Contents

Evaluation of three iterative fixpoint algorithms on randomly generated input

Niels Jørgensen

Computer Science Dept., Roskilde University
P.O. Box 260, DK-4000 Roskilde, Denmark
`nielsj@dat.ruc.dk`
phone: + 45 46 75 77 11
fax: + 45 46 75 42 01

Abstract

Three iterative fixpoint algorithms are evaluated experimentally. The input is a benchmark of randomly generated systems of equations over boolean functions. The randomizer is designed to target certain presumed weaknesses of the algorithms, rather than generate equations that are presentative of real world input. The "winner" is perhaps the simplest of the algorithms. It uses a variant of naive Kleene iteration and is found to perform surprisingly well even on the benchmark's largest problems comprising 1000 simultaneous equations. The second algorithm is a smarter algorithm using a principle referred to in the paper as *local stabilization* and which is based on an algorithm proposed by Le Charlier *et al.* [10]. The test indicates that the algorithm may perform a considerable amount of redundant work when the dependency graph associated with the fixpoint problem contains strongly connected components that are very large. Finally, the paper presents a new variant of a chaotic worklist algorithm that incorporates depth-first evaluation. It performs well on the given input, remaining on the average within 25 % above the optimal number of iterations. The difference between the latter two algorithms is discussed, on the basis of their definition within a common setting.

1 Introduction

There is an expectation that fixpoint algorithms will perform much better in average cases than in worst cases. But what is an average case — a typical input for a fixpoint algorithm ? In view of the fact that it is very hard to conceive of a representative benchmark of fixpoint problems, the experimental evaluation of fixpoint algorithms may perhaps benefit from using a benchmark of *randomized* input, as a supplement to real world input.

If we think of fixpoint computation in a context of optimizing compilers, an *empirical* approach involves a series of critical choices of data: One must choose a programming language, a benchmark of input programs, and a fixpoint-based abstract semantics. An example of a thorough, empirical evaluation is Le Charlier and Van Hentenryck's study [12] of the performance of their algorithm on a suite of 11 Prolog programs, using a large and very sophisticated abstract domain. The description of the abstract semantics chosen for the experiment takes up a full 22 pages. This indicates that it is difficult to cross the barriers between specific languages and analyses, and to design experiments that are are to reproduce.

In the context of deductive databases, the choice of data for evaluating algorithms that solve the (fixpoint) problem of finding answers to recursive queries is somewhat less arbitrary: There is no choice of abstract domains and operators, since the algorithms work directly on the query and the database's rules and facts. Bancilhon and Ramakrishnan's *Amateur's Introduction* ... [1] is a combined empirical and analytical evaluation, where 11 algorithms are tested on four sets of actual queries and rules and an analytical representation of the database facts. (The study finds no one algorithm to be generally superior to the others). However, even when the domain and primitive operations involved in the fixpoint computation is fixed, it seems very difficult to provide a complete benchmark of data, as acknowledged in [1].

For comparison, one may consider Sedgewick's extensive analysis of Quicksort [16] where problem size is the number of (randomly ordered) keys to be sorted. Certainly, there is no single parameter for measuring the size of a fixpoint problem. However, there is perhaps a common understanding of what might be a collection of key parameters. In the context of Prolog analysis, they would include the number of clauses, the proportion of recursive vs. non-recursive procedures, and others. This implies that in lack of a truly representative benchmark, we can at least describe the *profile* of a benchmark, in terms of certain parameters that are regarded as important.

As a supplement to experiments involving empirical data, an experiment using randomly generated fixpoint problems provides

- The ability to create input problems with specific profiles.
- The ability to generate very large input problems, in as large numbers as wanted.
- An easily reproducible test.

The randomly generated equation systems comprise from 100 to 1000 simultaneous equations of the form

$$f_i(x_1, x_2) = e_i \tag{1}$$

The functions f_i (referred to as user-defined functions) are interpreted as boolean functions. The induced fixpoint problem corresponds to finding the least solution with respect to the f_i's. The right hand sides e_i contain (possibly nested) calls to the user-defined functions, as well as calls to primitive boolean functions. A profile of the random benchmark is given in terms of the size of the minimal function graph, the number of strongly connected components in the dependency graph, the size of the components, and other parameters. The reproduction of the experiment is facilitated by the choice of the primitive, boolean domain, and a simple design for the random equation generator.

Three algorithms are subjected to evaluation: The first is an algorithm referred to here as the Kleene algorithm, with reference to the ascending Kleene sequence. The second is called the Local Stabilization algorithm. It is based on the algorithm defined by Le Charlier *et al.* [12] for abstract interpretation of Prolog. The third is a new worklist algorithm. It is similar to an algorithm proposed by this author in [9] and has the same low worst-case performance. The difference is that it incorporates a (straightforward) mechanism for depth-first evaluation, which is assumed to improve average case performance.

None of the three algorithms in the test are uniformly better than the others. In contrast, a comparative study of iterative fixpoint algorithms by Horwitz *et al.* [8] finds an iteration method called "Priority Strongly Connected Component-iteration" to be preferable in general. This is a worklist method that visits strongly connected components in topological order. (In being a worklist method, it resembles the worklist algorithm evaluated in this paper, while in evaluating in topological order it resembles the Local Stabilization algorithm). However, the result of [8] does not apply to equation systems of the form (1), because the equations relate functions rather than primitive variables. In that case the strongly connected components of the dependency graph cannot be computed prior to the fixpoint iteration. Algorithms for solving (1) are more difficult to analyse theoretically, and seem to call for experimental evaluation even more than those studied in [8]. Fixpoint problems that involve functions rather than primitive variables typically arise in analysis of logic programs and functional programs, as well as in some variants of inter-procedural analysis of imperative languages.

The profile of the randomized input has been tailored to provide *hard* problems, in order to test a collection of hypotheses about some possible weaknesses of the algorithms. The paper's contributions to the understanding of the fixpoint algorithms subjected to evaluation can be summarized as follows:

- Some evidence to contradict a hypothesis that the naive Kleene algorithm will perform poorly on a fixpoint problem involving a very large number of equations.
- Some evidence to support a hypothesis that the Local Stabilization algorithm will do a substantial amount of redundant work on equation systems where the dependency graph contains strongly connected components that are very large.
- Some evidence to contradict a hypothesis that the Chaotic Worklist algorithm will do a substantial amount of redundant work on elements that are not in the minimal function graph, and on elements that belong to strongly connected components that are evaluated too early.

The paper's discussion of the three algorithms in the test — based on their definition within a uniform setting — may also be of value in the clarification of the inter-relationship between a number of algorithmic techniques available for fixpoint computation.

There are a number of interesting algorithms that were not included in the test. Algorithms that solve the same fixpoint problem as the one considered in the test include Pending analysis [17]. Another early approach is the Frontiers algorithm [14] originally designed for the two point domain (*i.e.*, for strictness analysis of first order functions). It differs from the three algorithms tested in that it computes a global fixpoint, and appears to be difficult to modify so as to disregard elements not relevant for the computation of a local fixpoint.

In more recent work, Bourdoncle proposed [2] that iteration strategies be based on weak topological orderings of the dependency graph. For fixpoint problems involving primitive values, such an ordering can be used to find a good evalution order within each strongly connected component; and this can be combined with a worklist approach, as tested empirically with promising results by Schön [15]. For fixpoint problems involving functions the approach requires that a weak topological ordering be recomputed during interation. The ensuing trade-off (the cost of the recomputation vs. the benefits of the optimization) is still open for investigation. Chen's so-called guided entailment algorithm [4] is designed for functional problems. It resembles the Worklist and Local Stabilization algorithm of the present test in relying on a dependency graph. An important point of difference is that the guided entailment algorithm is based on transforming a functional problem into a primitive problem, implying that the solution found is a conservative approximation of the least fixpoint rather than the fixpoint itself.

The remainder of the paper is organized as follows: Section 2 defines a syntax of fixpoint-equations, and the notions of induced fixpoint and minimal function graph. Section 3 explains some common properties of the three algorithms in the test. Sections 4–6 define each particular algorithm, and pinpoint weaknesses to be targeted in the test. Section 7 describes the random procedure for generating equations, and discusses whether the sample actually generated is appropriate for the test. Finally, Section 8 presents the test results, and Section 9 concludes.

2 The fixpoint problem

The fixpoint problems of the test are given in the form of simultaneous equations.

$\mathcal{B}$ is the lattice $\{false, true\}$ with $false \sqsubseteq true$. A system of q equations induces the least fixpoint of a functional in the function space $S = D \rightarrow D$ where $D = \{1, \ldots, q\} \rightarrow \mathcal{B}^a \rightarrow \mathcal{B}$. (A fixed arity a is assumed). Function spaces and cartesian products are ordered "pointwise". Since D is a complete lattice, the least fixpoint $fix\ F$ exists for any monotonic $F \in S$.

Definition 1 (Global fixpoint induced by a system of equations)
Suppose that for each j, b_j is a base function symbol which is interpreted as a monotonic, boolean function $base_j : \mathcal{B}^a \rightarrow \mathcal{B}$. Let the language L of expressions be defined by

$$\begin{array}{rcll} e & ::= & x_1 \mid \cdots \mid x_a & \text{(primitive parameters)} \\ & \mid & b_j(e_1, \ldots, e_a) & \text{(base function symbols)} \\ & \mid & f_i(e_1, \ldots, e_a) & \text{(functional parameters, } 1 \leq i \leq q\text{)} \end{array}$$

The evaluation function $\mathbf{E} : L \rightarrow D \rightarrow \mathcal{B}^a \rightarrow \mathcal{B}$ is defined by:

$$
\begin{array}{lcl}
\mathbf{E}\ [\![x_i]\!]\ \overline{\phi}\ \overline{\alpha} & = & \alpha_i \ \ \text{(the } i\text{'th element of } \overline{\alpha}\text{)} \\
\mathbf{E}\ [\![b_j(e_1 \ldots, e_a)]\!]\ \overline{\phi}\ \overline{\alpha} & = & base_j(\mathbf{E}[\![e_1]\!]\overline{\phi}\overline{\alpha}, \ldots, \mathbf{E}[\![e_a]\!]\overline{\phi}\overline{\alpha}) \\
\mathbf{E}\ [\![f_i(e_1 \ldots, e_a)]\!]\ \overline{\phi}\ \overline{\alpha} & = & \phi_i(\mathbf{E}[\![e_1]\!]\overline{\phi}\overline{\alpha}, \ldots, \mathbf{E}[\![e_a]\!]\overline{\phi}\overline{\alpha})
\end{array}
$$

Let an equation system

$$f_i(x_1, \ldots, x_a) = e_i \quad (1 \leq i \leq q)$$

be given, and let F be the functional that maps $\overline{\phi}$ to $\langle \lambda\overline{\alpha}.\mathbf{E}[\![e_1]\!]\overline{\phi}\overline{\alpha}, \ldots, \lambda\overline{\alpha}.\mathbf{E}[\![e_q]\!]\overline{\phi}\overline{\alpha}\rangle$. Then the global fixpoint induced by the equation system is the least fixpoint $fixF$.
□

Sometimes $\overline{x}$ is used instead of $\langle x_1, \ldots, x_a\rangle$, *etc*. The values that the parameters $\overline{x}$ take on are usually represented by $\overline{\alpha}$ and $\overline{\beta}$, and those that $\overline{f}$ take on by $\overline{\phi}$ and $\overline{\psi}$. In examples we sometimes replace the base function symbols by the usual symbols for conjunction, *etc*.

The notion of a local fixpoint is defined in terms of a "neededness" function. The function takes an L-expression, and collects the recursive calls to the user defined functions. The functional parameter $\overline{\phi} = \langle \phi_1, \ldots, \phi_q\rangle$ is a parameter to the neededness function. This reflects that during fixpoint iteration, the recursive calls depend, in general, on the current fixpoint approximation.

Definition 2 (Neededness function)
The neededness function $\mathbf{N} : L \to D \to \mathcal{B}^a \to \wp(\{1, \ldots, q\} \times \mathcal{B}^a)$ is defined by:

$$
\begin{array}{lcl}
\mathbf{N}\ [\![x_i]\!]\ \overline{\phi}\ \overline{\alpha} & = & \emptyset \\
\mathbf{N}\ [\![b_j(e_1, \ldots, e_a)]\!]\ \overline{\phi}\ \overline{\alpha} & = & \mathbf{N}[\![e_1]\!]\overline{\phi}\overline{\alpha} \cup \ldots \cup \mathbf{N}[\![e_a]\!]\overline{\phi}\overline{\alpha} \\
\mathbf{N}\ [\![f_i(e_1, \ldots, e_a)]\!]\ \overline{\phi}\ \overline{\alpha} & = & \{\langle i, \langle \mathbf{E}[\![e_1]\!]\overline{\phi}\overline{\alpha}, \ldots, \mathbf{E}[\![e_a]\!]\overline{\phi}\overline{\alpha}\rangle\rangle\} \\
 & & \quad \cup\ \mathbf{N}[\![e_1]\!]\overline{\phi}\overline{\alpha} \cup \ldots \cup \mathbf{N}[\![e_a]\!]\overline{\phi}\overline{\alpha}
\end{array}
$$

□

Since the first argument in a pair $\langle i, \overline{\alpha}\rangle$ represents the symbol f_i, for clarity we usually write $\langle f_i, \overline{\alpha}\rangle$ for the pair.

The notion of derived calls is useful in applications to program analysis where we want to known how a program's procedures (or predicates, or functions) are called, assuming a description of the program's input.

Definition 3 (Local fixpoint)
Suppose $\overline{\phi}$ is the global fixpoint induced by a given equation system. Let $Calls$ be a set of initial calls, *i.e.*, a set of pairs $\langle f_i, \overline{\alpha} \rangle$. Then the derived calls are those in the set

$$fix(\lambda S.Calls \cup \bigcup_{\langle f_i, \overline{\alpha} \rangle \in S} \mathbf{N}\, [\![e_i]\!]\, \overline{\phi}\, \overline{\alpha})$$

A local fixpoint (with respect to $Calls$) is a function which is identical to $\overline{\phi}$ for all derived calls.

For given $Calls$ the mimimal function graph (cf. Mycroft and Jones [11]) is the set

$$\{ \langle f_i, \overline{\alpha}, \psi_i(\overline{\alpha}) \rangle \mid \langle f_i, \overline{\alpha} \rangle \text{ is a derived call} \}$$

where $\overline{\psi}$ is a local fixpoint. The size of the minimal function graph is the number of elements in the set. $\Box$

For a given equation system and fixpoint approximation $\overline{\phi}$, the dependency graph $\langle N, E \rangle$ is defined as follows. The set of nodes N is the set $\{1, \ldots, q\} \times \mathcal{B}^a$, *i.e.*, the set of all function-value pairs. There is an edge $\langle f_i, \overline{\alpha} \rangle \leftarrow \langle f_j, \overline{\beta} \rangle \in E$ if

$$\langle f_j, \overline{\beta} \rangle \in \mathbf{N}\, [\![e_i]\!] \overline{\phi}\, \overline{\alpha}$$

in which case $\langle f_i, \overline{\alpha} \rangle$ *needs* $\langle f_j, \overline{\beta} \rangle$.

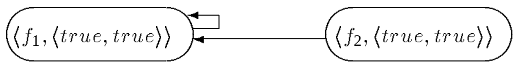
By a slight abuse of terminology, the notion of a minimal function graph is also used to refer to the dependency graph that connects the derived calls.

As an example, consider the equation system of Figure 1. The right hand sides have the needs (for given $\overline{\phi}$)

$$\begin{array}{lcl} \mathbf{N}\, [\![f_1(true, f_2(x_1, x_2))]\!]\, \overline{\phi}\, \overline{\alpha} & = & \{ \langle f_1, \langle true, \phi_2(\alpha_1, \alpha_2) \rangle \rangle, \langle f_2, \langle \alpha_1, \alpha_1 \rangle \rangle \} \\ \mathbf{N}\, [\![x_1 \wedge x_2]\!]\, \overline{\phi}\, \overline{\alpha} & = & \emptyset \end{array}$$

Note that the first need associated with the first equation depends on $\overline{\phi}$, because of the nested call to a user defined function. For $\overline{\alpha} = \langle true, true \rangle$, if $\overline{\phi}$ is a local fixpoint, the need is $\langle f_1, \langle true, true \rangle \rangle$, while if $\overline{\phi}$ is the bottom element of $\{1, 2\} \rightarrow \mathcal{B}^2 \rightarrow \mathcal{B}$, the need is $\langle f_1, \langle true, false \rangle \rangle$.

The minimal function graph associated with the local fixpoint problem is the following graph containing two nodes and two edges.



Finally, recall that two nodes N_1 and N_2 of a directed graph are strongly connected if there is a path from N_1 to N_2 and one from N_2 to N_1. A strongly connected component (scc) is a maximal set of strongly connected nodes.

Equation system:	Local fixpoint: (initial call $f_1(true, true)$)	Derived calls:
$f_1(x_1, x_2) = f_1(true, f_2(x_1, x_2))$	$\phi_1(true, true) = false$	$\{ \langle f_1, \langle true, true \rangle \rangle,$
$f_2(x_1, x_2) = x_1 \wedge x_2$	$\phi_2(true, true) = true$	$\langle f_2, \langle true, true \rangle \rangle \}$

Figure 1: A local fixpoint problem involving two binary user-defined functions.

3 Properties shared by the three algorithms in the test

The three algorithms take as input an equation system and a set of initial calls, and compute a local fixpoint as well as the derived calls. Some important properties shared by the algorithms are:

(i) They are basically *iterative* algorithms.
(ii) The fixpoint approximation is improved in a *pointwise* manner.
(iii) *Depth-first evaluation* is employed to improve upon recursive calls.
(iv) *Monotonicity* of the fixpoint approximation is enforced to hold at any time.

(i) The algorithms are *iterative* in the sense that they compute a sequence of iterands; each iterand is obtained by evaluating some right hand side expression using the previous iterand as the value of the functional parameter. An iteration computes $\mathbf{E}\,[\![e_i]\!]\,\overline{\phi}\,\overline{\alpha}$, for some i, $\overline{\phi}$, and $\overline{\alpha}$. Iteration is continued until stabilization is detected or inferred. In the local fixpoint problem of Figure 1 the initial fixpoint approximation is $\overline{\phi}^1 = \langle \phi_1^1, \phi_2^1 \rangle$ with

$$\begin{array}{rcl} \phi_1^1(\alpha_1, \alpha_2) & = & false \\ \phi_2^1(\alpha_1, \alpha_2) & = & false \end{array}$$

(ii) Iteration is *pointwise* in the sense that in each step, an equation is evaluated with respect to a single value $\overline{\alpha}$ for $\overline{x}$. In the example, the first iteration evaluates the equation for f_1 with input $\langle true, true \rangle$. Pointwise evaluation is preferred here because it allows for restricting attention to the derived calls (or a small superset thereof). Also, the pointwise approach means that in the optimized algorithms, dependency is between function-value pairs, which is a more refined approach than considering dependency merely between functions.

(iii) The algorithms employ *depth-first evaluation* in the sense that an iteration may in some cases be suspended, while a need is being improved upon. The equation for f_1 has two recursive calls. To evaluate the call to f_2, by definition of **E** we must apply ϕ_2^1 to $\langle true, true \rangle$, the result of which is $false$. Instead of implementing naively the definition of **E**, all three algorithms suspend the evaluation of

proc $update(\phi_i, \overline{\alpha}, val)$
begin
  $\phi_i(\overline{\alpha}) := \phi_i(\overline{\alpha}) \vee val$
  maintain monotonicity of ϕ_i
end $\{update\}$

funct $eval(e, \overline{\alpha})$
begin
  case e **of**
    x_j — $eval := \alpha_j$
    $b_j(\overline{e})$ — **for** $i := 1$ **to** a **do** $\beta_i := eval(e_i, \overline{\alpha})$
        $eval := base_j(\overline{\beta})$
    $f_j(\overline{e})$ — **for** $i := 1$ **to** a **do** $\beta_i := eval(e_i, \overline{\alpha})$
        maintain monotonicity of ϕ_j
        $visit(f_j, \overline{\beta})$
        $eval := \phi_j(\overline{\beta})$
end $\{eval\}$

Figure 2: Procedure $update$ and functional procedure $eval$ as used by all three algorithms in the test. The fixpoint approximation $\overline{\phi}$ is a global variable.

the first equation, and evaluate the second equation. This produces the improved iterand $\overline{\phi}^2 = \langle \phi_1^2, \phi_2^2 \rangle$ with

$$\begin{array}{rcl} \phi_1^2(\alpha_1, \alpha_2) & = & false \\ \phi_2^2(\alpha_1, \alpha_2) & = & \begin{cases} true & \text{if} \quad \langle \alpha_1, \alpha_2 \rangle = \langle true, true \rangle \\ false & \text{otherwise} \end{cases} \end{array}$$

When the evaluation of the first equation is resumed, the recursive call to f_2 yields the value $true$. Further on in the evaluation of e_1 we are faced with a call to f_1 with input $\langle true, true \rangle$. This time evaluation is not suspended, because this function-value pair is already under evaluation. All three algorithms adhere to this principle of "truncating" depth-first evaluation once a function-value pair already under evaluation is reached.

Upon the two iterations just described, the Worklist and Local Stabilization algorithms infer that a local fixpoint has been reached, and terminate. The Kleene algorithm performs two more iterations, and then detects stabilization.

All algorithms are defined in terms of a procedure $eval$ that evaluates an L-expression (see Figure 2). In case the expression is of the form $f_i(e_1, \ldots, e_a)$, procedure $eval$ evaluates the sub-expressions and then call procedure $visit$, which is defined separately for each algorithm.

Procedure $update$ takes care of modifying the fixpoint approximation when the evaluation of a right hand side yields a value not exceeded by the previous value.

(iv) Procedures $update$ and $eval$ take some measures to *preserve monotonicity* of the fixpoint approximation $\overline{\phi} = \langle \phi_1, \ldots, \phi_q \rangle$, in the following sense: It must hold invariantly that each ϕ_i is monotonic when restricted to the subset of $\mathcal{B}^a$ containing those values $\overline{\alpha}$ for which there has been a call to $visit(f_i, \overline{\alpha})$. Whenever ϕ_i becomes non-monotonic upon a modification, it is replaced by the least monotonic function lying above it. In procedure $update$, if $\overline{\alpha}'$ is such that $\overline{\alpha} \sqsubset \overline{\alpha}'$ and $\langle f_i, \overline{\alpha}' \rangle$ has been subjected to evaluation, then the inequality

$$\phi_i(\overline{\alpha}) \sqsubseteq \phi_i(\overline{\alpha}')$$

is enforced by possibly modifying the value of $\phi_i(\overline{\alpha}')$. In procedure $eval$, if $\overline{\beta}'$ is such that $\overline{\beta}' \sqsubset \overline{\beta}$ and $\langle f_j, \overline{\beta}' \rangle$ has been subjected to evaluation, then the inequality

$$\phi_j(\overline{\beta}') \sqsubseteq \phi_j(\overline{\beta})$$

is enforced by possibly modifying the value of $\phi_j(\overline{\beta})$. (These measures to preserve monotonicity correspond to procedures $adjust$ and $extend$, respectively, of GAIA [10]). The process of enforcing monotonicity can be shown easily to never "skip over" the fixpoint, and its computational cost seems acceptable because the number of future iterations is reduced.

The reader is warned that in some examples the three algorithms will appear rather naive, perhaps suggesting some obvious optimizations:

Simplification of right hand sides: A right hand side expression of the form $true \vee e$ can be rewritten into $true$ without affecting the global fixpoint. However, the optimization is not generally correct when a local fixpoint is sought, because e may contain a call to a user defined function, and so the associated derived call may be missed out.

Omitting evaluation in case the top value has already been reached: If a right hand side e is already known to evaluate to $true$, re-evaluation can never lead to an update. Again, since we are looking for a local fixpoint, omitting re-evaluation is not generally correct. Suppose e contains nested calls to user defined functions, say if e is of the form $f_j(\ldots f_k(\ldots)\ldots)$. Then omitting re-evaluation of e may lead to overlooking a derived call to f_j, since the value associated with the call to f_k may change.

An *early update optimization* is possible for right hand sides that are disjunctions. Consider initial call f_1 to equation system:

$$\begin{array}{lcl} f_1 & = & true \vee f_2 \\ f_2 & = & f_1 \end{array}$$

By the time the subexpression $true$ is encountered, it can be inferred that the current evaluation of f_1 will produce the value $true$, so this could be recorded in the fixpoint approximation. When f_2 is evaluated, at which time the evaluation of f_1 is suspended, the improved value could be retrieved for f_1, instead of the initial value $false$. The algorithm GAIA includes this optimization. The optimization is certainly very natural in a Prolog context, where the equation for predicate p defined by clauses $C1 \ldots Cn$ is of the form

$$f_p(\ldots) = f_{C1}(\ldots) \vee \ldots \vee f_{Cn}(\ldots)$$

Although this optimization is correct, it is not part of any of the three algorithms in the test, because of the goal of attaining algorithm definitions that are as simple as possible.

4 The Kleene algorithm

The Kleene algorithm is defined in Figure 3. It is included in the test to provide a simple alternative to the smarter algorithms. The basic principle is: Repeat iteration until there has been a sequence of iterations that evaluate all derived calls without producing any changes.

Input:
- a system $f_i(\overline{x}) = e_i$
- a set of initial calls $Calls$

Output:
- $\overline{\phi}$ is a local fixpoint
- $Visited$ is the derived calls

Method:

repeat
  $Visited := \emptyset$
  $stabile := true$
  for all $\langle f_i, \overline{\alpha}\rangle \in Calls$ do $visit(f_i, \overline{\alpha})$
until $stabile$

proc $visit(f_i, \overline{\alpha})$
if $\langle f_i, \overline{\alpha}\rangle \notin Visited$
then begin
  $Visited := Visited \cup \{\langle f_i, \overline{\alpha}\rangle\}$
  $val := eval(e_i, \overline{\alpha})$
  if $val \not\sqsubseteq \phi_i(\overline{\alpha})$ **then**
    $update(\phi_i, \overline{\alpha}, val)$
    $stabile := false$
end $\{visit\}$

Figure 3: The Kleene algorithm.

Description of the Kleene algorithm. The pseudo-code of Figure 3 uses a variable $\overline{\phi}$ to hold the current fixpoint approximation. The variable $Visited \subseteq \{1, \ldots, q\} \times \mathcal{B}^a$ is used to detect when to truncate depth-first evaluation; it is reset to the empty set for every execution of the *repeat* loop defined at the top-most level. Execution of the loop continues until all calls collected in $Visited$ evaluate to a stabile value. By the last execution of the loop, the fixpoint has already been reached, and so $Visited$ ultimately becomes exactly the set of derived calls.

Test hypothesis for the Kleene algorithm. One might expect the Kleene algorithm to behave poorly on large problems, since it evaluates the equation system all over until stabilization is detected. This leads to the somewhat skeptical hypothesis: *The Kleene algorithm behaves poorly on equation systems with many equations*.

5 The Chaotic Worklist algorithm

The Worklist algorithm is defined in Figure 4. The basic principle is to use a neededness analysis to avoid redundant iterations.

A function-value pair is only re-evaluated if one of its needs has been updated, in which case the function-value pair is placed in the worklist where it awaits processing. The algorithm terminates when the worklist is empty. A record of who needs who is kept in the form of a representation of the (dynamic) dependency graph. The algorithm is chaotic in the sense that the order in which elements are selected from the worklist is arbitrary. (The notion of chaotic iteration is due to Cousot and Cousot's [5, 6]).

Worst-case complexity. A major advantage of the worklist algorithm is its low worst-case complexity. For a-ary boolean functions, an equation $f_i(\overline{x}) = e_i$ is iterated over at most

$$2^a * (1 + R) \tag{2}$$

times, where R is the number of recursive calls to user defined functions in the expression e_i. The factor 2^a is by taking into consideration each of the possible input values. In the factor $(1+R)$, the constant 1 accounts for the first evaluation, and R for re-evaluation. Each recursive call $f_j(\ldots)$ occurring in the right hand side e_i will evaluate, during the entire fixpoint computation, to a monotonically increasing sequence of values, of form either $false, \ldots, false$ or $false, \ldots, false, true, \ldots, true$. Thus each such call contributes with at most a single update.

Note that the bound (2) is independent of the number and shape of the other equations in a given system. A detailed analysis of a neededness-based worklist algorithm is provided in [9]. The worklist algorithm evaluated in this paper is similar to the one studied in [9] except that the algorithm defined in Figure 4 incorporates depth-first evaluation to attain better average-case performance. When the right hand sides are small, the bound (2) improves significantly over the bound for the Kleene algorithm of $q^2 * a * 2^a$ (for the system as a whole). This bound also holds for the Worklist algorithm, and may be attained, as a worst case, when the number of recursive calls per equation is of the order of q.

```
Input:                                   proc visit(f_i, ᾱ)
  • a system f_i(x̄) = e_i               if ⟨f_i, ᾱ⟩ ∉ Visited
  • a set of initial calls Calls          or ⟨f_i, ᾱ⟩ ∈ Worklist \ Active
Output:                                  then begin
  • φ̄ is a local fixpoint                  Visited := Visited ∪ {⟨f_i, ᾱ⟩}
  • Visited is a superset                   Worklist := Worklist \ {⟨f_i, ᾱ⟩}
    of the derived calls                    Active := Active ∪ {⟨f_i, ᾱ⟩}
                                            val := eval(e_i, ᾱ)
Method:                                     Active := Active \ {⟨f_i, ᾱ⟩}
                                            if val ⋢ φ_i(ᾱ) then
Worklist := Calls                              update(φ_i, ᾱ, val)
while Worklist ≠ ∅ do                          add successors of ⟨f_i, ᾱ⟩ to Worklist
  select ⟨f_i, ᾱ⟩ ∈ Worklist              end {visit}
  visit(f_i, ᾱ)
```

Figure 4: The Chaotic Worklist algorithm.

Description of the Chaotic Worklist algorithm. The variables $\overline{\phi}$ and $Visited$ are as in the definition of the Kleene algorithm, except that $Visited$ grows monotonically. The set of function-value pairs $Active$ is used to truncate depth-first evaluation. $Worklist$ contains function-value pairs whose needs have been updated, at times past the previous evaluation of the pairs. Function-value pairs are placed in $Worklist$ in a way that preserves the following invariant:

> If $\langle f_i, \overline{\alpha} \rangle$ has been visited, and is not in $Worklist$, then the evaluation of the i'th equation with input $\overline{\alpha}$ and with respect to the current fixpoint approximation, will produce the same value as can be obtained by looking-up the value $\phi_i(\overline{\alpha})$.

The instruction for placing pairs in $Worklist$ is "add successors ..." in procedure $visit$. The pair $\langle f_i, \overline{\alpha} \rangle$ is a successor to $\langle f_j, \overline{\beta} \rangle$ if the current dependency graph contains an edge $\langle f_i, \overline{\alpha} \rangle \leftarrow \langle f_j, \overline{\beta} \rangle$, that is, if $\langle f_i, \overline{\alpha} \rangle$ needs $\langle f_j, \overline{\beta} \rangle$. However, $\langle f_i, \overline{\alpha} \rangle$ is not added to $Worklist$ in the following case, which singles out a situation due to the depth-first approach: The successor $\langle f_i, \overline{\alpha} \rangle$ is already under evaluation, and the evaluation of the right hand side has not yet reached the subexpression that

Equation system:	Local fixpoint: (initial call f_1)	Iterations:
$f_1 = f_2 \wedge f_3$	$\phi_1 = true$	Kleene : 12
$f_2 = f_3 \vee true$	$\phi_2 = true$	Worklist : 8
$f_3 = f_4$	$\phi_3 = true$	Local Stab. : 7
$f_4 = f_2$	$\phi_4 = true$	

Figure 5: An equation system that pinpoints a weakness of the Worklist algorithm.

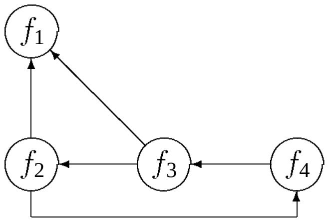
corresponds to the edge in the dependency graph. Re-evaluation is redundant in this case, since the enlarged value of the predecessor will be used at resumption of the suspended iteration.

It may be the case that a function-value pair is improved upon in order to maintain monotonicity of the fixpoint approximation (cf. point (iv) of Section 3), in which case its successors are added to $Worklist$ just as if the improvement were due to a proper iteration.

Test hypothesis for the Chaotic Worklist algorithm. The chaotic strategy for selecting elements from $Worklist$ for re-evaluation (cf. "select" in the top-most loop in Figure 4) is a possible source of inefficiency. The present test comprises three strategies: Last In - First Out, First In - First Out, and random. These strategies are coarse-grained in the sense that they are not based on analysing how the respective elements of $Worklist$ are placed in the current dependency graph. (Of course, when they are put in the worklist in the first place, this is because of their position in the dependency graph, sitting as successors of updated elements). This entails two possible sources of inefficiency:

First, the Worklist algorithm works indiscriminately on all elements that have been visited at some point of time. In addition to the derived calls, the set $Visited$ may contain so-called spurious calls (see Bruynooghe and Gallagher [7]) whose fixpoint values are not of interest, and so the time spent evaluating them is wasted.

Second, the three selection-strategies are too coarse-grained to incorporate knowledge about the strongly connected components in the dependency graph. Consider the equation system of Figure 5. As can be seen from the following diagram, the dependency graph comprises two strongly connected components: $\{f_1\}$ and $\{f_2, f_3, f_4\}$.



Clearly it is preferable to have work on component $\{f_2, f_3, f_4\}$ precede work on component $\{f_1\}$. The reader may wish to verify that by any of the selection strategies, the Worklist algorithm performs a total of eight iterations, two for each equation. In the first iteration over the equation for f_1, the value obtained for the call to f_3 is the incorrect $false$. Therefore, when ϕ_3 is later updated to $true$, the equation for f_1 must be reconsidered. However, it is redundant to iterate more than once over the equation for f_1, since the fixpoint values for the elements of component $\{f_2, f_3, f_4\}$ are independent of that equation.

The test hypothesis is: *The Worklist algorithm does a substantial amount of redundant work outside of the minimal function graph, and on function-value pairs belonging to strongly connected components that are evaluated too early.*

6 The Local Stabilization algorithm

The algorithm referred to here as the Local Stabilization algorithm is a variant of an algorithm proposed by Le Charlier, Musumbu, and Van Hentenryck [10]. One basic principle is similar to that underlying the Worklist algorithm: to rely on a dynamic neededness analysis to avoid redundant computation. Another principle is to achieve local stabilization. Because of the way the latter principle is realized in the algorithm, in a static dependency graph the strongly connected components will always be processed in topological order.

Given the equation system of Figure 5, the algorithm will find the fixpoint values for the elements of the component $\{f_2, f_3, f_4\}$ early enough to avoid redundant iteration over the equation for f_1. This is due to a strategy of re-evaluating any element *instantly upon its previous evaluation*, unless it can be inferred that re-evaluation will not lead to an update. Thus in solving the system of Figure 5, the equation for f_2 is re-evaluated immediately upon its first evaluation, and this ensures that ϕ_3 gets the correct value $true$, prior to the evaluation of the call to f_3 from f_1.

The algorithm that underlies that Local Stabilization algorithm is sometimes referred to as "GAIA", for Generic Abstract Interpretation Algorithm. GAIA is similar to an algorithm described by Muthukumar and Hermenegildo [13]. Both were originally proposed for analysis of Prolog programs, as refinements over an algorithm proposed by Bruynooghe *et al.* [3]. GAIA has been shown to be optimal for certain classes of programs, and the experimental evaluation reported in [12] showed the algorithm to perform very well.

The Local Stabilization algorithm is defined in Figure 6. Since GAIA is already generic in the sense of being essentially independent of the Prolog context, it is straightforward to translate it into the present formalism as laid down by Definition 1. We conjecture that the algorithm of Figure 6 is identical to GAIA in the sense that any equation system within the formalism entails the same sequence of iterations (in particular, the same number of iterations). This implies that they are similar with respect to the test observations. (Implementing GAIA using a worklist may not be a good idea in terms of actual computation time). This is with the exception of the optimization called "early update" mentioned in Section 3.

Input:

- a system $f_i(\overline{x}) = e_i$
- a set of initial calls $Calls$

Output:

- $\overline{\phi}$ is a local fixpoint
- $Visited$ is a superset of the derived calls

Method:

for all $\langle f_i, \overline{\alpha} \rangle \in Calls$ do $visit(f_i, \overline{\alpha})$

proc $visit(f_i, \overline{\alpha})$
while $\langle f_i, \overline{\alpha} \rangle \notin Visited$ or
 the dependency graph contains
 a path of non-active nodes
 from a node in $Worklist$ to $\langle f_i, \overline{\alpha} \rangle$
do begin
 ... < as in worklist algorithm >
end $\{visit\}$

Figure 6: The Local Stabilization algorithm.

Description of the Local Stabilization algorithm. The definition given in Figure 6 uses the same variables as the Worklist algorithm of Figure 4. In particular, the rules for inserting elements into $Worklist$ are the same.

The strategy for re-evaluation of function-value pairs is built into the condition controlling the $while$-loop of procedure $visit$. A function-value pair not previously visited is always evaluated. A pair $\langle f_i, \overline{\alpha} \rangle$ is re-evaluated if there is a path from an element of $Worklist$ to the pair, and the path consists of non-active pairs, that is, pairs that are not currently under evaluation. If there is such a path, the pair in the worklist is a (direct or indirect) need of $\langle f_i, \overline{\alpha} \rangle$. The depth-first approach means that re-evaluating $\langle f_i, \overline{\alpha} \rangle$ leads to re-evaluating a sequence of pairs, and eventually to the re-evaluation of the pair in $Worklist$.

Negating the condition of the $while$-loop yields that re-evaluation of a pair $\langle f_i, \overline{\alpha}\rangle$ already visited is omitted if all paths in the dependency graph leading to the pair are on the following form

$$\begin{array}{lll} \langle f_i, \overline{\alpha}\rangle & \leftarrow & \langle \text{not in } Worklist\rangle \leftarrow \ldots \leftarrow \langle \text{not in } Worklist\rangle \leftarrow \;\; \ldots \\ \langle f_i, \overline{\alpha}\rangle & \leftarrow & \langle \text{not in } Worklist\rangle \leftarrow \ldots \leftarrow \langle \text{not in } Worklist\rangle \leftarrow \;\; \langle \text{in } Active\rangle \leftarrow \ldots \end{array}$$

By the rules for inserting elements in the $Worklist$ it holds that if all paths are as above, re-evaluation of $\langle f_i, \overline{\alpha}\rangle$ cannot lead to an update. If all paths are of the first form, the pair has reached its fixpoint value. If one or more paths are of the second form, the fixpoint value may not have been reached; however, re-evaluation at the present time is redundant because but depth-first evaluation will be truncated at the active element.

Test hypothesis for the Local Stabilization algorithm. The re-evaluation strategy underlying the local stabilization algorithm is in a sense an *eager* strategy. Re-evaluation takes place on the fly, and whenever it may possibly lead to an update. The advantage noted above — that strongly connected components (of a static dependency graph) are evaluated in an appropriate order — is due to this. A disadvantage associated with the eager strategy is that redundant work is sometimes done *inside* a component. As an example, we will consider the equation system of Figure 7. (The quadratic performance observed here is consistent with the complexity analysis of [12]).

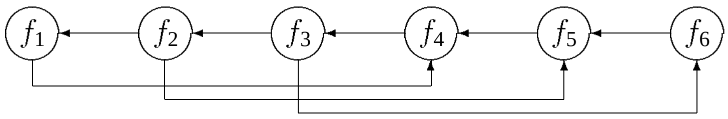
Equation system:				Local fixpoint (initial call f_1):			Iterations:		
f_i	=	f_{i+1}	$(1 \leq i < n)$	ϕ_1	=	$true$	Kleene	:	$6n$
f_n	=	$f_{n+1} \vee true$		ϕ_2	=	$true$	Worklist	:	$3n + 1$
f_{n+i}	=	$f_i \wedge f_{n+i+1}$	$(1 \leq i < n)$	ϕ_3	=	$true$	Local Stab.	:	$n^2 + 3n$
f_{2n}	=	f_n		ϕ_4	=	$true$			

Figure 7: An equation system yielding quadratic performance of the Local Stabilization algorithm.

For $n = 3$ the equations are:

$$\begin{array}{lll} f_1 & = & f_2 \\ f_2 & = & f_3 \\ f_3 & = & f_4 \vee true \\ f_4 & = & f_1 \wedge f_5 \\ f_5 & = & f_2 \wedge f_6 \\ f_6 & = & f_3 \end{array}$$

and the minimal function graph is:



The Local Stabilization algorithm will start by evaluating (in that order) $f_1, \ldots, f_6$. Unwinding the recursion, the (incorrect) value $false$ is obtained for each of the calls to f_6, f_5, and f_4. Then the iteration of f_3 finishes, producing $true$. Now there is a path in the dependency graph

$$f_3 \leftarrow f_4 \leftarrow f_5 \leftarrow f_6$$

leading to f_3 which has just been evaluated, and beginning with f_6 which is in the workset, because f_6 needs the newly updated f_3. Therefore all four nodes of the path are re-evaluated. Subsequently all nodes in the paths

$$f_2 \leftarrow f_3 \leftarrow f_4 \leftarrow f_5$$

and

$$f_1 \leftarrow f_2 \leftarrow f_3 \leftarrow f_4$$

are re-evaluated as well. The example also shows that unlike the Worklist algorithm, which re-evaluates nodes that have updated *direct* needs, the Local Stabilization algorithm re-evaluates nodes that have updated *direct or indirect* needs. The number of iterations required to process the elements a strongly connected component may therefore be quadratic in the size of the component, as in the above example. The test hypothesis is: *The Local Stabilization algorithm performs a large amount of redundant work on large strongly connected components*.

7 The random equation systems

This section first describes the equation generator, and then evaluates whether the samples of equation systems actually generated are appropriate.

7.1 The random equation generator

The syntax of the randomly generated equations is as in Definition 1, with a fixed arity of two. A system of q equations is generated as follows: The ith equation ($2 \leq i \leq q$) is of the form

$$f_i(x_1, x_2) = G_1(G_2(x_1, x_2), G_3(x_1, x_2))$$

where each symbol G_h is

- with probability $\frac{1}{2} * \frac{1}{q}$ the symbol f_i with $1 \leq i \leq q$, and
- with probability $\frac{1}{2} * \frac{1}{6}$ any one of the following six symbols: $base_{true}$, $base_{false}$, $base_{right}$, $base_{left}$, $base_{and}$ and $base_{or}$.

and where the base function symbols are interpreted as the following monotonic functions in $\mathcal{B}^2 \to \mathcal{B}$:

$$\begin{array}{lcl} base_{true}(\alpha_1, \alpha_2) & = & true \\ base_{false}(\alpha_1, \alpha_2) & = & false \\ base_{right}(\alpha_1, \alpha_2) & = & \alpha_2 \\ base_{left}(\alpha_1, \alpha_2) & = & \alpha_1 \\ base_{and}(\alpha_1, \alpha_2) & = & \alpha_1 \wedge \alpha_2 \\ base_{or}(\alpha_1, \alpha_2) & = & \alpha_1 \vee \alpha_2 \end{array}$$

Because the test problem was to find a local fixpoint with respect to an initial call to f_1, the function symbols in the right hand side of the equation for f_1 are constrained to be user function symbols: The first equation is always of the form

$$f_1(x_1, x_2) = f_i(f_j(x_1, x_2), f_h(x_1, x_2))$$

This exception from the general scheme is made to eliminate the possibility that the dependency graph merely contains the initial call, as would be the case if all three function symbols in e_1 were base function symbols.

Because of the definition of procedure $eval$ (cf. Figure 2) the expression

$$base_{true}(e_1, e_2)$$

evaluates as if it read

$$e_1 \vee e_2 \vee true$$

and similarly $base_{left}(e_1, e_2)$ evaluates as $e_1 \vee (e_2 \wedge false)$.

The random procedure sketched above is clearly very simple, as well as easy to reproduce or modify. The right hand side expressions of equations may contain nested calls to user defined functions, possibly yielding dependency graphs that change dynamically. Indeed, the depth of right hand sides is the smallest possible allowing for nested calls.

$$
\begin{array}{rcl}
f_1(x_1,x_2) & = & f_{88}(f_{81}(x_1,x_2), f_{44}(x_1,x_2)) \\
f_2(x_1,x_2) & = & f_{30}(f_1(x_1,x_2), x_2)) \\
f_3(x_1,x_2) & = & x_2 \wedge f_{13}(x_1,x_2)) \\
f_4(x_1,x_2) & = & f_{99}(f_{79}(x_1,x_2), f_{11}(x_1,x_2)) \\
f_5(x_1,x_2) & = & f_{53}(f_{30}(x_1,x_2), f_{70}(x_1,x_2)) \\
f_6(x_1,x_2) & = & f_{23}(true, x_1)) \\
f_7(x_1,x_2) & = & f_{62}(f_{87}(x_1,x_2), x_2) \\
f_8(x_1,x_2) & = & f_{19}(false, x_2) \\
f_9(x_1,x_2) & = & true \vee (f_{75}(x_1,x_2) \wedge false) \\
f_{10}(x_1,x_2) & = & true \\
 & \vdots & \\
f_{100}(x_1,x_2) & = & f_{63}(f_{78}(x_1,x_2), false))
\end{array}
$$

Figure 8: A fragment of a system of equations generated by the randomizer. Expressions that contain no calls to user defined functions have been simplified. The initial call is $\langle f_1, \langle false, false\rangle\rangle$.

7.2 The sample

The test comprises ten samples that each contain 1000 equation systems. The samples are described in Figure 9. The profile describes the minimal function graph that the equation systems give rise to, that is, it describes the input problems in terms of the dependency graph associated with the equations at the time when a fixpoint has been found.

Derived calls is the average number of derived calls in each sample. The column shows that there are, on the average, 1575 elements in the minimal function graphs of the sample containing the largest equation systems.

The derived calls are described further in two columns:

Derived calls (in % of $4 * q$**)** gives the proportion of the total number of function-value pairs that are derived calls.

Derived calls (in % of $4 * RF$**)** is perhaps a more interesting number. RF is the number of reachable functions, defined as follows: Function f_1 is a reachable function, and so is any function which is called in the right hand side of an equation for a reachable function. None of the algorithms will do any work on non-reachable functions. The column shows that when one restricts attention to the reachable functions, only about one third of the function-value pairs are not in the minimal function graph.

Size of scc's is the average size of the strongly connected components in the minimal function graph. The average which is displayed is defined as $(\Sigma_k s_k)/1000$,

Profiles of the samples					
Equations (q)	Derived calls	Derived calls (in % of $4 * q$)	Derived calls (in % of $4 * RF$)	Size of scc's	Size of scc's (weighted)
100	146	37 %	63 %	4.3	77
200	304	38 %	66 %	4.3	172
300	461	38 %	66 %	4.2	259
400	626	39 %	67 %	4.2	359
500	775	39 %	67 %	4.2	443
600	936	39 %	68 %	4.2	535
700	1092	39 %	68 %	4.2	630
800	1253	39 %	68 %	4.2	723
900	1420	39 %	68 %	4.2	821
1000	1575	39 %	68 %	4.2	912

Figure 9: Profiles of the ten samples. Each row describes a sample of 1000 equation systems comprising a fixed number of equations.

where s_k is the average size of scc's in the k'th minimal function graph (with $1 \leq k \leq 1000$). This column shows that on the average, there are about four function-value pairs in each strongly connected component.

Size of scc's (weighted) is defined as $(\Sigma_k ws_k)/1000$, where ws_k is the weighted size of scc's in the k'th minimal function graph. ws_k is defined as

$$\frac{\textit{Sum of squares}}{\textit{Derived calls}}$$

where *Sum of squares* is the sum of the squares of the sizes of the strongly components and *Derived calls* is the number of derived calls, both numbers for the k'th minimal function graph. For any minimal function graph, the size of the largest scc is at least ws_k. Thus, in the sample of systems of 1000 equations, on the average more than 900 nodes out of a total of 1575 nodes belong to a single strongly connected component.

A review of the sample profiles indicates that from the point of view of the minimal number of elements the algorithms have to work on, the samples are providing the algorithms with some fairly large problems to work on. Moreover, the input problems are hard in the sense that they contain some very large strongly connected components.

At the same time there are two apparent weaknesses of the samples. First, only about one third of the function-value pairs do not sit in the minimal function graph, when one restricts attention to the reachable functions. This is unfortunate because it is desirable that the relative number of derived calls is small enough to give a

Performance of the three algorithms			
q	Kleene	Worklist	Local Stab.
100	3.00	1.21	1.85
200	3.40	1.23	2.49
300	3.59	1.23	2.95
400	3.75	1.23	3.54
500	3.85	1.23	4.02
600	3.94	1.23	4.51
700	4.02	1.23	5.09
800	4.13	1.24	5.55
900	4.17	1.24	6.19
1000	4.24	1.24	6.66

Figure 10: Performance of the three algorithms in terms of the average number of iterations per element of the minimal function graph.

good change of trapping the algorithms working on the remaining ones. Second, there is a lack of variation in the structure of the dependency graph. The large weighted scc-size and the low unweighted size imply that each equation system typically gives rise to a dependency graph with one very large component plus a large number of components all of size 1.

8 Test results

The performance-measurements are displayed in Figure 10. Performance is measured in terms of the total number of iterations, divided by the size of the minimal function graph. The number given for each value of q is the (unweighted) average over the sample's 1000 systems containing q equations. A performance figure of 1 is optimal in the sense that any iterative algorithm must perform at least one iteration per derived call.

The Kleene algorithm. The test objective was to evaluate a hypothesis (see Section 4) that the algorithm would behave poorly on systems containing many equations. In the test, the Kleene algorithm did an average of 3 iterations per derived call on systems with 100 equations, going up to 4.24 iterations on systems with 1000 equations. Doubling the number of equations, say from 400 to 800, affects performance only marginally (from 3.75 to 4.13). Thus the test contradicts the hypothesis. The rate at which the performance parameter increases, as the number of equations goes up, is surprisingly low.

The Chaotic Worklist algorithm. The test objective (see Section 5) was to evaluate the amount of redundant work on spurious elements, and on strongly connected components that are evaluated too early. Separate tests were conducted for the three selection strategies Last In - First Out, First In - First Out, and random. Each test independently produced the figures shown in the table.

The number of iterations per derived call ranges from 1.21 times (100 equations) to 1.24 times (1000 equations), on the average. Since all iterations are counted, including those that evaluate spurious elements, the Worklist algorithm (as well as the other algorithms) is punished for working on these. The amount of redundant work observed in the test can be estimated as low, lying quite close to the optimal figure of 1.

The analysis of the profiles of the samples (see Section 7.2) should be taken into account at this point. The analysis revealed two weaknesses of the sample. First, the proportion of possible spurious elements was found to be only about one third. Working redundantly on these elements can at most increase the performance parameter by a factor of 1.5 (assuming no more work is done on spurious elements that on derived calls), and one could wish the test was more sensitive to redundant work on spurious elements. Second, the sample was found to lack variation in the structure of the dependency graphs. The inefficiency possibly entailed by the Worklist algorithm's unability to distinguish between different strongly connected components is only evaluated on graphs having the particular profile found in the test. (The example equation system of Figure 5 is consistent with this profile). Less good performance might be found on dependency graphs containing, say hundreds of middle-sized components of 5-10 elements. Thus, while the Chaotic Worklist algorithm performs well in the test, the performance is less impressive in view of the fact that the algorithm's presumed weaknesses were not targeted by the sample as much as one could wish.

The Local Stabilization algorithm. The test objective was to evaluate the algorithm's behavior on dependency graphs containing very large strongly connected components. The performance parameter is 1.85 for systems of 100 equations, going up to the fairly large value of 6.66 for systems with 1000 equations. The parameter increases proportionally to the number of equations. For systems of 500 equations and more, it exceeds the value found for the Kleene algorithm.

Suppose any strongly connected component's contribution to the total number of iterations is quadratic in the size of the component (as in the worst case). Then the number of iterations will be proportional to the weighted average size of the components, as measured in the rightmost column of Figure 9. This number also increases proportionally to the number of equations. This may indicate that the source of the increase found in the performance parameter for the Local Stabilization algorithm is due to the presence of large strongly connected components.

For all three algorithms, the sample size of 1000 systems for each value of q seems sufficient to attain reasonably stabile values for the performance parameter, that is, values that are independent of the initial state of the random number generator that underlies the random equation generator. A simple statistical analysis where the performance parameter is approximated by a normally distributed stocastic variable supports this. For instance, in the case $q = 1000$ a normal distribution whose variance is equal to the empirical variance of the performance parameter for the worklist algorithm yields a 95 % confidence-interval of width less than 0.01. For the Kleene and Local Stabilization algorithms the widths are less than 0.1 and 0.3, respectively. This indicates that the performance measurements would remain the same if the sample size were increased.

9 Conclusion

A new fixpoint algorithm — referred to as the The Chaotic Worklist algorithm — has been defined in a common setting along with two other algorithms: the simple Kleene-algorithm, and the more sophisticated Local Stabilization algorithm, being based on GAIA [10]. It was observed that the Local Stabilization algorithm processes the elements of the various strongly connected components in topological order (if the dependency graph is static), while the Worklist algorithm proceeds in a more chaotic and less orderly fashion. Also it was observed that the Worklist algorithm re-evaluates nodes that have updated *direct* needs, while the Local Stabilization algorithm re-evaluates nodes that have updated *direct or indirect* needs. Both observations were facilitated by the definition of the algorithms in a common setting.

The Kleene algorithm proved to perform much better than expected on large equation systems. The Chaotic Worklist algorithm was found to perform within 25 % above the optimal. The Local Stabilization algorithm was found to do much redundant work, very likely because of the presence of large strongly connected components in the samples.

The test results may give some indication of the algorithms performance on real world input with profiles that match the test profiles. No overall assessment of the relative performance of the three algorithms studied in the test can be made on the basis of the test. For the Chaotic Worklist algorithm, this caveat is emphasized strongly, because the profiles of the sample indicate that the test did not challenge the algorithm as much as one could wish.

However, if one takes into account that the Kleene algorithm is much easier to implement than the two more sophisticated algorithms, it is perhaps the "winner" of the test. — If bubble sort did as well compared to quicksort as the Kleene algorithm does compared to the two "quicksorts" of the test, bubble sort would probably be considered a very attractive alternative !

Acknowledgement. Thanks to Mads Rosendahl for inspiring discussions.

References

[1] F. Bancilhon and R. Ramakrishnan. *An Amateur's Introduction to Recursive Query Processing Strategies.* MCC Technical Report No. DB-091-86, March 1986.

[2] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. *Proc. FMP '93*, Lecture Notes in Computer Science 735, Springer-Verlag, 1993.

[3] M. Bruynooghe, G. Janssens, A. Callebaut, and B. Demoen. Abstract interpretation: towards the global optimisation of Prolog programs. *Proc. 4th IEEE Symposium on Logic Programming*, San Francisco, 1987, pp. 192–204.

[4] Li-Ling Chen. *Efficient computation of fixpoints that arise in abstract interpretation*. Ph. D. Thesis, University of Illinois at Urbana-Champaign, USA, 1994.

[5] P. Cousot and R. Cousot. Automatic synthesis of optimal invariant assertions: Mathematical foundations. *Proc. ACM Symposium on Artificial Intelligence and programming languages*, SIGPLAN Notices, **12** (8),1977, pp. 1–12.

[6] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. *IFIP Conf. on Formal description of programming concepts*, North-Holland Publ. Co., Amsterdam 1977, pp. 237-277.

[7] J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, **9**, 1991, 305–333.

[8] S. Horwitz, A. Demers, and T. Teitelbaum. An efficient general iterative algorithm for dataflow analysis. Acta Informatica **24**, 679–694 (1987).

[9] N. Jørgensen. Finding fixpoints in finite function spaces using neededness analysis and chaotic iteration. *Proc. First International Static Analysis Symposium*. Lecture Notes in Computer Science 864, Springer-Verlag, 1994. An enlarged version appears as technical report no. 48 in the series *Datalogiske Skrifter*, Computer Science Dept., Roskilde University Center, 1994.

[10] B. Le Charlier, K. Musumbu, P. Van Hentenryck. A generic abstract interpretation algorithm and its complexity analysis. *Proc. 8th International Conference on Logic Programming*, 1991, pp. 64–78.

[11] N. D. Jones and A. Mycroft. Data flow analysis of applicative programs using minimal function graphs. *Proc. 13th ACM Symposium on Principles of Programming Languages*, Florida, 1986, pp. 296–306.

[12] B. Le Charlier and P. Van Hentenryck. *Experimental evaluation of a generic abstract interpretation algorithm for Prolog*. ACM Transactions on Programming Languages and Systems, Vol. 16, No. 1, 1994, pp 35–101.

[13] K. Muthukumar and M. Hermenegildo. Compile-time derivation of variable dependency using abstract interpretation. *J. Logic Programming* 1992, **13**, pp. 315–347.

[14] S. Peyton-Jones and C. Clack. Finding fixpoints in abstract interpretation, in: S. Abramski and C. Hankin (eds): *Abstract interpretation of declarative languages*. Ellis Horwood, 1987, pp. 246–265.

[15] E. Sch´on. On the computation of fixpoints in static program analysis with an application to analysis of AKL. Master Thesis, 1995. *SICS Research Report* R95-06.

[16] R. Sedgewick. The analysis of quicksort programs. *Acta Informatica* 7, 1977, pp. 327–355.

[17] J. Young and P. Hudak. Finding fixpoints on Function Spaces. Research Report RR-505, Dept. Computer Science, Yale University, Dec. 1986.