On the Use of Tabling for Abstract Interpretation: An Experiment with Abstract Equation Systems

(Extended Abstract)

Gerda Janssens* Konstantinos Sagonas†
Department of Computer Science
Katholieke Universiteit Leuven
B-3001 Heverlee, Belgium

Abstract

Abstract interpretation is a widely used method for the static analysis of programs. This extended abstract reports on our recent experiments with an alternative approach to implementing program analyses based on abstract interpretation. Instead of using a special-purpose system for abstract interpretation (such as `PLAI`, `GAIA`, or `AMAI`), we follow the approach advocated in [5] and use a logic programming system which supports tabling (namely XSB). To obtain the results of the analysis, we directly compile the program to be analysed into a tabled logic program such that its execution by XSB coincides with the abstract interpretation of the original program. We show how this approach is adapted to abstractions which satisfy the requirements of the framework of Bruynooghe [1] and that the approach is practical and competitive with state-of-the-art generic abstract interpretation systems implemented in Prolog. Our experiments are based on the non-trivial domain of abstract equation systems which is a parameterized domain which captures structure information together with modes, linearity and sharing.

1 Introduction

Abstract interpretation [7] is a powerful formal method for describing global data flow analyses of programs. In the context of (constraint) logic programs, abstract data flow analyses are usually realised by coupling the implementation of an abstract interpretation framework (e.g. that of Bruynooghe [1]) with a particular abstract domain and the domain-dependent abstract operations. Recent years have witnessed the development of tools for abstract interpretation: either interpreter-based generic frameworks such as `PLAI` [19] or `GAIA` [17], or abstract machine-based approaches to abstract interpretation such as `AMAI` [14] or those described in [21, 16]. We briefly review the main characteristics of these approaches: Systems specifically designed for abstract interpretation such as `PLAI` and `GAIA` are interpreter-based implementations of [1] which typically start from a query and interpret the program over a non-standard domain of data descriptions (i.e. the abstract domain) in order to obtain the program analysis results as a fixpoint. In approaches such as `AMAI`, the program to be analysed is compiled into low-level abstract machine instructions in the spirit of the WAM, and the resulting program is then executed directly by the "emulator" of the abstract machine. In either case, the result of the analysis is typically characterised by a set of tuples which give for each abstract call pattern in the program the corresponding success pattern.

*Supported by the National Fund for Scientific Research. e-mail: gerda@cs.kuleuven.ac.be.
†Supported by the K.U. Leuven Research Council. e-mail: kostis@cs.kuleuven.ac.be.

In the slightly more distant past, various researchers had proposed the use of *abstract compilation* for certain simple analysis domains: Debray and Warren [8] transformed the program to be analysed and used Prolog coupled with *extension tables* to obtain a simple mode analysis; Hermenegildo *et al.* [12] reported more experiments along those lines. Despite the conceptual elegance and simplicity of the idea, mainly for performance reasons (perhaps due to the inefficiency of the tabling mechanisms available at that time) abstract compilation was not used much and the interest shifted towards more "interpreter-based" approaches to abstract interpretation such as the generic frameworks mentioned above. More recently, the idea of abstract compilation was reexamined but only for relatively very simple types of analysis such as groundness dependency analyses in the POS domain [22, 4, 11] and some other analyses [10, 9, 6] usually in combination with a magic-set transformation and bottom-up execution.

In this extended abstract we revisit the idea of abstract compilation in the context of a complex abstract domain and show how this technique can be adapted to all abstract domains that satisfy the requirements imposed by frameworks like the one of Bruynooghe. This implies that it can be immediately used for all types of analysis for which there exist domain-dependent abstract operations (e.g., abstract unification) and support *local* (SLD) resolution [2]. Local resolution restricts at the moment of a call the (abstract) substitution to the variables in the call and at the moment of the return from the call the effects are propagated/extended to the rest of the variables of the clause in which the call occurs. The proposed abstract compilation approach can be used for normalised and non-normalised programs. Furthermore, it does not impose a particular representation for the program variables. Typical options are a non-ground representation as in `PLAI` or a ground representation as in `GAIA` and `AMAI`. The compiled program together with the domain-dependent abstract operations can be executed by any tabled LP system which is based on variance and allows for selective tabling of the predicates in the program. An off-the-shelf such system is XSB [20] which supports SLG resolution which for definite programs is similar to OLDT resolution. Using such a system, the result of the analysis (the set of abstract call and success patterns) remains in the tables.

For our experiments we used as abstract domain the abstract equations system [18, 15] which satisfies the requirements of [1] and has been coupled with `PLAI` and `AMAI` before. Abstract equation systems have been proposed as a powerful formalism to represent run-time properties of values of program variables. The abstract domain contains definite structure information and term equality together with annotations about modes, linearity and sharing.

The main advantage of the current approach is its simplicity and flexibility. For abstract domains for which abstract operations exist, one only needs to provide a simple source-to-source program transformation that generates the abstract program. This program is then directly compiled and executed and all the fixpoint computation and the management of the analysis results[1] is automatically handled by the XSB system with its efficient facilities for tabling. Abstract analysis can then be organised in several ways such as goal-dependent, goal-independent, or a combination [3]. For the experiments in this extended abstract we used a top-down goal-dependent analysis, but with our approach based on abstract compilation the kind of analysis obtained can be easily controlled through simple variations of the program transformation.

The main contributions of the extended abstract are that we show that abstract compilation can be used for any abstract domain which can be coupled with traditional abstract interpretation frameworks and that for the first time (to our knowledge) we show that the approach

[1] In particular a set of tuples of the form $(p(\bar{T}), ASentry, ASexit)$ with $ASentry$ the abstract entry substitution, i.e., the abstract substitution at the moment just before the call projected on the variables of the call $p(\bar{T})$ and with $ASexit$ the abstract exit substitution, i.e., the abstract substitution at the moment just after the call $p(\bar{T})$ and in terms of the variables of $p(\bar{T})$.

is still practical, moreover it is flexible in the sense that the transformation itself determines the kind of abstract analysis. The remaining part of this extended abstract first describes the basic compilation aspects, then gives an example of a compilation for a goal-dependent top-down analysis, and finally reports on the performance of our approach compared with that of other state-of-the-art Prolog-based tools for abstract interpretation. This direct performance comparison using the same implementation of abstract operations is to our knowledge done for the first time and thus is of independent interest.

2 Abstract Compilation in a Tabled Environment: Basic Aspects

For our purposes, we require a tabling LP system such as XSB's which is: i) based on *variance* (identity up to variable renaming) and ii) flexible in the sense that one can indicate which predicates should be tabled and which should not. In such a system one then can collect for each call to such designated predicates the *set* of their answers.

With these assumptions, the idea of our approach is to compile the program P to be analysed into a program P^* which can be directly executed by the tabled LP system when the domain-dependent abstract operations are provided (perhaps reusing them from other Prolog-based AI tools such as `PLAI` or `AMAI`), and have the underlying tabling mechanism efficiently handle the fixpoint computations and keep the results of the analysis using the tabling data structures. The idea has already been shown to result in very efficient analyses when non-ground program abstraction in conjunction with concrete or abstract tabulation can be applied for abstract interpretation; i.e., when the variables in the abstracted terms are used to collect information about the abstract substitutions [5]. When this is not possible, e.g., in abstract domains in which abstract terms (and substitutions) do not get more and more instantiated, but rather change their values from one term to the other during abstract execution, or when abstract substitutions also contain some information about relations of these variables, abstract entry and exit substitutions need to be explicitly kept as additional arguments to the tabled predicates. In such domains the computation in terms of the abstract substitutions is done by domain-dependent abstract operations: *abstract unification* is used at the entry and the exit from a clause to mimic the unification between the call and the head. Moreover, to deal with the local aspect of the resolution, each call in the body of a clause is preceded by an *abstract projection* and followed by an *abstract extension* operation.

Because of the introduction of abstract substitutions as arguments, the variant test used for the detection of "identical" calls becomes in general too strict and might miss possibilities for recognizing calls which are identical in the abstract domain (see [5] for a more detailed explanation of the problem). The remedy is to employ a new domain-dependent abstract operation, called *abstract normalisation*, to ensure a canonical representation of the terms when these are to be inserted in the tables (note that in this case the calls will be for the form $p(\bar{T}, ASentry, ASexit)$ with $ASexit$ a free variable). Some designs of abstract domains deliberately do not define nor impose normalisation [13], but use their own abstract comparison operations to test "equality" or "lower than" on abstract substitutions. Due to the hard-coded variant test of tabled evaluation a normalisation becomes necessary. Note that in the case of a ground representation for variables also the variable names need to be normalised. A sufficient condition to normalise calls is to guarantee that for $p(\bar{T_1}, ASentry_1, ASexit_1)$ and $p(\bar{T_2}, ASentry_2, ASexit_2)$ with $\bar{T_1}$ and $\bar{T_2}$ "renamings", the abstract equality test of $ASentry_1$ and $ASentry_2$ succeeds when taking into account this renaming. So, in our implementation $p(\bar{T_1}, ASentry_1, ASexit_1)$ and $p(\bar{T_2}, ASentry_2, ASexit_2)$ are normalised to canonical representations for which the variant test succeeds. Other kinds of normalisations can be defined and may affect the number of tabled information.

The answers of a call are stored in the table one after the other as they are found during the execution of the program. In the compiled programs the "interesting" part of the answers are the values of $ASexit$ in the calls $p(\bar{T}, ASentry, ASexit)$. Usually, in abstract interpretation, especially in a top-down framework, only an upper bound of the set of abstract exit substitutions is kept. To do so in our approach, a call to a domain-independent `ensure_one_exit` predicate is added as last call in the compiled clauses to force the computation of the abstract upper bound. This predicate interacts with the tabling mechanism in the following way: if $Ae_1, Ae_2, \ldots, Ae_n$ are successive abstract exit substitutions, the following upper bounds are stored as the singleton answers for $ASexit$: $Ae'_1 = Ae_1$, $Ae'_2 = as\_upperbound(Ae'_1, Ae_2), \ldots, Ae'_n = as\_upperbound(Ae'_{n-1}, Ae_n)$. The final result for $ASexit$ is then Ae'_n. We elaborate more in the following section.

3 Abstract Compilation for a Goal-dependent Top-down Analysis

We briefly present the kind of abstract compilation used to obtain a goal-dependent top-down analysis in the case of a non-ground variable representation. For a ground representation minor changes are needed and are not presented due to space limitations. For each predicate `p/`n, a Prolog "entry" predicate `p/`$n+2$ defined by a single clause is generated; its two additional arguments carry the abstract entry and exit substitutions. All that `p/`$n+2$ does, is the normalisation of the arguments of calls to `p/`n and of the abstract entry substitution, and then calls the tabled version of the predicate, `t_p/3` (note the fixed arity). For each clause of `p/`n, the compiled version of the program has a corresponding one in `t_p/3` to which domain-specific abstract operations (having prefix `as_`) for abstract *unification*, *project*, and *extend* operations are added. Let `BodyVars` denote the set of variables occurring only in the body of a clause and $vars(\bar{T})$ the set of variables in $\bar{T}$. Below we sketch the compilation of a predicate defined by a single clause: [2]

$$p(\bar{T}) :- q_1(\bar{T_1}), \ldots, q_i(\bar{T_i}), \ldots, q_n(\bar{T_n}).$$

```
p(S̄,ASentry,ASexit) :-
    as_normalise(S̄,ASentry,S̄_N,ASentryN), t_p(p(S̄_N),ASentryN,ASexit), S̄ = S̄_N.
:- table t_p/3.     % record in the XSB tables calls and answers of t_p/3
t_p(p(S̄_N),ASentryN,ASexit) :-
    as_unification_head(p(S̄_N),ASentryN,p(T̄),AS_H), % AS_H over vars(T̄)
    as_extend_init(AS_H,BodyVars,AS_1),
        ⋮         ⋮
    % code for the call to q_i in the body
    as_project(AS_i,vars(T̄_i),ASentry_i),
    q_i(T̄_i,ASentry_i,ASexit_i),
    as_extend(AS_i,ASexit_i,vars(T̄_i),AS_{i+1}),
        ⋮         ⋮
    as_project(AS_n,vars(T̄),AS_B),
    as_unification_return(p(T̄),AS_B,p(S̄_N),ASend), % ASend over vars(S̄_N)
    ensure_one_exit(ASend,t_p(p(S̄_N),ASentryN,_),ASexit).
```

[2]In the appendix we show the abstract compilation of a `reverse/2` predicate as a concrete example which also illustrates a straightforward optimisation of our abstract compilation scheme for facts.

As mentioned, we only keep one abstract exit substitution, `ASexit`, per abstract entry in the table. This step is performed by the call to `ensure_one_exit/3` in the clause of `t_p/3` which computes `ASexit` as follows: If no answer is stored yet for the tabled call `t_p(p(`$\bar{S_N}$`),ASentryN,_)`, `ASexit` equals `ASend` (the abstract substitution just computed by this clause). Otherwise, assuming that the (only) tabled answer for the call is `t_p(p(`$\bar{S_N}$`),ASentryN,ASexitT)`, if `ASend` is lower or equal (in the given domain) than `ASexitT`, then `ASexit` equals `ASexitT`, else `ASexit` is the upper bound of `ASend` and `ASexitT` and the (previous) tabled answer is substituted by the `ASexit` (i.e., `ASexitT` is removed). This is done through low-level XSB builtins (see [5] for details).

4 Experimental Results

In this section we report on the performance of our approach compared to state-of-the-art systems for generic abstract interpretation of Prolog programs. Our results are shown in Table 1. We note that they are of independent interest, as to our knowledge this is the first time where more than two AI systems are compared for performance using the same domain and implementation of abstract operations under the same machine. Sizes of benchmark programs range from 50 lines (serialize) to ≈ 500 lines (boyer, qplan, read_sn). Times reported are in seconds and are the best times obtained from conducting the experiments 10 times on a Sun Ultrasparc 2 with 2 processors (168 MHz) running SunOS 5.5.1. Only analysis times are reported as this measurement heavily dominates (accounts for more than 70% of) the total cost of the analysis on this domain (i.e. when preprocessing/compilation and collection times are also taken into account). `AMAI` is running under BIMprolog release 4.1.0, `PLAI` under SICStus 3 #5 (fastcode), and XSB is version 1.7.3 with tabling using *local* scheduling. Both BIMprolog and SICStus produce native code, while XSB is an emulator. When used as just Prolog systems, BIMprolog is around 5-6 times faster than XSB and SICStus 3 #5 fastcode is around 7-8 times faster than XSB. Tabling is implemented at the abstract machine level in XSB using tries as data-structures and

	Pair-Sharing				Set-Sharing			
Program	$\mathtt{AMAI}_{\mathrm{g}}$	$\mathtt{PLAI}_{\mathrm{ng}}$	$\mathtt{XSB}_{\mathrm{g}}$	$\mathtt{XSB}_{\mathrm{ng}}$	$\mathtt{AMAI}_{\mathrm{g}}$	$\mathtt{PLAI}_{\mathrm{ng}}$	$\mathtt{XSB}_{\mathrm{g}}$	$\mathtt{XSB}_{\mathrm{ng}}$
akl	1.06	1.04	1.63	1.45	0.71	0.66	1.03	0.88
bid_sn	0.56	0.33	0.75	0.65	0.61	0.34	0.76	0.67
boyer	4.33	3.47	5.02	4.43	4.98	4.21	6.07	5.56
browse	1.02	0.57	1.39	1.24	4.43	4.55	21.06	20.09
deriv	1.38	0.81	1.75	1.53	1.42	0.82	1.81	1.55
grammar	0.09	0.07	0.13	0.11	0.10	0.07	0.13	0.12
mycolor	0.34	0.27	0.45	0.40	0.38	0.20	0.46	0.43
peephole	3.18	2.35	3.88	3.47	3.74	3.55	5.68	5.31
qplan	15.13	14.45	23.18	21.55	3.51	2.45	4.68	3.98
rdtok	1.77	1.82	1.84	1.72	1.83	1.94	1.95	1.73
read_sn	5.52	4.05	6.57	5.61	5.88	4.34	6.83	5.87
serialize	0.46	0.28	0.75	0.63	0.51	0.32	0.81	0.70
Geom. Mean	1.30	0.97	1.73	1.52	1.35	1.01	1.95	1.74

Table 1: Performance of various AI systems on the `aeq` domain.

is "mimicked" using the `record` family of predicates in `AMAI` and `PLAI`. For fixpoint computations, XSB uses incremental stack-based algorithms implemented in C which keep track (during run-time) dependencies based on atoms encountered during execution, while both `AMAI`'s and

PLAI's fixpoint algorithms are implemented in Prolog and are based on predicate dependencies found by examining the static call graph of the program at preprocessing time.

As mentioned, we used the abstract domain of abstract equation systems. Its experimental evaluation under various parameters has been presented in detail before [18]. For each representation of program variables (ground vs. non-ground), the domain-specific abstract operations are implemented in around 4000 lines of Prolog code, are part of the PLAI distribution and have been coupled with AMAI before; they were ported to XSB with minimal effort. For this abstract domain, the abstract operations are computationally very demanding and account for a very big percentage of the the analysis times (the full version of this paper experimentally supports this claim). Note that currently PLAI imposes a non-ground representation of the program variables and AMAI a ground representation. In general, both representations have their pros and cons: the ground representation simplifies some operations as less tests are required, but requires a costly "renaming" (renumbering) step so that terms are always in a canonical form. Using XSB, we could easily experiment with both versions as the abstract compilation process is affected in a very limited way. We refer to the ground version as $\mathtt{XSB}_{\mathtt{g}}$ and to the non-ground version as $\mathtt{XSB}_{\mathtt{ng}}$ and use similar subscripts for AMAI and PLAI. In the experiments a "Pair-Sharing" and a "Set-Sharing" instantiation of the abstract equations are used. We carefully compared the analysis results of the different systems used. For most benchmarks all systems produce the same set of abstract entry-exit substitutions. An exception to this was qplan, where all systems produced some redundant entry-exit tuples (i.e., abstract substitutions which are subsumed by other analysis results or whose generation could possibly have been avoided by a better control strategy that does not propagate abstract exit information so eagerly: i.e., before the "real" upper bound is produced).

Times in Table 1 show the competitiveness of our approach, compared to high-performance special-purpose tools and more "established" approaches to abstract interpretation. Excluding the case of browse (Set-Sharing) which we believe most probably is an anomaly in the current version of XSB, of the systems used, PLAI is clearly the fastest for this domain: overall around 60–70% faster than $\mathtt{XSB}_{\mathtt{ng}}$ and 35% faster than AMAI. Note, however, that the latter number is very close to the performance difference between SISCtus 3 #5 fastcode and BIMprolog. It would be interesting to see the performance of AMAI under SISCtus and we plan to investigate this option in the full version of this paper. For XSB, times are better than the performance difference between the underlying Prolog systems; the performance is very encouraging if one takes into account that time spent in abstract operations (which do not use tabling) heavily dominates the analysis times in this domain. Currently, abstract compilation coupled with an efficient tabling mechanism such as XSB's has overall about the same performance as AMAI which is designed and optimised specifically for abstract interpretation and runs under a faster Prolog system producing native code. We find that this result shows the viability, current competitiveness and future potential of our approach (especially for abstract domains whose abstract operations are computationally less demanding than aeq and a bigger percentage of the time is spent in fixpoint computations). However, ideally one would hope for more. Perhaps a negative lesson to be learned from this experiment is that although tabling has a great potential, one should not forget that it is sometimes only a component of systems aiming to solve real-life complex problems and applications.

Acknowledgements

We are grateful to Maria García de la Banda, Manuel Hermenegildo, and the other developers and implementors of the PLAI system at UPM-CLIP for making their system available to us and for answering all our questions under "deadline" pressure.

References

[1] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *J. of Logic Program.*, 10(2):91–124, Feb. 1991.

[2] M. Bruynooghe and D. Boulanger. Abstract Interpretation for (Constraint) Logic Programming. In B. Mayoh, E. Tyugu, and J. Penjam, editors, *Constraint Programming, NATO ASI Series, Vol. F/131*, pages 228–258. Springer-Verlag, 1994.

[3] M. Codish, M. Bruynooghe, M. García de la Banda, and M. Hermenegildo. Exploiting Goal Independence in the Analysis of Logic Programs. *J. of Logic Program.*, 32(3):247–262, Sept. 1997.

[4] M. Codish and B. Demoen. Analysing Logic Programs using "Prop"-ositional Logic Programs and a Magic Wand. *J. of Logic Program.*, 25(3):249–274, Dec. 1995.

[5] M. Codish, B. Demoen, and K. Sagonas. Semantic-Based Program Analysis for Logic-Based Languages using XSB. Extended version of K.U. Leuven Tech. Rep. CW 245. Available at: `http://www.cs.kuleuven.ac.be/~kostis/Papers/papers.html.`, Dec. 1996.

[6] M. Codish and V. Lagoon. Type Dependencies for Logic Programs using ACI-unification. In *Proceedings of the 1996 Israeli Symposium on Theory of Computing and Systems*, pages 136–145. IEEE Computer Science Press, June 1996. Full version at `http://www.cs.bgu.ac.il/~codish/papers.html`.

[7] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, Jan. 1977. ACM.

[8] S. K. Debray and D. S. Warren. Automatic Mode Inference for Logic Programs. *J. of Logic Program.*, 5(3):207–229, Sept. 1998.

[9] J. Gallagher, D. Boulanger, and H. Saglam. Practical Model-Based Static Analysis for Definite Logic Programs. In J. W. Lloyd, editor, *Proceedings of the 1995 International Logic Programming Symposium*, pages 351–365, Portland, Oregon, Dec. 1995. The MIT Press.

[10] J. P. Gallagher and D. A. de Waal. Fast and Precise Regular Approximations of Logic Programs. In P. Van Hentenryck, editor, *Proceedings of the Eleventh International Conference on Logic Programming*, pages 599–613, Santa Margarita, Italy, June 1994. The MIT Press.

[11] A. Heaton, P. Hill, and A. King. Practical Analysis of Logic Programs with Delay. In N. Fuchs, editor, *Proceedings of LOPSTR'97: Logic Program Synthesis and Transformation*, LNCS, Leuven, Belgium, July 1997. Springer-Verlag.

[12] M. Hermenegildo, R. Warren, and S. K. Debray. Global Flow Analysis as a Practical Compilation Tool. *J. of Logic Program.*, 13(4):349–366, Aug. 1992.

[13] G. Janssens and M. Bruynooghe. Deriving Descriptions of Possible Values of Program Variables by Means of Abstract Interpretation. *J. of Logic Program.*, 13(2 & 3):205–258, July 1992.

[14] G. Janssens, M. Bruynooghe, and V. Dumortier. A Blueprint for an Abstract Machine for Abstract Interpretation of (Constraint) Logic Programs. In J. W. Lloyd, editor, *Proceedings of the 1995 International Logic Programming Symposium*, pages 336–350, Portland, Oregon, Dec. 1995. The MIT Press.

[15] G. Janssens, M. Bruynooghe, and A. Mulkers. Abstract Equation Systems: Descriptions and Insights. Technical Report CW 217, K.U. Leuven, Nov. 1995.

[16] A. Krall and T. Berger. A Progress Report on Incremental Global Compilation of Prolog. In K. D. Bosschere, B. Demoen, and P. Tarau, editors, *Proceedings of the ILPS'94 Post-Conference Workshop on Implementation Techniques for Logic Programming Languages*, pages 59–67, Ithaca, NY, Oct. 1994.

[17] B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Trans. Prog. Lang. Syst.*, 16(1):35–101, Jan. 1994.

[18] A. Mulkers, W. Simoens, G. Janssens, and M. Bruynooghe. On the Practicality of Abstract Equation Systems. In L. Sterling, editor, *Proceedings of the 12th International Conference on Logic Programming*, pages 781–795, Tokyo, Japan, June 1995. The MIT Press.

[19] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency using Abstract Interpretation. *J. of Logic Program.*, 13(2 & 3):315–347, July 1992.

[20] K. Sagonas, T. Swift, and D. S. Warren. XSB as an Efficient Deductive Database Engine. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 442–453, Minneapolis, Minnesota, May 1994. ACM Press.

[21] J. Tan and I.-P. Lin. Compiling Dataflow Analysis of Logic Programs. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 106–115, San Francisco, California, June 1992. ACM Press.

[22] P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Evaluation of the Domain Prop. *J. of Logic Program.*, 23(3):237–278, June 1995.

Transformed program for reverse/2

For our experiment we used the abstract equations systems as abstract domain. For this abstract domain there exist versions with ground and non-ground variable representation. We experimented with both and here we describe the compilation we use for the non-ground version using the naive reverse predicate as an example. As one of our aims was to use the *same* set of abstract operations across all systems that we compared, we did not modify the interface of the domain-dependent *as_* operations. It is conceivable that some performance gain could be achieved by avoiding some unnecessary overhaed due to the generality of the abstract operations or by slightly tailoring them to XSB.

```
% original reverse/2 predicate
reverse([],[]).
reverse([X|T],Y) :- reverse(T,RT), append(RT,[X],Y).

%transformed program
reverse(X,Y,ASentry,ASexit):-
        as_normalisation(X,Y,ASentry,ArgsN,ASentryNorm),
        CallN =.. [reverse|ArgsN],
        t_reverse(CallN,ASentryNorm,ASexit),
% for the non-ground representation of program variables
% no renaming is necessary, it suffices to unify
% the original Args with ArgsN (which also affects ASexit).
        [X,Y] = ArgsN.

:- table t_reverse/3. % t_reverse/3 is tabled by XSB
t_reverse(CallN,ASentryN,ASexit):-
      HeadAtom = reverse([],[]),
      as_unification_fact(CallN,ASentryN,HeadAtom,Res),
      ensure_one_exit(Res,t_reverse(CallN,ASentryN,_),ASexit).

t_reverse(CallN,ASentryN,ASexit):-
      HeadAtom = reverse([X|T],Y), Hv = [X,T,Y],
      as_unification_head(CallN,ASentryN,HeadAtom,AS_H),
              % CallN,ASentryN are already normalised
              % var(CallN) \= var(HeadAtom); AS_H over vars(HeadAtom)
      as_extend_init(AS_H,[RT],A2),  % RT is a variable occurring only in the body
% call to reverse
```

```
      as_project(A2,[T,RT],A3),    % [T,RT] variables in reverse/2 call
      reverse(T,RT,A3,A4),
      as_extend(A2,A4,[T,RT],A5),
% call to append
      as_project(A5,[X,Y,RT],A8),  % [X,Y,RT] variables in append/3 call
      append(RT,[X],Y,A8,A9),
      as_extend(A5,A9,[X,Y,RT],A10),
% return
      as_project(A10,Hv,A11),
      as_unification_return(HeadAtom,A11,CallN,ASend),
              % A11 over Hv; ASend over vars(CallN)
      ensure_one_exit(ASend,t_reverse(CallN,ASentryN,_),ASexit).
```