Iteration schemes for fixed point computation*

Thomas Jensen   Florimond Ployette   Olivier Ridoux

IRISA
Campus de Beaulieu
F- 35042 Rennes
France

Abstract

We report on work whose overall aim is to obtain a template fixed point algorithm that can be used to compare existing algorithms such as work-set algorithms, top-down solvers, local solvers *etc.* for calculating solutions of monotone systems of equations over lattices. The approach taken here is to focus on the dependency graph between the variables of the system and exploit this graph to schedule the iteration of the system. The resulting iteration scheme reduces the number of expressions to be re-evaluated both in theory and in practice (up to 50 % in certain cases).

1 Introduction

Data flow analysis usually involves a phase of fixed point computation whose algorithmics is largely independent of the particular property analysed for [4, 9]. This has lead to the search for generic fixed point algorithms that can serve as "back ends" in program analyzers. A common feature of these algorithms is a dependency relation that describes how the value of one variable depends on the value of another variable. However, the algorithms are often expressed in a manner that makes it difficult to compare *how* the dependency relation is exploited. The aim of the work reported here is

1. to make clear how the dependency relation can be exploited to implement and optimise a fixed point computation,
2. to provide a framework (in the form of a template algorithm plus accompanying data structures) that enables a comparative study of iteration-based fixed point algorithms.

The paper is organised as follows. After introducing the basic notation we define a theoretical iteration scheme (Section 3) based on the notion of *minimal elements* of a dependency relation. The theoretical algorithm defines an ideal iteration scheme but is defined in terms of an exact dependency relation which makes it unfeasible in practice. Section 4 shows how to calculate a safe approximation to the exact dependency relation based on a syntactic notion of dependency. This leads to a basic iteration scheme for which we then present benchmarks comparing it with other iteration schemes in terms of space, time and number of iterations. Section 6 draws some conclusions and outlines further issues to be investigated.

*This work was partially supported by the IST FET/Open project "Secsafe".

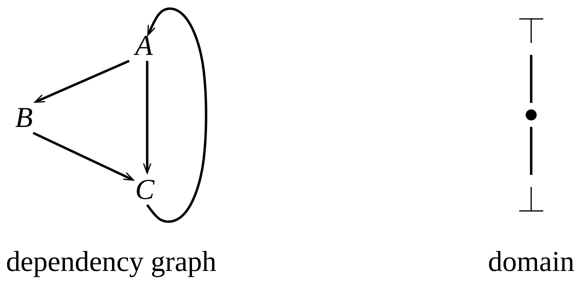

Figure 1: A toy dependency graph and domain

2 Notation

In this section we briefly define the notation to be used in the paper. For a much more comprehensive introduction to fixed points, see the recent textbook by Arnold and Niwinski [1]. In the following, i, j range over the integers $1, \ldots, n$. Let $E_1, \ldots, E_n$ be finite lattices and let e_i be an expression satisfying that all free variables in e_i are included in $\{f_1, \ldots, f_n\}$, that e_i defines a monotone function in all variables, and that evaluating e_i in an environment ϕ satisfying $\phi(f_j) \in E_j$ yields a value in E_i. We write $e_i\ \phi$ for the value of e_i in environment ϕ. A solution to a system of equations of the form

$$\{f_1 = e_1, \ldots, f_n = e_n\}$$

can be characterised as a fixed point of the function

$$e : (E_1 \times \ldots \times E_n) \to (E_1 \times \ldots \times E_n)$$

induced by the e_i and defined by $e\phi = (e_1\ \phi, \ldots,\ e_n\ \phi)$ The least solution to the system of equations can be found as the limit of the *ascending Kleene chain* (AKC)

$$\{e^{\,i}(\perp_{E_1}, \ldots, \perp_{E_n})\}_{i=0}^{\infty} \tag{1}$$

We do not put any restrictions on the type of the expressions e_i. In particular, these expressions can denote functions.

Example 1 (A toy example) *We will use as a running example the following equation system*

$$\begin{array}{l} B = incr(A) \\ C = max(B, A) \\ A = C \end{array}$$

where the variable domain is as in Figure 1 and incr is the mapping $\{\perp \mapsto \bullet, \bullet \mapsto \top, \top \mapsto \top\}$*. The syntactic dependency graph (to be defined in Section 4 of this equation system is as in Figure 1. Figure 2 shows the AKC for this example.*

A	B	C
$\bot$	$\bot$	$\bot$
$\bot$	$\bullet$	$\bot$
$\bot$	$\bullet$	$\bullet$
$\bullet$	$\bullet$	$\bullet$
$\bullet$	$\top$	$\bullet$
$\bullet$	$\top$	$\top$
$\top$	$\top$	$\top$
$\top$	$\top$	$\top$

Figure 2: The AKC for Example 1: 21 evaluations

3 Scheduling for an ideal fixed point algorithm

A somewhat less naïve method for calculating the limit of an AKC consists in iterating all equations in order (the *round-robin* algorithm). However, this is still an inefficient approach and is in general only feasible for the simplest dataflow problems [7]. The inefficiency stems from the possibility that an equation may be iterated even though the value of its defining expression will not change. This may happen either because there have been no changes in the variables occurring in the expression or because these changes do not have any impact on its value. Thus, only expressions that are guaranteed to change value should be re-evaluated during an iteration. This set of equations depends on the current approximation ϕ and is given by

$$W_\phi = \{f_i \mid \phi_i \neq e_i\ \phi\}$$

Not all equations in W_ϕ should necessarily be scheduled for iteration. If the value of f_i depends on the value of f_j then it is usually wasted effort to re-evaluate f_i before having found the new value of f_j. The only exception to this is when f_i and f_j are mutually dependent. The *dynamic dependency relation* is defined relative to the current values of variables ϕ by

$$f_j \stackrel{\phi}{\to} f_i \equiv e_i\ \phi \sqsubset e_i\ \phi[f_j \mapsto e_j\ \phi].$$

The notation "$f_j \stackrel{\phi}{\to} f_i$" should be read as "$f_i$ depends on f_j under the valuation ϕ".

3.1 Minimality

The equations in W_ϕ that should be iterated can now be characterised as the *minimal* elements of W_ϕ with respect to the dynamic dependency relation. We define for a set $S \subseteq \{1, \ldots, n\}$ and relation $\to$ its set of "minimal" elements as follows:

$$min(S, \to) \equiv \{j \in S : \mathit{if}\ \exists i \in S. i \to^+ j\ \mathit{then}\ j \to^+ i\}.$$

where $\to^+$ is the transitive closure of $\to$.

The notion of minimality corresponds to the intuitive notion of minimality when there are no cycles in the graph induced by $\to$. However, in case of a cycle in S, all elements of the cycle will belong to the *min*-set. In that case, Algorithm 1 can be further improved by only iterating *one* (arbitrarily chosen) element from each cycle. We leave it as an open problem to

A	B	C	I	W_ϕ
$\bot$	$\bot$	$\bot$	$\{B\}$	$\{B\}$
	$\bullet$		$\{C\}$	$\{C\}$
		$\bullet$	$\{A\}$	$\{A\}$
$\bullet$			$\{B\}$	$\{B\}$
	$\top$		$\{C\}$	$\{C\}$
		$\top$	$\{A\}$	$\{A\}$
$\top$			$\emptyset$	$\emptyset$

Figure 3: Trace of the ideal algorithm (Example 1): 6 evaluations

find a better definition of minimality that takes cycles into account—perhaps as an iterative definition starting from elements without predecessors.

3.2 Theoretical algorithm

The above considerations lead to the following iterative algorithm for calculating the limit of the sequence (1).

Algorithm 1

forall $i \in \{1, \ldots, n\} \phi_i := \bot$;
repeat
  $I := min(W_\phi, \xrightarrow{\phi})$;
  forall $i \in I$
    $\phi_i := e_i\ \phi$;
until $I = \emptyset$

Theorem 3.1 *Algorithm 1 terminates with ϕ a fixed point of e*

PROOF: If $I = \emptyset$ implies that $W_\phi = \emptyset$ and hence, by definition of W_ϕ, we have $\phi = e(\phi)$. $\square$

This algorithm is mainly of theoretical interest since it is in general not possible to calculate W_ϕ and $\xrightarrow{\phi}$ exactly without evaluating the defining expression of each equation and thus no work is saved. Hence approximations of these entities are needed to obtain practical algorithms.

Figure 3 shows the trace of the ideal algorithm applied to the toy example from Figure 1. We assume that every time a set of variables is selected for evaluation (I in the sequel) the corresponding expressions are evaluated in parallel.

A	B	C	WS
$\perp$	$\perp$	$\perp$	$\{A, B, C\}$
$\perp$	$\bullet$	$\perp$	$\{C\}$
		$\bullet$	$\{A\}$
$\bullet$			$\{B, C\}$
	$\top$	$\bullet$	$\{C\}$
		$\top$	$\{A\}$
$\top$			$\{B, C\}$
	$\top$	$\top$	$\emptyset$

Figure 4: Trace of the workset algorithm (Example 1): 11 evaluations

4 Scheduling for work-set algorithms

A common approach to calculating the limit of an AKC is to employ a *work-set* algorithm [8] that keeps a work-set W of equations to be iterated next. The algorithm continues to select a set of equations from the work-set for iteration according to some strategy and adds elements that are liable to have changed value to the work-set after each such iteration. The process ends when the work-set is empty. The following is a template work-set algorithm, parameterised on the strategies for selecting the next equation to iterate and for modifying the work-set.

Algorithm 2 (The template work-set algorithm)

forall $i \in \{1, \ldots, n\}$ $\phi_i, \tau_i := \perp$;
$W := init_W$;
repeat
  $I :=$ strategy(ϕ, W);
  forall $i \in I$
    $\tau_i := e_i\ \phi$;
  $W :=$ modify(W, ϕ, τ);
  $\phi := \tau$;
until $I = \emptyset$

The naïve method mentioned above is obtained by always keeping the whole set of equations in the work-set W and using a strategy that after evaluation of equation i selects equation $i + 1 \bmod n$. The theoretical iteration algorithm is obtained by setting

$$\text{strategy}(\phi, W) = min(W_\phi, \xrightarrow{\phi}),$$

replacing τ with ϕ and removing the last two assignments to W and ϕ since these no longer play any rôle in the algorithm.

A standard approximation, dating back at least to Kildall [8], of the set W_ϕ of equations that will change value amounts to replacing it with the set of equations in which a variable has changed value [8]. More precisely, the system of equations defines a syntactic *dependency*

A	B	C	I	WS
$\perp$	$\perp$	$\perp$	$\{A, B, C\}$	$\{A, B, C\}$
$\perp$	$\bullet$	$\perp$	$\{C\}$	$\{C\}$
		$\bullet$	$\{A\}$	$\{A\}$
$\bullet$			$\{B\}$	$\{B, C\}$
	$\top$		$\{C\}$	$\{C\}$
		$\top$	$\{A\}$	$\{A\}$
$\top$			$\{B\}$	$\{B, C\}$
	$\top$		$\{C\}$	$\{C\}$
		$\top$	$\emptyset$	$\emptyset$

Figure 5: Trace of the basic algorithm (Example 1): 10 evaluations

relation, denoted by $\rightarrow$, between the variables: variable f_i depends on variable f_j if f_j occurs in the expression e_i:

$$f_j \rightarrow f_i \equiv f_j \text{ occurs textually in } e_i.$$

From the $\rightarrow$ relation we define the "*image-under-*$\rightarrow$" operation !

$$I! \equiv \{j \mid \exists i \in I : f_i \rightarrow f_j\}$$

that yields the set of indices of the equations containing an x_i with $i \in I$. This is used to define "modify", as shown in Algorithm 3 below. Figure 4 shows the trace of applying a work-set algorithm to the toy example from Figure 1.

The set $I!$ over-estimates the set of variables that change value as a result of changes in the f_i, and it is this set that is added to the work-set after each iteration. As noticed in Section 3, it is possible that there are dependencies between the elements in a set $I!$ of equations to be iterated and only the minimal elements (with respect to $\rightarrow$) are scheduled for evaluation. Instantiating the work-set algorithm above, we arrive at the *basic iteration scheme*:

Algorithm 3 (Basic iteration scheme)

forall $i \in \{1, \ldots, n\} \phi_i, \tau_i := \perp$;
$W := \{1, \ldots, n\}$
repeat
  $I := min(W, \rightarrow)$;   % strategy
  forall $i \in I$
    $\tau_i := e_i\ \phi$;
  $W := (W \setminus I) \cup \{i \in I : \phi_i \neq \tau_i\}!$ % modify
  $\phi := \tau$;
until $I = \emptyset$

Proposition 4.1 *If $W_\phi \subseteq W$ holds on entry to the loop then it also holds on exit.*

Theorem 4.2 *Algorithm 3 terminates with a fixed point of e*

PROOF: Since $W_\phi \subseteq W$ trivially holds initially, it also holds when the algorithm terminates. Then $I = \emptyset$ implies $W = \emptyset$ hence $W_\phi = \emptyset$ which implies that ϕ is a fixed point. $\square$

In Figure 5 we show the trace of the basic iteration scheme applied to the example from Figure 1.

5 Experiments

The Basic Iteration Scheme has been used to implement a series of program analyses ranging from pointer analysis of C programs [6] over polyhedral analysis of synchronous SIGNAL programs [2] to class analysis of Java programs. We here report some experimental figures to show that the number of equations evaluated in the different iteration schemes do indeed decrease as we exploit the dependency relation better. For each program analysed we give the number of equations evaluated with the three different iteration schemes presented in the paper.

Analysis	No. of eqns.	Naive	Workset	Basic
C pointer analysis	42 eqs	1050	120	60
Java class analysis	49 eqs	147	75	44
Java class analysis	188 eqs	1692	272	158

As can be seen from the table, the basic algorithm obtained a 40–50 % reduction in the number of expressions that were evaluated during iteration, compared to a standard work-set algorithm. The impact of avoiding superfluous evaluations on the overall running time varies with the type of analysis. We observed significant savings only for the C pointer analysis. This is due to the fact that the expression in the C pointer analysis are all rather complex matrix manipulations whereas the expressions in the Java class analysis are simple set manipulations. Another point is that the computation of the transitive dependency relation is in itself costly. No attempt has been made to optimise this part of the computation, so for the moment the Basic Iteration Scheme only improves the overall running time of the C pointer analysis.

6 Conclusions

The Basic Iteration Scheme (BIS) constitutes a simple algorithm for implementing a fixed point solver. It relies on the dependency relation between variables to improve the standard work-set algorithm by only scheduling variables that do not depend on other variables ready for evaluation. This theoretical improvement in the number of expressions to evaluate manifests itself in practical experiemnts. There are further related issues that we have not addressed here:

- The BIS exploits the dependency relation $\rightarrow$ to avoid certain useless re-evaluations of expressions and has been used to implement several program analyzers. The practical experiments reported in Section 5 show that the overhead incurred by calculating the sets $min(W, \rightarrow)$ in certain cases exceeds the savings obtained. Thus, it is necessary to experiment with cheaper approximations to these sets.

- The top-down solvers [3, 5] will *suspend* evaluation of an expression if some of its variables need re-evaluation, and instead schedule these variables for evaluation. The top-down solvers use an evaluation stack combined with cycle detection to implement this. We obtain an effect similar to this by calculating the $min(W, \rightarrow)$-sets, by which we suspend evaluation of a variable if it depends on other variables that migt evolve. A precise comparison between the two algorithms is challenging future work.

Finally, we remark that in certain cases, we are only interested in the value of some of the variables in the system of equations. An algorithm for calculating such a *local fixed point* was given in [10]. This algorithm can be expressed quite naturally as an extension to the BIS. We will show this extension in a longer version of the paper.

References

[1] A. Arnold and D. Niwinski. *Rudiments of μ-calculus*, volume 146 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 2001.

[2] F. Besson, T. Jensen, and J.-P. Talpin. Polyhedral analysis for synchronous languages. In G. Filé, editor, *Proc. of 7th Int. Symp. on Static Analysis*. Springer LNCS vol. 1694, 1999.

[3] B. Le Charlier and P. van Hentenryck. Experimental evaluation of a generic abstract interpretation algorithm for PROLOG. ACM Trans. on Programmling Languages and Systems, 16(1):35–101, 1994.

[4] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proc. of 2. Int. Symposium on Programming*, pages 106–130, Paris, France, 1976. Dunod.

[5] C. Fecht and H. Seidl. A faster solver for general systems of equations. *Science of Computer Programming*, 35(2-3):137–162, 1999.

[6] P. Fradet, R. Gaugne, and D. Le Métayer. Static detection of pointer errors: an axiomatisation and a checking algorithm. In *Proc. European Symposium on Programming, ESOP'96*, volume 1058 of *LNCS*, pages 125–140, Linköping, Sweden, April 1996. Springer-Verlag.

[7] M. S. Hecht. *Flow Analysis of Computer Programs*. Programming Languages series. North-Holland, New York, 1977.

[8] G. Kildall. A unified approach to global program optimization. In *Proc. of ACM Symp. on Principles of Programming Languages*, 1973.

[9] F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer Verlag, 1999.

[10] B. Vergauwen, J. Wauman, and J. Levi. Efficient fixpoint computation. In B. Le Charlier, editor, *Proc. of 1st Int. Static Analysis Symp.*, pages 314–328. Springer LNCS vol. 864, 1994.