Efficient chaotic iteration strategies with widenings

François Bourdoncle

DIGITAL Paris Research Laboratory
85, avenue Victor Hugo
92500 Rueil-Malmaison — France
Tel: +33 (1) 47 14 28 22

Centre de Mathématiques Appliquées
Ecole des Mines de Paris
Sophia-Antipolis
06560 Valbonne — France

bourdoncle@prl.dec.com

Abstract. Abstract interpretation is a formal method that enables the static and automatic determination of run-time properties of programs. This method uses a characterization of program invariants as least and greatest fixed points of continuous functions over complete lattices of program properties. In this paper, we study precise and efficient *chaotic iteration strategies* for computing such fixed points when lattices are of infinite height and speedup techniques, known as *widening* and *narrowing*, have to be used. These strategies are based on a *weak topological ordering* of the dependency graph of the system of semantic equations associated with the program and minimize the loss in precision due to the use of widening operators. We discuss complexity and implementation issues and give precise upper bounds on the complexity of the intraprocedural and interprocedural abstract interpretation of higher-order programs based on the structure of their control flow graph.

1 Introduction

Abstract interpretation [7, 10, 11] is a formal method that enables the static and automatic determination of run-time properties of programs, such as the range [2, 4, 5] or congruence properties [15] of integer variables, linear inequalities [9] between variables, data aliasing [2, 4, 12, 13], etc. This method is based on a characterization of programs invariants as either least or greatest fixed points of continuous functions over complete lattices, which are classically computed by iterative computations starting from either the smallest element or the largest element of the lattice. Efficient computation of extremal fixed points of functions over lattices of finite height is a classical topic [17, 19]. Unfortunately, abstract interpretation also has to deal with lattices of infinite or very large height. For instance, when the values of the integer variables of a program are coded on n bits, the lattice of intervals, which is used to compute the maximum range of these variables, is of height 2^n and iterative computations of extremal fixed points of functions over this lattice have a worst-case complexity of 2^n, which is unacceptable in practice. Speed-up techniques, known as widening and narrowing [7, 11], have been designed to determine *safe approximations* of extremal fixed points of continuous function over lattices of infinite height, non-complete lattices [9], and even complete partial

orders [3, 4]. When the control flow graph of the program being analyzed is known in advance (as is the case for intraprocedural abstract interpretation) the fixed point equation to be solved amounts to a system of equations, each equation being associated with a control point $c \in C$. In this case, widening techniques require that widening (i.e. generalization) operators be applied at each control point of a *set of widening points* W such that every cycle in the dependency graph of the system is cut by at least one widening point. Of course, it is always possible to choose $W = C$, but this leads to very poor results. In this paper, we propose efficient and precise algorithms for computing approximate fixed points of continuous functions over lattices of infinite height by an appropriate use of widening and narrowing operators.

The paper is organized as follows. In section 2, we review the classical notions of widening operators, narrowing operators and chaotic iterations. Then, in section 3, we introduce the notion of *weak topological ordering* of directed graphs, which generalizes the notion of topological ordering of directed acyclic graphs. We show that this notion is very well suited to the design of chaotic iteration strategies with widenings and give the worst-case complexity of the corresponding algorithms. In section 4, we present three algorithms for computing weak topological orderings with different price/performance ratios. In section 5, we apply the previous theoretical framework to the intraprocedural abstract interpretation of programs. Finally, in section 6, we describe a simple algorithm for the interprocedural abstract interpretation of higher-order programs (for which the control flow graph is not known in advance) and deduce its worst-case complexity from a canonical weak topological ordering of the interprocedural call graph.

2 Chaotic iterations

A central problem in the abstract interpretation of a program is to compute the least (or greatest) solution of a system of semantic equations of the form:

$$\begin{cases} x_1 &= \Phi_1(x_1, \ldots, x_n) \\ &\vdots \\ x_n &= \Phi_n(x_1, \ldots, x_n) \end{cases}$$

where each index $i \in C = [1, n]$ represents a *control point* of the program, and each function Φ_i is a continuous function from L^n to L (L being the abstract lattice of program properties) which computes the property holding at point i after one program step executed from any point leading to i. The *dependency graph* of this system is a graph with set of vertices C such that there is an edge $i \to j$ if Φ_j depends on its i-th parameter, that is, if it is possible to jump from point i to point j by executing a single program step. In most cases, this graph is thus identical to the control flow graph of the program. For the sake of simplicity, we shall suppose that point 1 is the entry point of the program and that every other point is reachable from 1.

A naive algorithm for solving this system consists in applying each equation in parallel until the vector $(x_1, \ldots, x_n)$ stabilizes, starting from the least (or greatest) element of L^n. If the lattice L is of height h, then the lattice L^n is of height $h \cdot n$ and, therefore, at most $h \cdot n^2$ equations can be applied before the solution is reached.

But this algorithm is far from optimal, since it does not follow the control flow of the program and recomputes the program property associated with every control point at each iteration step. However, since each Φ_i is continuous, and hence monotonic, any sequential algorithm *à la* Gauss-Seidel can also be used, provided that every equation is applied infinitely many times. Such algorithms are called *chaotic iteration algorithms* [8, 10], and a particular choice of the order in which the equations are applied is called a *chaotic iteration strategy*.

When the dependency graph is acyclic, an optimal and linear iteration strategy thus consists in applying the equations in any topological ordering of the set of vertices of the dependency graph, but when there are loops in the program, this method is not applicable. Furthermore, even the naive algorithm cannot be effectively applied to compute least fixed points when the height of the abstract lattice is very large or infinite, as for the lattice of intervals.

A speed-up technique, pioneered by Patrick and Radhia Cousot [7, 10, 11] consists in choosing a subset $W \subseteq C$ and replacing each equation $i \in W$ by the equation:

$$x_i \;=\; x_i \nabla \Phi_i(x_1, \ldots, x_n)$$

where "∇" is a widening operator, i.e. a safe approximation of the least upper bound such that for every increasing chain $(l_k)_{k\geq 0}$, the chain $(l'_k)_{k\geq 0}$ defined by $l'_0 = l_0$ and $l'_{k+1} = l'_k \nabla l_{k+1}$ is eventually stable.

When W is such that every cycle in the dependency graph contains at least an element of W, then any chaotic iteration strategy is guaranteed to terminate and stabilize on a safe approximation of the least fixed point (actually a post-fixed point).

Similarly, narrowing operators can be used to improve the post-fixed points determined by widening operators and to compute safe approximations of greatest fixed points.

However, since widening operators generally lead to an important loss in precision, it is essential that W be as small as possible. Unfortunately, the problem of finding a minimal set W, which happens to be a classical problem (*minimal feedback vertex set*), is a NP-complete problem[14], and since the worst-case complexity of the naive algorithm is quadratic, finding this set would be by far too costly. Hence, two distinct problems have to be solved:

- Determine a good iteration strategy, that is, an order in which to apply the equations.
- Determine a good set of widening points W.

The first problem has been addressed by many authors [6, 16, 17, 19] but, to our knowledge, no algorithm exists for finding good sets of widening points, and authors who mention widening operators use them everywhere or improperly [18].

In the next section, we introduce the notion of weak topological ordering of a directed graph and we show that this notion yields an interesting answer to both problems. In particular, and contrary to what is done by many authors, the iteration strategies we propose are guided by the structure of the dependency graph rather than dynamically selected through the use of ad-hoc data structures, such as work lists. We shall see

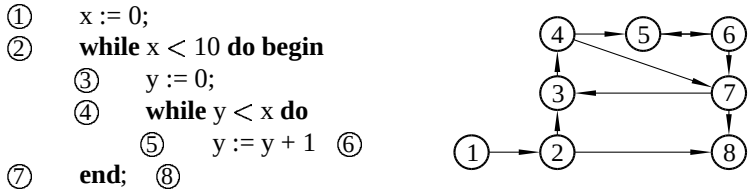

Figure 1: Intraprocedural dependency graph

that this property ensures excellent theoretical upper bounds on the complexities of the resulting algorithms.

3 Weak topological ordering

3.1 Definition

Definition 1 (Hierarchical ordering) *A hierarchical ordering of a set is a well-parenthesized permutation of this set without two consecutive "(".*

A hierarchical ordering of a set C defines a total order $\preceq$ over C. The elements between two matching parentheses are called a *component* and the first element of a component is called the *head*. We call $\omega(c)$, $c \in C$, the set of heads of the nested components containing c, and W the set of components' heads. We define the *depth* of c by $\delta(c) = |\omega(c)|$. An element has depth 0 if it is not contained in a component.

Definition 2 (Weak topological ordering) *A weak topological ordering of a directed graph (w.t.o. for short) is a hierarchical ordering of its vertices such that for every edge $u \to v$:*

$$(u \prec v \ \wedge \ v \notin \omega(u)) \ \vee \ (v \preceq u \ \wedge \ v \in \omega(u))$$

An edge $u \to v$ such that $v \preceq u$ is called a *feedback edge*. A w.t.o. of a directed graph is such that the head v of every feedback edge is the head of a component containing its tail u. For instance, a w.t.o. of the dependency graph of figure 1 is:

$$1\ 2\ (\underline{3}\ 4\ (\underline{5}\ 6)\ 7)\ 8$$

This decomposition consists of two nested components with heads 3 and 5 and, for instance, $\omega(1) = \emptyset$, $\omega(6) = \{3, 5\}$, and the feedback edge $7 \to 3$ is such that $3 \in \omega(7)$. Note that a w.t.o. without parentheses is a topological ordering and that every directed graph over a set of vertices $C = \{1, \ldots, n\}$ always has a trivial w.t.o.:

$$(1\ (2\ (3\ \cdots\ (n))))$$

with n nested components. The following theorem shows that every w.t.o. naturally defines an admissible set of widening points.

Theorem 3 (Widening points) *The set W of components' heads of a w.t.o. of the dependency graph of a system of semantic equations is an admissible set of widening points.*

Proof. Let $c_1 \to \cdots \to c_k \to c_1$, be a cycle of k distinct elements. If $k = 1$, then $c_1 \preceq c_1$ and $c_1 \to c_1$, and thus $c_1 \in \omega(c_1) \subseteq W$. If $k > 1$, then either there exist $i < j$ such that $c_j \prec c_i$ and thus $c_j \in \omega(c_i) \subseteq W$, or $c_1 \prec \cdots \prec c_k$ and thus $c_1 \prec c_k$ and $c_k \to c_1$, which shows that $c_1 \in \omega(c_k) \subseteq W$.

In either case, the cycle is thus cut by at least one widening point, which proves the theorem. ■

Of course, since widenings are costly in terms of precision, one should attempt to minimize the cardinal of W, and the trivial w.t.o. is not very interesting in this respect.

3.2 Iteration strategies

We have seen that a w.t.o. of a dependency graph is useful for determining sets of widening points, but every w.t.o. also defines at least two chaotic iteration strategies. The first strategy, called the *iterative strategy*, simply applies the equations in sequence and "stabilizes" outermost components whereas the second strategy, called the *recursive strategy*, recursively stabilizes the subcomponents of every component every time the component is stabilized.

For instance, the w.t.o. "1 2 (3 4 (5 6) 7) 8" of the graph of figure 1 yields the iterative strategy:

$$1\ 2\ [\underline{3}\ 4\ \underline{5}\ 6\ 7]^*\ 8$$

where $[\,]^*$ is the "iterate until stabilization" operator, and the recursive strategy:

$$1\ 2\ [\underline{3}\ 4\ [\underline{5}\ 6]^*\ 7]^*\ 8$$

It is easy to see that these strategies are correct, since for every vertex v of depth 0 and every edge $u \to v$, u is necessarily listed before v, i.e. $u \prec v$, and the value of u used in the computation of v already has its final value, which implies that it is not necessary to iterate over the vertices of depth 0. Note that this idea forms the heart of the method described in Jones [16], where the strongly connected components are listed in topological order (c.f. section 4.3). However, our approach is superior in that we also give algorithms for computing fixed points of strongly connected systems of semantic equations instead of using the brute-force, $O(n^2)$ algorithm.

Theorem 4 (Iterative strategy) *For the iterative strategy, the stabilization of an outermost component of a w.t.o. can be detected by the stabilization of its widening points.*

Proof. Let us suppose that the w.t.o. consists of a single outermost component. We are going to show that after applying the equations in sequence, either one (at least) of the the semantic values associated to a widening point has increased, or none has changed.

Suppose that the contrary holds, and that the value associated to a vertex $v \notin W$ has changed. Then by definition of a w.t.o., every edge $u \to v$ is such that $u \prec v$. Thus, there exists at least one vertex $u \prec v$ whose value has changed since the last iteration.

But since $u \notin W$ by hypothesis, the inductive application of the same argument to u shows that the head of the component has changed, which is absurd. ■

Theorem 5 (Recursive strategy) *For the recursive strategy, the stabilization of a component of a w.t.o. can be detected by the stabilization of its head.*

Proof. Let us suppose that the w.t.o. consists of a single outermost component and that, after applying the equations and stabilizing the sub-components, the value associated to the head of the component remains unchanged after recomputation. Then the argument used to prove the fact that no iteration is necessary over the vertices of depth 0 shows that the values associated to the vertices within the component won't change when the equations are applied once more. Therefore, the stabilization of the component's head imply the stabilization of the entire component. ■

These two theorems show that the iterative and recursive strategies minimize the number of comparisons between elements of the lattice L of program properties, which can be very useful when this test is very costly, as with the abstract interpretation of functional or logic programs for instance. Also, note that even though the ordering $\sqsubseteq$ over L is not explicitly used by the algorithm, it can be used to improve precision and force the convergence of the computation to the first post-fixed point when the widening operator is not *stable* [3, 4], i.e. does not satisfy:

$$\forall x, y \in L \ : \ y \sqsubseteq x \implies x \nabla y = x$$

The following theorem gives an upper bound on the complexity of each strategy.

Theorem 6 (Complexity) *When the lattice L is of finite height h, or when the increasing chains built by the widening operator are at most of length h, the maximum complexity of the iterative iteration strategy for a strongly connected graph is:*

$$h \cdot |C| \cdot |W|$$

and the maximum complexity of the recursive iteration strategy is:

$$h \cdot \sum_{c \in C} \delta(c)$$

Proof. Since theorem 4 shows that each iteration yields a strictly greater element over the lattice $L^{|W|}$ of height $h \cdot |W|$, the first result is trivial. The second result is easily proved inductively by showing that each equation (i.e. vertex) of depth k is applied at most $h \cdot k$ and each sub-component of depth $k+1$ is stabilized at most $h \cdot k$ times. A detailed proof can be found in Bourdoncle [4]. ■

The complexity of the recursive iteration strategy is thus a linear function of the sum of the individual depths of the vertices of the graph. It is interesting to remark that since $\delta(c) \leq |W|$ for every vertex c, the upper bound of the recursive iteration strategy is always better than that of the iterative strategy. Practice shows that the recursive strategy is indeed almost always better than the iterative strategy. Furthermore, it is

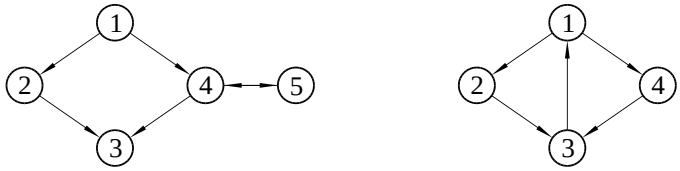

Figure 2: Flow graphs

clear that the worst-case of each strategy for a program of size n, which is obtained with the trivial w.t.o., is $h \cdot n^2$ for the iterative strategy and:

$$h \cdot (1 + 2 + \cdots + n) \quad = \quad \frac{h \cdot n \cdot (n+1)}{2}$$

for the recursive strategy. These results show that not only does the number of widening points impact on the precision of the fixed point computation, but it also impacts on the cost of the analysis. An essential goal is thus to minimize the number of widening points as well as the sum of the individual depths of the graph's vertices.

The following section presents three algorithms with different price/performance ratios that can be used to compute weak topological orderings of directed graphs and relate them to previous works.

4 Algorithms

4.1 Depth-first numbering

A first idea for building a non-trivial w.t.o. of a graph is to use a depth-first numbering of this graph, which can be obtained in linear time, and 1) open a parenthesis before every head b of edges $a \rightarrow b$ whose tail a has a greater number than b, 2) close all the parentheses after the last vertex. For instance, this algorithm would yield the following result on the graph of figure 1:

$$1\ 2\ (\underline{3}\ 4\ (\underline{5}\ 6\ 7\ 8))$$

which is better than the trivial w.t.o. but not as good as the "optimal" ordering:

$$1\ 2\ (\underline{3}\ 4\ (\underline{5}\ 6)\ 7)\ 8$$

Also, note that this algorithm has a tendency to overestimate the number of widening points. For instance, the graph on the left of figure 2 would yield the following w.t.o.:

$$1\ 2\ (\underline{3}\ (\underline{4}\ 5))$$

with two widening points, although the graph has a single cycle. However, it is shown in Bourdoncle [4] that the set of heads of the *retreating edges* (c.f. [1], p. 661) of the depth-first spanning tree, i.e. edges $a \rightarrow b$ such that b is an ancestor of a in the tree, is a smaller admissible set of widening points. For the graph on the left of figure 2, the only retreating edge is $5 \rightarrow 4$ and $\{4\}$ is thus a set of widening points.

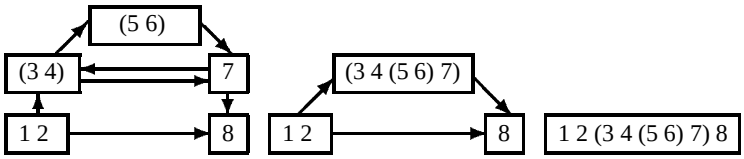

Figure 3: Limit flow graph

Consequently, the w.t.o. decomposition can be used as the basis of the iteration strategy whereas widening points can be detected through the retreating edges of the depth-first spanning tree (which can be easily found by using a stack of "currently visited vertices" during the depth-first visit of the graph).

In spite of its drawbacks, the advantage of this algorithm is that it is very simple and incremental, and can be applied even when the graph is not known in advance as for the abstract interpretation of higher-order programs (c.f. section 6).

4.2 Reducible graphs

When the dependency graph is *reducible* [1], as is always the case for structured programming languages without goto statement, it has been suggested [11] to choose as widening points the heads of the *intervals* of the graph which are also the head of a back-edge.

This idea can be pushed a step further to build a w.t.o. of the graph and, hence, an iteration strategy. The idea consists in computing the *limit flow graph* obtained by iteratively collapsing the graph into its *intervals* (c.f. [1], p. 666). When the graph is reducible, this process is guaranteed to converge to a limit graph reduced to a single vertex containing all the vertices of the original graph. This process is illustrated figure 3 for the reducible graph of figure 1 and gives the following result:

$$1\ 2\ (\underline{3}\ 4\ (\underline{5}\ 6)\ 7)\ 8$$

Note that an interval I is parenthesized only when there exists a feedback edge $u \to v$ from a vertex $u \in I$ to the *header* v of I.

To prove that the resulting decomposition of the graph is a w.t.o., the only thing to show is that the head v of every feedback edge $u \to v$ is the header of an interval containing u.

First, remark that the property trivially holds when u and v belong to the same interval of level 1. So let K denote the interval who first "merged" the two distinct intervals I containing u and J containing v during the computation of the limit flow graph. It is known (c.f. [1], prop. 2, p. 669) that v is necessarily the header of J. Now, if J is the first "vertex" of K (as for the interval $(3\ 4)$ of figure 3) then v is the head of K and the property holds. Otherwise, I and J are proper "vertices" of K, J is listed before I (since $u \to v$ is a feedback edge) and there is an edge from I to J, which is incompatible with the fact that J has been added to K before I.

This algorithm has a worst-case complexity equal to $O(\delta \cdot \varepsilon \cdot \alpha(\varepsilon))$ where δ is the *depth* of the graph, ε is the number of edges, and α is the inverse of Ackerman's function

(which is nearly constant).

Finally, note that the iteration strategies built using this algorithm are similar to the ones described in Burke [6] in a data-flow analysis framework.

4.3 Strongly connected components

Reducible graphs have been extensively studied in the literature but, unfortunately, interprocedural dependency graphs are not reducible in general. For instance, the graph of figure 5, which is an unfolded version of the graph of function "Fact", is not reducible since the head $6'$ of the back-edge $5' \rightarrow 6'$ does not dominate its tail, i.e. there is a path from 1 to $5'$ that does not go through $6'$.

The base of the algorithm we propose in this case is an algorithm due to R.E. Tarjan [20] to compute in linear time the strongly connected components of a directed graph. Since Tarjan's algorithm computes a list of (possibly trivial) strongly connected components in topological order, the basic idea of the algorithm, given figure 4, is to recursively apply Tarjan's algorithm to each non-trivial component after having removed its head b and all the back-edges of the form $a \rightarrow b$. Note that "::" denotes the list constructor operator.

Theorem 7 *The algorithm of figure 4 computes a w.t.o. of any directed graph.*

The proof is omitted here for the sake of brevity and can be found in Bourdoncle [4]. Note that when the graph is reducible, the interval-based algorithm can give better results. For instance, depending on the order in which the graph's vertices are visited, the algorithm of figure 4 gives one of the following results for the graph on the right of figure 2:

$$(\underline{1}\ 2\ 3\ 4)$$
$$(\underline{1}\ 4\ 3\ 2)$$

whereas the interval-based algorithm gives the optimum result:

$$(\underline{1}\ 2\ 4\ 3)$$

However, excellent decompositions are obtained for non-reducible graphs, such as the graph of figure 5:

$$(\underline{1}\ 4\ 1'\ 4')\ 2'\ 3'\ 2\ 3\ (\underline{6}\ 5'\ 6'\ 5)$$
$$(\underline{1}\ 4\ 1'\ 4')\ 2\ 3\ 2'\ 3'\ (\underline{6'}\ 5\ 6\ 5')$$

The worst-case complexity of this algorithm is $\delta \cdot \varepsilon$, where δ is the maximum depth of the graph's vertices and ε is the number of edges. Hence, this algorithm does not cost more than the fixed point computation itself, and its complexity is a linear function of the intrinsic complexity of the graph.

Finally, note that to our knowledge, other hierarchical decompositions of directed graphs into strongly connected components [21] are not weak topological orderings and, hence, cannot be used to perform chaotic iteration strategies with widenings.

```
function Partition
    var vertex, partition
begin
    foreach vertex ∈ vertices_G do
        DFN[vertex] ← 0
    NUM ← 0
    partition ← nil
    Visit(root_G, partition)
    return partition
end
function Component(in vertex)
    var succ, partition
begin
    partition ← nil
    foreach succ ∈ succ_G[vertex] do
        if DFN[succ] = 0 then
            Visit(succ, partition)
    return (vertex :: partition)
end
function Visit(in vertex, inout partition)
    var head, min, succ, element, loop
begin
    push(vertex)
    head ← DFN[vertex] ← NUM ← NUM + 1
    loop ← false
    foreach succ ∈ succ_G[vertex] do
        if DFN[succ] = 0 then
            min ← Visit(succ, partition)
        else min ← DFN[succ]
        if min ≤ head then
            head ← min
            loop ← true
    if head = DFN[vertex] then
        DFN[vertex] ← +∞
        pop(element)
        if loop then
            while element ≠ vertex do
                DFN[element] ← 0
                pop(element)
            partition ← Component(vertex) :: partition
        else partition ← vertex :: partition
    return head
end
```

Figure 4: Hierarchical decomposition of a directed graph into strongly connected components and subcomponents.

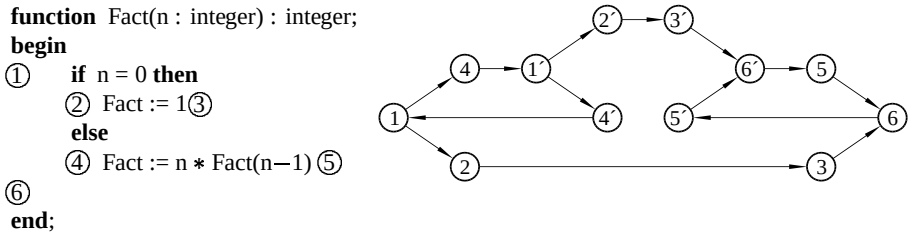

Figure 5: Interprocedural dependency graph

5 Intraprocedural abstract interpretation

The algorithm of figure 4 is thus directly applicable to intraprocedural abstract interpretation, and has been implemented in the abstract debugger *Syntox* [2, 4, 5]. The advantage of using a w.t.o. is that it is computed once at the beginning of the analysis and that the resulting chaotic iteration strategy minimizes the use of widening operators as well as the number of tests needed to detect the stabilization of iterative computations. Furthermore, the algorithm has a predictable worst-case complexity, which is n for a program without loops and:

$$h \cdot p \cdot (n - \frac{p-1}{2})$$

for a program with p nested loops.

6 Interprocedural abstract interpretation

The method of the previous section is applicable to the interprocedural abstract interpretation of first-order program but cannot be used for higher-order programs for which the dependency graph is not known in advance. However, the incremental algorithm of section 4.1 can be used, and since the w.t.o. determined by this algorithm has the generic form:

$$a_1 \cdots (a_{k_1} \cdots (a_{k_2} \cdots (a_{k_3} \cdots)))$$

the iterative strategy seems easier to implement and corresponds to a straightforward depth-first execution of the program. Furthermore, if we note that each point a_{k_i} is necessarily either an entry point of a procedure, the head of an intraprocedural loop or the return point of a procedure call, we have the following property on the worst-case complexity of the interprocedural abstract interpretation of a program.

Theorem 8 *If a program has p procedures, n control points, c procedure calls and l intraprocedural loops, then the abstract interpretation over a lattice of height h using the iterative iteration strategy has a worst-case complexity of:*

$$h \cdot (p + c + l) \cdot n = \rho \cdot h \cdot n^2$$

where $\rho \leq 1$ *is sum* $c/n + l/n$ *of the densities of procedure calls and intraprocedural loops and of the inverse of the average size* n/p *of procedures. Furthermore, if each*

procedure of a higher-order program has at most m procedural formal parameters, then the computation of the interprocedural call graph of the program has a worst-case complexity of $\rho \cdot m \cdot p \cdot n^2$.

Note that, as hinted in section 4.1, the depth-first visit of the interprocedural dependency graph allows for the on-line determination of a better set of widening points than $\{a_{k_1}, a_{k_2}, \ldots\}$ and, in practice, it is sufficient to use widening operators at the head of intraprocedural loops and at the entry and exit points of formally recursive procedures, as opposed to the return points of procedure calls ([4], p. 52).

Note that the algorithm proposed by Le Charlier et al. [18], which is in fact a particular implementation of *basic functional partitioning* [3], uses widening operators at the entry point of every logic predicate but not at the exit point. This is justified in their context, since they use *noetherian* domains with no infinite strictly increasing chain, but their algorithm can loop when the abstract domain is of infinite height.

7 Conclusion

In this paper, we have addressed the problem of the efficient computation of least and greatest fixed points of continuous functions over lattices of infinite height using widening and narrowing operators.

We have introduced the notion of weak topological ordering of directed graphs and shown how this notion can be used to determine admissible sets of widening points and design efficient chaotic iteration strategies.

We have given several algorithms with different price/performance ratios to compute weak topological orderings of directed graphs and shown how to apply them to the intraprocedural and interprocedural abstract interpretation of higher-order languages.

Further work will be to design an incremental version of the algorithm of figure 4 to handle the interprocedural abstract interpretation of higher-order programs more efficiently.

References

[1] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman: "Compilers — Principles, Techniques and Tools", Addison-Wesley Publishing Company (1986)

[2] François Bourdoncle: "Interprocedural Abstract Interpretation of Block Structured Languages with Nested Procedures, Aliasing and Recursivity", *Proc. of the International Workshop PLILP'90*, Lecture Notes in Computer Science 456, Springer-Verlag (1990)

[3] François Bourdoncle: "Abstract Interpretation By Dynamic Partitioning", *Journal of Functional Programming*, Vol. 2, No. 4 (1992)

[4] François Bourdoncle: "Sémantiques des langages impératifs d'ordre supérieur et interprétation abstraite", *Ph.D. dissertation*, Ecole Polytechnique (1992)

[5] François Bourdoncle: "Abstract Debugging of Higher-Order Imperative Languages", *Proc. of SIGPLAN '93 Conference on Programming Language Design and Implementation* (1993)

[6] Michael Burke: "An Interval-Based Approach to Exhaustive and Incremental Interprocedural Data-Flow Analysis", *ACM Transactions on Programming Languages and Systems*, Vol. 12, Num. 3 (1990) 341–395

[7] Patrick and Radhia Cousot: "Abstract Interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints", *Proc. of the 4th ACM Symp. on POPL* (1977) 238–252

[8] Patrick Cousot: "Asynchronous iterative methods for solving a fixpoint system of monotone equations", Research Report IMAG-RR-88, Université Scientifique et Médicale de Grenoble (1977)

[9] Patrick Cousot and Nicolas Halbwachs: "Automatic discovery of linear constraints among variables of a program", *Proc. of the 5th ACM Symp. on POPL* (1978) 84–97

[10] Patrick Cousot: "Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis. Analyse sémantique de programmes", *Ph.D. dissertation*, Université Scientifique et Médicale de Grenoble (1978)

[11] Patrick Cousot: "Semantic foundations of program analysis", in Muchnick and Jones Eds., *Program Flow Analysis, Theory and Applications*, Prentice-Hall (1981) 303–343

[12] Alain Deutsch: "On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications", *Proc. of the 17th ACM Symp. on POPL* (1990)

[13] Alain Deutsch: "A Storeless Model of Aliasing and its Abstractions using Finite Representations of Right-Regular Equivalence Relations", *Proc. of the IEEE'92 International Conference on Computer Languages*, IEEE Press (1992)

[14] Michael. R. Garey and David S. Johnson: "Computers and Intractability: A Guide to the Theory of NP-completeness", W.H. Freeman and Company (1979)

[15] Philippe Granger: "Static analysis of arithmetical congruences", *International Journal of Computer Mathematics* (1989) 165–190

[16] Larry G. Jones: "Efficient Evaluation of Circular Attribute Grammars", *ACM Transactions on Programming Languages and Systems*, Vol. 12, Num. 3 (1990) 429–462

[17] B. Le Charlier, K. Musumbu and P. Van Hentenryck: "A generic abstract interpretation algorithm and its complexity analysis", in K. Furukawa editors, *Proc. of the Eight International Conference on Logic Programming*, MIT Press (1991) 64–78

[18] B. Le Charlier and P. Van Hentenryck: “A universal top-down fixpoint algorithm”, Technical Report 92-1, Institute of Computer Science, University of Namur, Belgium (1992)

[19] R.A. O’Keefe: “Finite fixed-point problems”, in J.-L. Lassez editor, *Proc. of the Fourth International Conference of Logic Programming*, MIT Press (1987) 729–743

[20] R.E. Tarjan: “Depth-first search and linear graph algorithms”, *SIAM J. Comput.*, 1 (1972) 146–160

[21] R.E. Tarjan: “An Improved Algorithm for Hierarchical Clustering Using Strong Components”, *Information Processing Letters 17*, Elsevier Science Publishers B.V. (1983) 37–41