1.5 The General Case

In general, a program segment may feature intermingled structured and unstructured parts. Each part should be handled according to the appropriate execution order described above. Consider for example the CFG in Figure 6. An appropriate index for instances of Statement **5** is

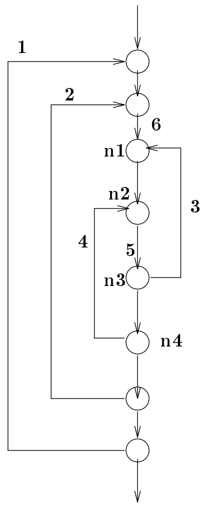

Figure 6: Control-flow graph of an unstructured programs.

(**#1**,**#2**,**#3**,**#4**). (Other indices are acceptable too, e.g. (**#1**,**#2**,**#4**,**#3**)). Remember that these edge counts are reinitialized to 0 when the unstructured program part is entered, i.e., each time edge **6** is traversed.

To capture the execution order, we need to use the fact that Edges **1** and **2** make a structured part, and edges **3** and **4** make a nonstructured part. Therefore, the correct order is defined in two steps:

1. The index vector is cut into subvectors, each subvector corresponding to a structured or unstructured part. Subvectors corresponding to a structured part are compared by the lexi-

cographic order $\ll$, those corresponding to an unstructured part are compared by pointwise $\leq$. Each comparison gives $<$, $=$, or $>$. (Note that the case of uncomparability does not appear for instances of the same execution.) For instance, in the running example, vectors are compared lexicographically on the first two components, and with pointwise $\leq$ on the last two. E.g., (4, 0, 2, 3) and (4, 1, 2, 2) yield $(<, >)$.

2. Two vectors are ordered if their first respective subvectors are ordered by $<$, or if the first two subvectors are ordered by $(=, <)$, or if the first three are ordered by $(=,=,<)$, etc.

5.2 MFP Formulation

Typically, the data-flow facts are given by the elements of a complete (not necessarily finite) lattice of finite height, the data-flow functions by transformations on these lattices, which in practice are monotonic or even distributive. Formally, the dataflow analysis is defined by a lattice $\mathcal{C}$ and a *local semantic functional* $[\![\ ]\!] : E \rightarrow (\mathcal{C} \rightarrow \mathcal{C})$ on graph edges E. This functional gives abstract meaning to the (elementary) statements of a program in terms of a transformation on a complete lattice $(\mathcal{C}, \sqcap, \sqsubseteq, \perp, \top)$ with least element $\perp$ and greatest element $\top$.

For a flow graph G, let $pred(n)$ and $succ(n)$ denote the set of all immediate predecessors and successors of a node n, and let $source(\mathbf{e})$ and $dest(\mathbf{e})$ denote the source and the destination node of an edge $\mathbf{e}$. A *finite path* in G is a sequence $e_1 \ldots e_q$ of edges such that $dest(\mathbf{e}_j) = source(\mathbf{e}_{j+1})$ for $j \in \{1, \ldots, q-1\}$. It is a path from m to n, if $source(\mathbf{e}_1) = m$ and $dest(\mathbf{e}_q) = n$. Additionally, $p[i, j]$, where $1 \leq i \leq j \leq q$, denotes the *subpath* $e_i, \ldots, e_j$ of p. Finally, $\mathbf{P}[m, n]$ denotes the set of all finite paths from m to n. Without loss of generality we assume that every node $n \in N$ lies on a path from start to end.

The point of the *maximal-fixed-point* (*MFP*) approach is to iteratively approximate the greatest solution of a system of equations which specifies the consistency between conditions expressed in terms of data-flow information of $\mathcal{C}$:

$$\mathbf{info}(n) \quad = \quad \bigsqcap \{ [\![(m, n)]\!](\mathbf{info}(m)) \mid m \in pred(n) \} \qquad \text{if } n \neq \mathsf{end} \tag{1}$$

The initial value for $n = \mathsf{end}$ is defined later.

Checking that our MFP coincides with its MOP is an interesting theoretical issue left for future work. In particular, the coincidence theorem states that the algorithmic MFP-solution is equal to (coincides with) the expected result specified by MOP if all semantic functionals are distributive.

5.3 Intuitive Overview of our Algorithm

We present an MFP algorithm to compute definitions reaching a given read instance. Basically, this algorithm boils down to propagating up the control flow graph a set which contains three kinds of information:

- The identity of definitions that can reach a given read of some memory location. These definition instances are described by (in)equalities on loop counters and symbolic constants. When these (in)equalities are not affine, an affine approximation will be made along the lines of [3].
- A conjunction of predicates collected while traversing `if`s and `while`s
- A predicate that captures the execution order on computations (more precisely, a disjunct of the lexicographical order. See below.)

We start with a conservative information. Then, when we encounter an assignment when walking up the control-flow graph, local semantic functionals collect the effects of assignments, and the lattice element we obtain contains only those definitions that are not killed.

The last point above serves to avoid iterating along a loop in the control flow graph. When a loop comes from an iterative control structure (a `for` or `while` loop), direct computation of the global effect is applied: we take benefit of the property of the loop, "guessing" the result in one step, which can therefore be considered as a special case of widening [7]. This fact is detailed in Section 5.5 to guarantee convergence.

The algorithm, therefore, considers 3-tuples (α, e, p), and computes the solution as a recurrence $(\alpha^{(n+1)}, e^{(n+1)}, p^{(n+1)}) = [\![\, \mathbf{s}_n \,]\!](\alpha^{(n)}, e^{(n)}, p^{(n)})$, where $\mathbf{s}_n$ is the n-th statement we encounter.

5.4 Formalization

Let us consider a single array `A`. Its range is $\mathsf{Range}(\mathtt{A})$. The iterationwise reaching definition (IRD) problem is restricted here to one statement $\mathbf{r}$ reading array `A`. The problem is defined as follows: given one instance $\langle \mathbf{r}, r \rangle$ of the read, give the set of assignment instances whose definitions may reach $\langle \mathbf{r}, r \rangle$. Obviously, the result has to be parameterized with r. If $\mathbf{s}$ is an assignment to `A`, then let $f_{\mathbf{s}} : D(\mathbf{s}) \rightarrow \mathsf{Range}(\mathtt{A})$ be the subscript function in the left-hand expression.

We now define the three features, hinted to in the previous section, which are propagated up the control flow graph.

Accesses to array elements Intuitively, set S codes which statement instance writes in which array element[2]:

$$S = \{(a, \langle \mathbf{s}, x \rangle) \ : \ a \in \mathsf{Range}(\mathtt{A}), \mathbf{s} \in E, x \in D(\mathbf{s})\}$$

This set associates to element a of array `A` an instance of a statement. Intuitively, for a given instance of a read reference, an element of this set will tell which is the reaching definition. Note that, instead of giving the "vector" of the solutions for all elements of `A`, the result is parameterized by a.

Predicate on execution This predicate is built from the conjunction of predicates copied from conditionals. They are not interpreted yet. For instance, the execution predicate for **1** and **3** in Figure 2 is just P. Let $\mathcal{T}$ be the set of all syntactically possible predicates.

Execution order In addition, we have to capture the fact that the write executes before the read. The relative order between the write instance and the read instance will be given by a predicate on the iteration vectors of the instances. E.g., we will first look for definitions in the current iteration, so a predicate will initially captures that $x = r$. (i.e., component-wise, if n is the minimum of the dimensions of x and r: $\forall i, 1 \leq i \leq n, x_i = r_i$). Similar predicates code previous iterations of parts of, or all surrounding loops. These predicates have the form $p_k(x, r) = (\forall i, 1 \leq i \leq k : x_i = r_i) \wedge (x_{k+1} < r_{k+1})$. We denote with $\mathcal{P}$ the set of such predicates on iteration vectors.

The lattice is then the set of all parts in S augmented with set $\mathcal{P}$:

$$\mathcal{C} = 2^S \times \mathcal{T} \times \mathcal{P},$$

so the information **info** attached to a node is a triple $(\alpha, e, p) \in \mathcal{C}$. **Except when predicate e is needed, this triple will be simply denoted with (α, p).**

These predicates also define the *lexicographic order* $\ll$ on vectors:

$$x \ll y \ =_{df} \ \exists k, \ 0 \leq k < n, \ p_k(x, y)$$

where n, again, is the lowest dimensionality of the two vectors: $n = \min(|x|, |y|)$.

The lexicographic order on vectors gives the order on statement instances. Indeed, two instances are executed in sequence if their iteration vectors are lexicographically ordered, or if these vectors

[2] To make equations easier to read, we often denote conjunction with a comma instead of $\wedge$.

(restricted to shared surrounding loops) are equal and the statements themselves appear in order in the program text. Formally:

$$\langle \mathbf{s}, x\rangle \prec \langle \mathbf{t}, y\rangle \;=_{df}\; (x \ll y) \vee (x[1..N_{\mathbf{s},\mathbf{t}}] = y[1..N_{\mathbf{s},\mathbf{t}}] \wedge \mathbf{s} \text{ appears before } \mathbf{t})$$

where $N_{\mathbf{s},\mathbf{t}} = \min(x, y)$.

The partial order $\sqsubseteq$ is defined as follows: $\forall(\alpha, p), (\alpha', p') \in \mathcal{C}$, we have:

$$\begin{array}{rcl}(\alpha, p) \sqsubseteq (\alpha', p') & =_{df} & \{a : \exists u, (a, u) \in \alpha\} \subseteq \{a : \exists u, (a, u) \in \alpha'\} \\ & \wedge & ((a, u) \in \alpha \wedge (a, u') \in \alpha') \Rightarrow u \preceq u'\end{array}$$

Intuitively, two elements in the lattice are ordered if all the array elements defined by the 'smaller' are also defined by the 'larger', and if the reaching definitions given by the 'larger' are executed later than the reaching definitions of the 'smaller'.

The merge operator $\sqcap$ is similar to union, except that two writes competing for the definition of the same array element are filtered according to the order $\prec$:

$$\begin{array}{rcl}(\alpha, p) \sqcap (\beta, p) & = & (\{(x, v) \;:\; \exists v,\; (x, v) \in \alpha,\; \neg\exists w,\; (x, w) \in \beta,\; p(w) = tt,\; v \prec w\} \\ & & \cup\{(x, w) \;:\; \exists w,\; (x, w) \in \beta,\; p(w) = tt,\; \neg\exists v,\; (x, v) \in \alpha,\; w \prec v\} \;\;, p) \\ & \Downarrow & \text{Since } p \text{ is associated to both } \alpha \text{ and } \beta \\ & = & (\{(x, v) \;:\; \exists v,\; (x, v) \in \alpha,\; \neg\exists w,\; (x, w) \in \beta,\;\; v \prec w\} \\ & & \cup\{(x, w) \;:\; \exists w,\; (x, w) \in \beta,\; \neg\exists v,\; (x, v) \in \alpha,\; w \prec v\} \;\;, p)\end{array}$$

Notice that the predicates of two objects merged by $\sqcap$ must have equal predicates (intuitively, this is because the merge point is at a single nesting depth).

After having defined which information must be propagated and how two states of information can be merged, we now have to define how the various statements influence this information. The corresponding local semantic functional is defined as follows:

For assignments: The semantic functional gives the elements of `A` defined by the current statement during the last possible (i.e., complying with the predicate in $\mathcal{P}$) iteration x. Let us first define

$$W(\mathbf{s}) = \{(a, \langle \mathbf{s}, x\rangle) \;:\; a \in \mathsf{Range}(\mathtt{A}),\; x \in D(\mathbf{s}), a = f_{\mathbf{s}}(x)\} \tag{2}$$

which captures the effect of $\mathbf{s}$ on array `A`. The functional is then:

$$\forall(\alpha, p) \in \mathcal{C},\; [\![\, \mathbf{s} \,]\!](\alpha, p) = (\alpha, p)\Box W(\mathbf{s}) \tag{3}$$

Operator $\Box$ adds to the set α of (array element, definition) pairs a new pair if an instance satisfying p is a reaching definition. Adding a new pair is thus only possible if the definition associated to the corresponding array element is older, or if there is no such definition yet. This operator is reminiscent from the one proposed by Barthou [2]:

$$\begin{array}{rcl}\Box & : & \mathcal{C} \rightarrow 2^S \rightarrow \mathcal{C} \\ (\alpha, p)\Box\beta & = & \left(\begin{array}{l}\{(x, v) \;:\; (x, v) \in \alpha,\; \neg\exists w,\; (x, w) \in \beta,\; p(w) = tt,\; v \prec w\} \\ \cup\{(x, w) \;:\; (x, w) \in \beta,\; p(w) = tt,\; \neg\exists v,\; (x, v) \in \alpha \cup \beta,\; p(v),\; w \prec v\}\end{array}, p\right)\end{array}$$

For statements incrementing loop counters: If $\mathbf{s}$ is the increment statement of loop counter `i`, then predicate p has to code that the value i' of `i` for writes becomes smaller than the value i for the read. In other words, the effect of $\mathbf{s}$ is to change from predicate p_k to predicate p_{k-1}:

$$\forall(\alpha, p_k) \in \mathcal{C},\; [\![\, \mathbf{s} \,]\!](\alpha, p_k) = (\alpha, p_{k-1}) \tag{4}$$

For conditionals: When entering a conditional predicated with P, P (resp. $\neg P$) is added (by conjunction) to the previous predicate in the `then` (resp. in the `else`) branch.

$$\forall(\alpha, e, p) \in \mathcal{C},\; [\![\, \mathsf{s_{then}} \,]\!](\alpha, e, p) = (\alpha, e \wedge P, p) \tag{5}$$

$$\forall(\alpha, e, p) \in \mathcal{C},\; [\![\, \mathsf{s_{else}} \,]\!](\alpha, e, p) = (\alpha, e \wedge \neg P, p) \tag{6}$$

5.5 Solving the Dataflow Equations Iteratively

We start with **info**(start) $= (\{(a, \perp) : a \in \mathsf{Range}(\mathtt{A})\}, p_0)$ and propagate up the control-flow graph.

Since our lattice of dataflow facts is of unbounded height, we have to make sure our semantic functionals never follow an infinite and monotonically increasing sequence. Doing so is here very simple: A back edge is traversed as many times as there are surrounding loops, plus 1. So it is $O(n^2)$ with n being the maximal nesting depth of the program. Before it is traversed, it means this edge is not traversed between the instances of the writes and of the reads we consider (intuitively, reads and writes are in the same iteration w.r.t. this edge). However, traversing the back edge in the analysis corresponds to traversing the edge once *or more* at the current depth of the nest. (intuitively, we consider all the write instances from previous iterations along that edge.)

This scheme is exactly the one used in [9] for structured programs, and suffers no loss in precision. Concerning unstructured programs, however, we might lose some precision. We are not able, for the moment, to quantify this loss.

6 Back to the Example

Let us consider the example in Figure 1 and compute the instancewise reaching definition for an instance (i, j) of **7**. The effects of assigments **3** and **6** are, respectively:

$$W(\mathbf{3}) = \{(a, \langle \mathbf{3}, i', j' \rangle) : a = i' + j' + 1 = i + j, 0 \leq i', j' \leq N\}$$

and:

$$W(\mathbf{6}) = \{(a, \langle \mathbf{6}, i', j' \rangle) : a = i' + j' = i + j, 0 \leq i', j' \leq N, P(i', j')\}$$

Initially, all elements of `A` have an undefined reaching write:

$$\alpha^{(0)} = \{(a, \perp), 0 \leq a \leq 2N + 1\}$$

Predicate p initially says that we look for definitions in the same iteration. I.e., the iteration vector (i', j') of the write has to be equal to iteration vector (i, j) of the read:

$$p^{(0)} \;=\; p_2 \;=\; (i' = i \wedge j' = j)$$

Walking up edge **6** gives the first iterate, denoted by $(\alpha^{(1)}, p^{(1)})$ as explained in Section 5.3. From (3) we have:

$$\begin{aligned}(\alpha^{(1)}, p^{(1)}) = [\![\, \mathbf{6} \,]\!](\alpha^{(0)}, p^{(0)}) &= (\alpha^{(0)}, p_2) \Box W(\mathbf{6}) \\ &= (\{(a, \langle \mathbf{6}, i, j \rangle) : a = i + j, 0 \leq i, j \leq N, P(i, j)\}, p_2)\end{aligned}$$

Up the fall-through edge **4**: no effect, so:

$$[\![\, \mathbf{4} \,]\!](\alpha^{(0)}, p_2) = (\alpha^{(0)}, p_2)$$

Merging at the confluence of **4** and **5**:

$$\begin{aligned}(\alpha^{(2)}, p^{(2)}) = State(n_2) &= [\![\, \mathbf{4} \,]\!](\alpha^{(0)}, p_2) \;\cup\; [\![\, \mathbf{5} \,]\!](\alpha^{(1)}, p^{(1)}) \\ &= \left(\begin{array}{l} \{(a, \langle \mathbf{6}, i, j \rangle) : a = i + j, 0 \leq a \leq 2N + 1, P(i, j)\} \\ \cup\{(a, \perp) : 0 \leq a \leq 2N + 1\} \end{array} ,\; p_2 \right)\end{aligned}$$

Up Edge **3**:

$$\begin{aligned}(\alpha^{(3)}, p^{(3)}) &= [\![\, \mathbf{3} \,]\!](State(n_2)) \\ &= (\alpha^{(2)}, p^{(2)}) \Box W(\mathbf{3}) \\ &= \left(\begin{array}{l} \{(a, \langle \mathbf{6}, i, j \rangle) : a = i + j, 0 \leq a \leq 2N + 1, P(i, j)\} \\ \cup\{(a, \perp) : 0 \leq a \leq 2N + 1\} \end{array} ,\; p_2 \right) \Box W(\mathbf{3})\end{aligned}$$

But p_2 implies that equality $i'+j'+1=i+j$ in $W(\mathbf{3})$ cannot be verified. So:

$$(\alpha^{(3)},p^{(3)}) = State(n_1) = State(n_2) = (\alpha^{(2)},p^{(2)})$$

Traversing Edge **13** change the order predicate: Applying (4):

$$[\![\,\mathbf{13}\,]\!](State(n_1)) = (\alpha^{(3)},p_1) = \left(\begin{array}{l}\{(a,\langle \mathbf{6},i,j\rangle) : a=i+j, 0\le a\le 2N+1, P(i,j)\}\\ \cup\{(a,\perp) : 0\le a\le 2N+1\}\end{array}, i'=i\wedge j'<j\right)$$

Back to node n_3:

$$(\alpha^{(4)},p^{(4)}) = State(n_3) = [\![\,\mathbf{6}\,]\!](\alpha^{(3)},p_1) = (\alpha^{(3)},p_1)\Box W(\mathbf{6})$$

Since we cannot simultaneously have $i'=i\wedge j'<j$ (from p_1) and $i'+j'=i+j$ (from $W(\mathbf{6})$), $(\alpha^{(4)},p^{(4)}) = (\alpha^{(3)},p_1)$.

Back to node n_2: since edges $\mathbf{4},\mathbf{5}$ and $\mathbf{6}$ have no effect on $(\alpha^{(4)},p^{(4)})$, and since the merge operator is just plain union, we have:

$$(\alpha^{(5)},p^{(5)}) = (\alpha^{(3)},p_1)$$

Walking up edge **3**, we take advantage of the definition of $\Box$ which requires that j' must be maximal. First, to be maximal compared with the previously found writes, we have to require that $\langle \mathbf{6},i,j\rangle$ is not executed, i.e., $\neg P(i,j)$. Second, j' must be maximal also among the new writes with $j'<j$. Hence we have $j'=j-1$ for $j>0$. Thus, we have $\{(a,\langle \mathbf{3},i,j-1\rangle) : 1\le a\le 2N+1, a=i+j, j\ge 1, \neg P(i,j)\}$. :

$$\begin{array}{rcl}(\alpha^{(5)},p^{(5)}) &=& [\![\,\mathbf{3}\,]\!](\alpha^{(3)},p_1)\\ &=& \left(\begin{array}{l}\{(a,\langle \mathbf{6},i,j\rangle) : a=i+j, 0\le a\le 2N+1, P(i,j)\}\\ \cup\{(a,\perp) : 0\le a\le 2N+1\}\end{array}, i'=i\wedge j'<j\right)\Box W(\mathbf{3})\\ &=& \left(\begin{array}{l}\{(a,\langle \mathbf{6},i,j\rangle) : a=i+j, 0\le a\le 2N+1, P(i,j)\}\\ \cup\{(a,\langle \mathbf{3},i,j-1\rangle) : 1\le a\le 2N+1, a=i+j, j\ge 1, \neg P(i,j)\}\\ \cup\{(a,\perp) : j=0, a=i\}\end{array}, i'=i\wedge j'<j\right)\end{array}$$

Up edge **12**:

$$(\alpha^{(6)},p^{(6)}) = [\![\,\mathbf{12}\,]\!](\alpha^{(5)},p^{(5)}) = (\alpha^{(5)},p_0)$$

Going back up **6**,

$$\begin{array}{rcl}(\alpha^{(7)},p^{(7)}) &=& [\![\,\mathbf{6}\,]\!](\alpha^{(6)},p^{(6)})\\ &=& \left(\begin{array}{l}\{(a,\langle \mathbf{6},i,j\rangle) : a=i+j, 0\le a\le 2N+1, P(i,j)\}\\ \cup\{(a,\langle \mathbf{3},i,j-1\rangle) : 1\le a\le 2N+1, a=i+j, j\ge 1, \neg P(i,j)\}\\ \cup\{(a,\perp) : j=0, a=i\}\end{array}, i'<i\right)\Box W(\mathbf{6})\end{array}$$

Let us consider the three subsets[3] to be compared with $W(\mathbf{6})$.

- No definition in the first subset definitely kills the effect of **6** because of uninterpreted predicate P.
- The second subset definitely kills effects of $W(\mathbf{6})$ if $j\ge 1$. For $j=0$, $\Box$ makes sure that only definitions in $W(\mathbf{6})$ that are not killed by other definitions in $W(\mathbf{6})$ are added. In this case, definition $\langle \mathbf{6},i-1,1\rangle$ may reach $\langle \mathbf{7},i,0\rangle$, $i\ge 1$.
- The definitions in the last subset are just bottoms, so they do not prevent any new definitions, so $W(\mathbf{6})$ wins again. (Notice that writes in $W(\mathbf{6})$ will not eliminate the last subset of bottoms, since instances of **6** may themselves *not* execute.)

[3] Beware that $\Box$ does not work this way, since $\max((S_1\cup S_2),S_3)$ is not equal to $\max(S_1,S_3)\cup\max(S_2,S_3)$.

We get:

$$(\alpha^{(7)}, p^{(7)}) = \left(\begin{array}{l} \{(a, \langle \mathbf{6}, i, j\rangle) : a = i + j, 0 \le a \le 2N+1, P(i,j)\} \\ \{(a, \langle \mathbf{6}, i-1, 1\rangle) : a = i, 1 \le i \le N, j = 0, P(i-1, 1)\} \\ \cup\{(a, \langle \mathbf{3}, i, j-1\rangle) : 1 \le a \le 2N+1, a = i + j, j \ge 1, \neg P(i,j)\} \\ \cup\{(a, \perp) : j = 0, a = i\} \end{array}, i' < i \right)$$

Up edge **3**, we have $(\alpha^{(8)}, p^{(8)}) = [\![\, \mathbf{3} \,]\!](\alpha^{(7)}, p^{(7)})$, which is equal to:

$$\left(\begin{array}{l} \{(a, \langle \mathbf{6}, i, j\rangle) : a = i + j, 0 \le a \le 2N+1, P(i,j)\} \\ \{(a, \langle \mathbf{6}, i-1, 1\rangle) : a = i, 1 \le i \le N, j = 0, P(i-1, 1)\} \\ \cup\{(a, \langle \mathbf{3}, i, j-1\rangle) : 1 \le a \le 2N+1, a = i + j, j \ge 1, \neg P(i,j)\} \\ \cup\{(a, \perp) : j = 0, a = i\} \end{array}, i' < i \right) \Box W(\mathbf{3}),$$

that is:

$$(\alpha^{(8)}, p^{(8)}) = \left(\begin{array}{l} \{(a, \langle \mathbf{6}, i, j\rangle) : a = i + j, 0 \le a \le 2N+1, P(i,j)\} \\ \{(a, \langle \mathbf{6}, i-1, 1\rangle) : a = i, 1 \le i \le N, j = 0, P(i-1, 1)\} \\ \cup\{(a, \langle \mathbf{3}, i, j-1\rangle) : 1 \le a \le 2N+1, a = i + j, j \ge 1, \neg P(i,j)\} \\ \cup\{(a, \langle \mathbf{3}, i-1, j\rangle) : 1 \le a \le 2N+1, a = i, i \ge 1, j = 0\} \\ \cup\{(a, \perp) : i = 0, j = 0, a = 0\} \end{array}, i' < i \right) \quad (7)$$

Notice how precise and informative this result is:

- Only one array element (`A[0]`) may be uninitialized when read. Moreover, we know this read is the very first instance of statement **7** ($i = 0, j = 0$).
- For a parametric instance $\langle \mathbf{7}, i, j\rangle$, $\alpha^{(8)}$ gives all the definition instances that may reach this use. This result has the same precision as with methods in [3, 15, 2].

7 Implementation

The analysis presented in this paper is developped on top of two compiler platforms: LooPo, developped at the University of Passau, and SUIF, from Stanford. A preliminary phase has been implemented for each of these platforms, the one for LooPo in Passau and the one for SUIF in Versailles. This phase takes the output of the parser and generates the control flow graph annotated with Omega relations.

The CFG is then the input to the analysis itself, which propagates reaching definitions as described above. This analysis phase is common to both implementations, i.e., the same files are used in the LooPo- and the SUIF-version of the analysis.

A preliminary version may be found at

`http://www.prism.uvsq/~jfc/scff/scff.html`

The LooPo version will be released soon. Please check:

`http://brahms.fmi.uni-passau.de/cl/loopo/`

Don't expect, however, to find a robust, production-quality compiler pass. The current implementation just serves as a prototype to check the analysis on simple examples.

A subtle difference between the result derived in the previous section and the output of our software is that most reaching definitions are guarded by uninterpreted function in the software. For instance, the fourth subset of $\alpha^{(8)}$ in (7) could be guarded by a predicate corresponding to cases where the predicate of instance of **6** is *false*. Therefore, the result of the software can be considered as *more precise* than (7).

8 Conclusions

In this paper we presented two frameworks for reaching definition analysis: one based on iteratively solving data-flow equations, and one based on integer linear programming techniques. We combined both techniques and, thus, got an analysis which rejoins the flexibility of the first and the precision of the second framework.

Furthermore, our analysis method scales very well to larger programs (in contrast to the techniques based on integer linear programming), mainly for two reasons:

- Using a bottom-up traversal of the control-flow graph guarantees that we first focus on possible reaching definitions "close" to the read. The order in which we traverse the control-flow graph allows us to stop as soon as we have found a write for every read we are interested in; writes which execute earlier and are rewritten, are never considered by our method.
- Our analysis is flexible enough to trade off between precision and analysis time, e.g., by ignoring loops and colleting the respective instances in super-instances as indicated in Section 5.1, or by stopping the propagation of data-flow information, adding all not yet considered writes as possible reaching definitions for still undefined array cells.

In our prototype implementation, it turned out that induction variable recognition is crucial for the precision in unstructured programs. Further experiments will show how to select a good balance between analysis time and precision, i.e., depending on the purpose of the analysis, in which parts of the program it makes sense to do the analysis as tight as possible, and in which parts an approximation is sufficient.

Acknowledgment We would like to thank Jens Knoop for several discussions that gave the initial motivation for this work [6]. Part of the sofware was implemented by Andreas Dischinger and Ivan Djelic.

References

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

[2] D. Barthou. *Array Dataflow Analysis in Presence of Non-affine Constraints.* PhD thesis, Univ. Versailles, February 1998.

[3] D. Barthou, J.-F. Collard, and P. Feautrier. Fuzzy array dataflow analysis. *Journal of Parallel and Distributed Computing*, 40:210–226, 1997.

[4] A. Cohen and J.-F. Collard. Instance-wise reaching definition analysis for recursive programs using context-free transductions. In *PACT'98*, Paris, France, October 1998.

[5] J.-F. Collard and M. Griebl. Array dataflow analysis for explicitly parallel programs. *Parallel Processing Letters*, 1997.

[6] J.-F. Collard and J. Knoop. A comparative study of reaching definitions analyses. Technical Report 1998/22, PRiSM, U. of Versailles, 1998.

[7] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In M. Bruynooghe and M. Wirshing, editors, *4th Int. Symp. on Prog. Lang. Implem. and Logic Prog (PLILP'92)*, volume 631 of *LNCS*, pages 269–295, Leuven, Belgium, August 1992.

[8] E. Duesterwald, R. Gupta, and M.-L. Soffa. A practical data flow framework for array reference analysis and its use in optimization. In *ACM SIGPLAN'93 Conf. on Prog. Lang. Design and Implementation*, pages 68–77, June 1993.

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

[10] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:309 – 317, 1977.

[11] J. Knoop, D. Koschützki, and B. Steffen. Basic–block graphs: Living dinosaurs? In *Proc. of the 7th International Conference on Compiler Construction (CC'98)*, LNCS, Lisbon, Portugal, 1998.

[12] J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *Proc. of the 4th International Conference on Compiler Construction (CC'92)*, number 641 in LNCS, Paderborn, Germany, 1992.

[13] S. S. Muchnick. *Advanced Compiler Design & Implementation.* Morgan Kaufmann, 1997.

[14] W. Pugh and D. Wonnacott. Going beyond integer programming with the omega test to eliminate false data dependences. Technical Report CS-TR-3191, U. of Maryland, December 1992.

[15] D. Wonnacott and W. Pugh. Nonlinear array dependence analysis. In *Proc. Third Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers*, 1995. Troy, New York.