Optimized Eager Evaluation of Fixed Points for the Analysis of Logic Programs

Muhammad Abozaed
Intel, Haifa
Israel

Samir Genaim and Michael Codish
Department of Computer Science
Ben-Gurion University, Israel

Abstract

We introduce an interpreter which performs optimized fixed point evaluation based on an eager evaluation strategy. The interpreter is written in Prolog and provides a core engine for the analysis of logic programs. Eager evaluation is the preferred strategy chosen in several recent analysis implementations and we demonstrate that our technique can improve the performance of these analyzers considerably. The result is a simple and efficient, yet lightweight approach to implementing program analyses.

1 Introduction

Static analysis of (logic) programs using abstract interpretation [8] is obtained through the evaluation of an abstract semantics which is guaranteed to provide, at compile-time, finite approximations of program behaviors. A specific abstract semantics for a given type of analysis can be derived more or less systematically from a corresponding concrete semantic definition and a suitable notion of data description. Typically, the abstract semantics is obtained by evaluating the least fixed point of a set of recursive equations over the abstract semantic domain (defined in terms of the data descriptions).

We take the bottom-up approach to analysis for logic programs (see e.g. [2, 5]) adopting for the concrete semantic basis the s-semantics [3, 9, 10] which is a non-ground version of the classic T_P operator. In this approach the semantics of a program P is a set of possibly non-ground atoms which characterizes both declarative as well as operational properties of a program. In particular: (a) the ground instances of the s-semantics gives precisely the least Herbrand model of P; and (b) the computed answer substitutions for any initial query q with P are determined by solving q using the (potentially infinite) set of atoms in the s-semantics of P.

Letting $\mathcal{H}_v$ denote the set of possibly non-ground atomic formulas modulo variable renaming, the s-semantics of P is defined as the least fixed point of an operator $T_P^v : \wp(\mathcal{H}_v) \to \wp(\mathcal{H}_v)$, similar to the standard least Herbrand model semantics. More precisely,

$$T_P^v(I) = \left\{ h\theta \;\middle|\; \begin{array}{l} C \equiv h \leftarrow b_1, \ldots, b_n \in P, \\ \langle a_1, \ldots, a_n \rangle \ll_C I, \\ \theta = mgu(\langle b_1, \ldots, b_n \rangle, \langle a_1, \ldots, a_n \rangle) \end{array} \right\}$$

where $\langle a_1, \ldots, a_n \rangle \ll_C I$ expresses that $a_1, \ldots, a_n$ are variants of elements of I renamed apart from C and from each other.

In the bottom-up approach: (1) *goal-independent* analyses are obtained by abstracting the s-semantics to obtain an approximation of the relations defined by the program and of the answers to queries; and (2) *goal-dependent* analyses are obtained by abstracting the semantics in combination with the magic-sets transformation (as described in [5]) to obtain approximations to the sets of calls and answers encountered in the execution of a given (set of) initial goal(s)[1]. In practice, we do not actually apply the magic sets transformation. Rather, from the original program we extract and solve directly the recursive equations corresponding to the transformed magic program as described in [6].

Our approach to implementation is based on the specification of meta-circular interpreters which given a program solve the recursive equations to compute its semantics. For a survey on the meta-circular approach to abstract interpretation see [7] where interpreters implementing: naive, semi-naive and eager evaluation strategies are described.

Section 2 presents the eager evaluation strategy and Section 3 describes our optimisations. The experimental evaluation described in Section 4 indicates considerable speedups in analysis times and number of operations performed.

2 Eager Evaluation

The fundamental operation in the bottom-up approach for the analysis of logic programs is the iterative application of a set of facts to the bodies of program clauses to generate new facts. The essence of this approach originates in the evaluation of the least fixed point of the classic T_P operator. Semi-naive evaluation [13] optimizes this approach so that at each iteration at least one new fact (derived in the previous iteration) is used when considering a clause body.

Eager evaluation also aims to insure that at least one "new" atom is used when a clause body is solved. The difference between this strategy and the semi-naive strategy is in the book-keeping of "what is new". Eager evaluation uses a newly derived fact F immediately, postponing other tasks while exploring which new facts can be derived now that F has been established. In the process, new facts may be derived, and their investigation will then get higher priority. That is, tasks are handled in a first-in last-out manner. Whenever a new instance of a clause head is derived it is used immediately to re-solve all clauses which contain a call to the corresponding predicate in their body. Consequently eager evaluation is simple to describe as a recursive depth first strategy.

[1] The idea behind magic sets is similar to the motivation for minimal function graphs [12].

```
go(File) :-
    load_file(File),
    eager(true),
    fail.
go(_).
prove([B|Bs]) :-
    fact(B),
    prove(Bs).
prove([]).
```

```
eager(true) :-
    my_clause(H,[]),
    eager(H).
eager(Atom) :-
    cond_assert(Atom),
    my_clause(H,Body),
    select(Atom,Body,Rest),
    prove(Rest),
    eager(H).
```

Figure 1: A Prolog Interpreter for Eager Evaluation

Figure 1 depicts an interpreter distilled from that introduced by Wunderwald in [14]. This interpreter corresponds to a concrete semantic operator and as such, its evaluation does not terminate when the semantics of the program is not finite.. For applications of program analysis we apply abstract interpreters which are guaranteed to terminate. Evaluation is triggered by a call to `go(File)`. The predicate `load_file` reads in the program and represents each clause of the form $h \leftarrow b_1, \ldots, b_n$ by a fact of the form `my_clause(`h`,[`b_1`,`$\ldots$`,`b_n`])`. The predicate `cond_assert` asserts a derived fact to the set of facts derived so far, provided it is new. Eager evaluation is triggered by a call to `eager(true)` which operates under a failure driven loop. The call to `select(Atom,Body,Rest)` selects an *Atom* from the *Body* such that *Rest* contains the other body calls.

Eager evaluation can be used also for query directed bottom-up evaluation as implemented by the interpreter depicted in Figure 2 where the lines are numbered to facilitate the commentary. Here we annotate a fact as a *call* (`call(F)`) or an *answer* (`ans(F)`). Evaluation for the initial goal G is triggered by a call of the form `eager(call(G))`. There are two cases to consider when a new fact is found:

- If the new fact is an answer (line `14`), then for each clause `Head :- Body` (line `15`) that has been called (line `16`) and contains a matching call for the new answer (line `17`), we solve the `Body` (line `18`) and conditionally assert the corresponding instance of the `Head` (line `19`). If it is new then there is a recursive call (line `20`) (otherwise we fail at line `19`) and proceed.
- If the new fact is a call (line `21`), then for each clause with a matching head (line `22`), we solve the body (line `23`) and conditionally assert the corresponding instance of the head (line `24`). If it is new then there is a recursive call (line `25`) (otherwise we fail at line `24`) and proceed.

In both cases, solving a clause body involves asserting and processing new calls (lines `7-9`) and matching body atoms with with facts derived so far (lines `10-12`).

We can improve the code in Figure 2 for the case when a new answer is encountered (lines `14-20`). A new answer that matches a call b_i in the clause $h \leftarrow b_1, \ldots, b_i, \ldots, b_n$ may give rise to new call patterns for $b_{i+1}, \ldots, b_n$ but will

```
(01)  go(File,Goal) :-
(02)     load_file(File),
(03)     cond_assert(call(Goal)),
(04)     eager(call(Goal)),
(05)     fail.
(06)  go(_,_).

(07)  prove_ind([B|_]) :-
(08)     cond_assert(call(B)),
(09)     eager(call(B)).
(10)  prove_ind([B|Bs]) :-
(11)     fact(ans(B)),
(12)     prove_ind(Bs).
(13)  prove_ind([]).

(14)  eager(ans(Atom)) :-
(15)     my_clause(Head,Body),
(16)     fact(call(Head)),
(17)     select(Atom,Body,Rest),
(18)     prove_ind(Rest),
(19)     cond_assert(ans(Head)),
(20)     eager(ans(Head)).

(21)  eager(call(Atom)) :-
(22)     my_clause(Atom,Body),
(23)     prove_ind(Body),
(24)     cond_assert(ans(Atom)),
(25)     eager(ans(Atom)).
```

Figure 2: Induced Eager Evaluation

not influence calls for $b_1, \ldots, b_i$. This means that the case covered by lines 7-9 in the code is redundant for atoms to the left of (and including) b_i. A more efficient interpreter is obtained by replacing lines 17 and 18 by

```
(17')   append(Left,[Atom|Right],Body),
(18')   prove(Left), prove_ind(Right),
```

respectively and defining `prove` by

```
prove([B|Bs]) :- fact(ans(B)), prove(Bs).
prove([]).
```

We introduce, through an example, the notion of an *eager evaluation tree* as a formalisation of goal directed eager evaluation. This is an abstraction of the computation tree for the interpreter with a given program and initial goal.

Example 1 *Consider the following program P which defines the* subset/2 *relation and the initial query $G = subset([a],[a])$.*

```
(c1) subset([],Ys).                       (c3) member(X,[X|_]).
(c2) subset([X|Xs],Ys) :-                 (c4) member(X,[Y|Xs]) :-
       member(X,Ys), subset(Xs,Ys).              member(X,Xs).
```

The tree in Figure 3 illustrates the execution of the interpreter. A node labelled by a fact `F` *represents a call to* `cond_assert(F)` *(and the subsequent call to* `eager(F)`*). Square nodes indicate that* `F` *is of the form* `ans(Atom)` *and circle nodes that* `F` *is of the form* `call(Atom)`*. Dashed nodes indicate that the fact is not new and hence that this call to* `cond_assert` *fails (and hence that the subsequent call to* `eager` *is not executed). An edge from node n_1 to n_2 labelled by c_i indicates a recursive call to* `eager/1` *due to the i^{th} clause (as chosen at line* 15 *or* 22*). Atoms labelling nodes representing calls (circles) from the j^{th} atom in the i^{th} clause body are annotated as in* `i:j-atom`*.*

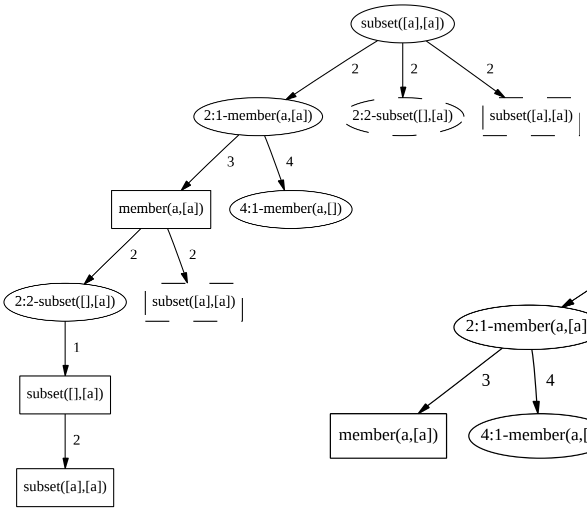

Figure 3: Eager evaluation

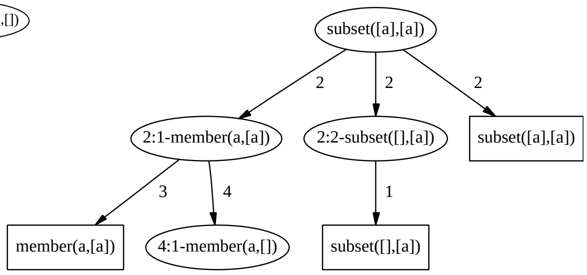

Figure 4: Optimised eager evaluation

The interpretation of the program with the initial goal follows the depth first traversal of this tree. At any stage in the computation, corresponding to a node n, the nodes that precede n (in the depth first traversal) represent the set of facts derived so far; and the nodes on the path from the root to n represent the calls to `eager/1` *currently on the recursion stack. An* `i:j` *annotated call (circle node) can be seen as a* consumer *of new answers (square nodes). Any new answer (past, present and future) which matches the corresponding body atom must be considered by this clause body to derive new instances of its head. To this end, whenever a new answer is derived, it is the task of the subsequent call to* `eager(ans(Atom))` *to make sure that all clauses containing matching calls are reconsidered.*

The root of the tree in Figure 3 corresponds to the initial goal `subset(a,[a])` *from executing line 4 in the interpreter. Next (following the depth first traversal) we choose* `c2` *at line 22 in the interpreter and have the recursive call for* `eager(call(member(a,[a])))` *at line 8. Then we choose* `c3` *(again at line 2) and assert the answer* `ans(member(a,[a]))` *(square box) at line 24.*

The code at lines 14–20 aims to ensure that if at any prior time during the execution we needed to solve the body of a clause containing a call to `member/2` *with all answers for that call then that past activity will become aware of the answer* `member(a,[a])` *which was not present at that time. With semi-naive evaluation we would apply another iteration. Here, the call to* `eager(ans(member(a,[a])))` *reactivates the evaluation of all clauses which contain a call to* `member/2`*. This generates the new call to* `subset([],[a])` *as well as the answers* `subset([],[a])` *and* `subset([a],[a])`*.*

In many cases by observing the topology of the program's call graph (while preprocessing the program) and/or by observing the structure of the evaluation tree (during interpretation time) we can avoid the propagation of new answers to (some of) the program clauses which contain matching calls. This can be done when we can guarantee that the new answers will anyway be considered by these clauses later on. In the next section we will see that this is the case also for the `subset` example and that the optimised execution gives rise to the tree in Figure 4.

3 Optimized induced eager interpreter

The aim of the optimization described in this section is to avoid, when possible, the propagation of new answers to clause bodies which contain matching calls. More specifically, the interpreter will be able to ignore reactivating at line 15 a specific clause for a new answer, or even better, to avoid even not starting the propagation process for the new answer at line 25.

Consider again the evaluation tree from Example 1 depicted in Figure 3. When the new answer `member(a,[a])` is derived, the clauses that contain a call to `member/2` are reactivated to consider this new answer. As a result we find nodes corresponding to the call to `subset([],[a])` and the answers `subset([],[a])` and `subset([a],[a])`. Notice that these three nodes occur also later in the interpretation (as dashed nodes) indicating that the same information is derived once again.

How can we detect that this will be the case? Let us consider the call to `eager(ans(member(a,[a])))`. We can observe that at the time the answer `ans(member(a,[a]))` is derived, all the calls to `member/2`, previously invoked (at line 9 for `prove_ind`) are still active on the recursion stack. This means that eventually control will return to this point where the next choice point for `prove_ind` (at line 10) will consider the new answer (at line 11).

Lemma 3.1 *We can ignore the propagation of a new answer (at line 25) if all prior matching calls are still on the recursion stack.*

In practice, we can apply Lemma 3.1 to obtain an optimisation by checking at evaluation time the conditions of the lemma.

In some cases we can determine before hand that answers do not need to be propagated. This is the case for example, with non-recursive programs because it cannot happen that we encounter a call to predicate p while processing another call for predicate p. This means that before removing the call to p from the recursion stack all of the answers for p will be available and observed when control backtracks to the second clause for `prove_ind`.

Lemma 3.2 *For programs without recursive predicate we can always ignore the propagation of a new answer (at line 25).*

The above lemmata can be refined by considering the strongly connected components (scc's) in the program's call graph.

Program	Meet			Lub			Times				
	E	OE	Δ	E	OE	Δ	E	OE			Δ
								SCC	FP	TOT	
aircraft	8521	1790	78	3864	1346	65	0.78	0.18	0.24	0.42	46
ann	3235	913	71	1309	424	67	0.33	0.02	0.12	0.14	57
asm	2331	701	69	919	349	61	0.26	0.03	0.09	0.12	53
bryant	1784	830	53	585	405	30	0.17	0.02	0.12	0.14	17
chat-80	18074	8167	54	6626	3636	45	1.92	0.21	1.04	1.25	34
chat-parser	5498	3044	44	2220	1446	34	0.59	0.07	0.38	0.45	23
essln	9599	2539	73	2497	992	60	0.77	0.16	0.26	0.42	45
flatten	1499	422	71	515	161	68	0.14	0.01	0.04	0.05	63
ili	3608	1912	46	1319	810	38	0.33	0.03	0.25	0.28	15
ime-v2-2-1	1638	651	60	404	162	59	0.13	0.00	0.07	0.07	45
lnprolog	1830	872	52	812	464	42	0.18	0.05	0.11	0.16	11
map	2273	236	89	841	89	89	0.19	0.02	0.02	0.04	78
music	995	88	91	242	33	86	0.08	0.01	0.00	0.01	86
nand	2867	1023	64	1041	525	49	0.27	0.04	0.12	0.16	40
nbody	1762	589	66	594	206	65	0.16	0.02	0.07	0.09	43
peval	2845	1212	57	1060	541	48	0.33	0.06	0.16	0.22	33
press	2852	1495	47	1193	607	49	0.27	0.02	0.16	0.18	33
qplan	1298	641	50	381	242	36	0.11	0.02	0.08	0.10	9
read	1942	808	58	645	257	60	0.17	0.02	0.08	0.10	40
reducer	1909	724	62	743	334	54	0.18	0.02	0.08	0.10	44
rubik	5133	1154	77	2224	694	68	0.49	0.07	0.14	0.21	57
semi	2112	297	85	934	153	83	0.24	0.01	0.03	0.04	82
sim	4816	1513	68	1309	560	57	0.55	0.05	0.19	0.24	56
sim-v5-2	2595	1308	49	1029	662	35	0.23	0.06	0.14	0.20	12
simple-an	5681	958	83	1855	367	80	0.57	0.02	0.11	0.13	77
trs	3795	2769	27	934	729	21	0.75	0.01	0.70	0.71	5
unify	1644	675	58	382	215	43	0.15	0.01	0.08	0.09	39

Table 1: Pos: Induced eager vs. Optimized induced eager

Lemma 3.3 *Let p be a new answer derived during eager evaluation. Then we can restrict the propagation of p (at line 15) to clauses defined in the scc of p. Moreover, if p is a non-recursive predicate then it is safe to ignore its propagation altogether (at line 25).*

4 Implementation and experimental evaluation

We have implemented the two types of optimisations described in the previous section for two abstract domains for groundness dependencies analysis: The domain *Pos* of positive Boolean functions implemented using binary decision diagrams (BDD's) as described in [1]; and the domain Def of definite Boolean functions implemented using ACI1 unification as described in [11].

The optimisations include: (1) static optimisations based on the strongly connected components of the program's call graph; and (2) evaluation time optimisations based on the introduction of flags which denote for which predicates some of the calls already derived have been removed from the recursion stack.

Program	Meet			Lub			Times				
	E	OE	Δ	E	OE	Δ	E	OE			Δ
								SCC	FP	TOT	
aircraft	9537	2458	74	3741	1346	64	0.97	0.20	0.31	0.51	47
ann	2383	885	62	993	411	58	0.76	0.03	0.46	0.49	35
asm	2057	833	59	791	389	50	0.26	0.04	0.13	0.17	34
bryant	1325	780	41	488	358	26	1.73	0.02	0.75	0.77	55
chat-80	15400	7705	49	5961	3364	43	5.43	0.23	3.16	3.39	37
chat-parser	5829	3144	46	2490	1452	41	1.79	0.08	0.95	1.03	42
essln	7482	2179	70	2049	882	56	1.20	0.17	0.43	0.60	49
flatten	1293	451	65	455	161	64	0.31	0.01	0.14	0.15	51
ili	2508	1548	38	1014	691	31	3.84	0.04	1.11	1.15	70
ime-v2-2-1	1380	749	45	318	159	49	0.21	0.01	0.16	0.17	18
lnprolog	1592	910	42	763	471	38	0.26	0.05	0.17	0.22	15
map	2405	174	92	693	87	87	0.34	0.01	0.03	0.04	87
music	616	84	86	139	28	79	3.38	0.00	0.19	0.19	94
nand	2266	1284	43	773	512	33	0.30	0.04	0.17	0.21	29
nbody	1603	627	60	519	206	60	0.26	0.02	0.10	0.12	53
peval	2014	1047	47	883	515	41	0.88	0.09	0.47	0.56	36
press	1940	1276	34	840	595	29	8.23	0.02	2.00	2.02	75
qplan	1390	626	54	436	237	45	0.14	0.01	0.08	0.09	35
read	1785	716	59	634	266	57	0.34	0.02	0.13	0.15	55
reducer	1551	879	43	614	352	42	0.40	0.02	0.24	0.26	34
rubik	3602	1150	68	1369	647	52	0.58	0.08	0.20	0.28	51
semi	2677	368	86	922	152	83	0.66	0.01	0.11	0.12	81
sim	3892	1533	60	1127	549	51	1.22	0.06	0.45	0.51	58
sim-v5-2	2536	1763	30	956	699	26	0.24	0.06	0.18	0.24	0
simple-an	4457	1043	76	1423	367	74	1.19	0.03	0.34	0.37	68
trs	3761	2497	33	932	635	31	1.11	0.02	0.88	0.90	18
unify	1401	680	51	388	210	45	0.39	0.02	0.19	0.21	46

Table 2: Def: Induced eager vs. Optimized induced eager

The experimental evaluation compares the number of abstract operations (lubs and meets) performed and the total runtimes (with and without the optimisations). The results for Pos and for Def are presented in Tables 1 and 2 where the columns indicate: **Program**, the benchmark program (available from [4]); **Meet** and **Lub**, the number of meets and lubs performed; and **Times**, the analysis time (in seconds). The sub-columns indicated by **E** and **OE** correspond to eager evaluation and optimised eager evaluation respectively; and Δ indicates the relative difference (in percentage) with and without the optimisations. In the **Times** column the time for eager evaluation is fixed point time, and the time for optimised eager evaluation distinguishes: SCC, the additional overhead to compute scc's; FP, the fixed point evaluation time; and TOT, their sum. The Δ compares the total times. From a larger suit [4] we have included in the tables results only for the benchmark programs for which non-optimised Pos evaluation requires at least 0.1 seconds.

References

[1] T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Two classes of Boolean functions for dependency analysis. *Science of Computer Programming*, 31(1):3–45, 1998.

[2] R. Barbuti, R. Giacobazzi, and G. Levi. A general framework for semantics-based bottom-up abstract interpretation of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(1):133–181, 1993.

[3] Annalisa Bossi, Maurizio Gabrielli, Giorgio Levi, and Maurizio Martelli.

[4] M. Codish. Eager evaluation benchmarks (collected from various sources). `http://www.cs.bgu.ac.il/~mcodish/Software/eager-bench.tar.gz`

[5] M. Codish, D. Dams, and E. Yardeni. Bottom-up abstract interpretation of logic programs. *Journal of Theoretical Computer Science*, 124:93–125, 1994.

[6] Michael Codish. Efficient goal directed bottom-up evaluation of logic programs. *The Journal of Logic Programming*, 38(3):354–370, 1999.

[7] Michael Codish and Harald Søndergaard. Meta-circular abstract interpretation in prolog, June 2001.

[8] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. pages 269–282, January 1979.

[9] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative modeling of the operational behavior of logic languages. *Theoretical Computer Science*, 69(3):289–318, 1989.

[10] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A Model-Theoretic Reconstruction of the Operational Semantics of Logic Programs. *Information and Computation*, 102, 1993.

[11] Samir Genaim and Michael Codish. The def-inite approach to dependency analysis. In *Proceedings of the 10th European Symposium On Programming*, volume 2028 of *Lecture Notes in Computer Science*, pages 417–431. Springer-Verlag, April 2001.

[12] N. D. Jones and A. Mycroft. Dataflow analysis of applicative programs using minimal function graphs. In *Proc. Thirteenth ACM Symp. Principles of Programming Languages*, pages 296–306. ACM Press, 1986.

[13] J. Ullman. *Principles of Database and Knowledge – Base Systems*, volume 1. Computer Science press, 1988.

[14] J. Wunderwald. Memoing Evaluation by Source-to-Source Transformation. In *Fifth International Workshop on Logic Program Synthesis and Transformation*, pages 17–32. Springer, 1995. LNCS 1048.