TPLP **24** (4): 863–884, 2025. © The Author(s), 2025. Published by Cambridge University Press. This is an Open Access article, distributed under the terms of the Creative Commons Attribution licence (https://creativecommons.org/licenses/by/4.0/), which permits unrestricted re-use, distribution and reproduction, provided the original article is properly cited.
doi:10.1017/S1471068424000358

*Abstract Environment Trimming**

DANIEL JURJO-RIVAS and JOSE F. MORALES

Universidad Politécnica de Madrid (UPM), IMDEA Software Institute, Madrid, Spain
(*e-mails:* daniel.jurjo@alumnos.upm.es, daniel.jurjo@imdea.org, josefrancisco.morales@upm.es, josef.morales@imdea.org)

PEDRO LÓPEZ-GARCÍA

Spanish Council for Scientific Research, IMDEA Software Institute, Madrid, Spain
(*e-mail:* pedro.lopez@csic.es)

MANUEL V. HERMENEGILDO

Universidad Politécnica de Madrid (UPM), IMDEA Software Institute, Madrid, Spain
(*e-mails:* manuel.hermenegildo@upm.es, manuel.hermenegildo@imdea.org)

submitted 20 August 2024; accepted 13 September 2024

Abstract

Variable sharing is a fundamental property in the static analysis of logic programs, since it is instrumental for ensuring correctness and increasing precision while inferring many useful program properties. Such properties include modes, determinacy, non-failure, cost, etc. This has motivated significant work on developing abstract domains to improve the precision and performance of sharing analyses. Much of this work has centered around the family of *set-sharing* domains, because of the high precision they offer. However, this comes at a price: their scalability to a wide set of realistic programs remains challenging and this hinders their wider adoption. In this work, rather than defining new sharing abstract domains, we focus instead on developing techniques which can be incorporated in the analyzers to address aspects that are known to affect the efficiency of these domains, such as the number of variables, without affecting precision. These techniques are inspired in others used in the context of compiler optimizations, such as expression reassociation and variable trimming. We present several such techniques and provide an extensive experimental evaluation of over 1100 program modules taken from both production code and classical benchmarks. This includes the Spectector cache analyzer, the s(CASP) system, the libraries of the `Ciao` system, the `LPdoc` documenter, the PLAI analyzer itself, etc. The experimental results are quite encouraging: we have obtained significant speedups, and, more importantly, the number of modules that require a timeout was cut in half. As a result, many more programs can be analyzed precisely in reasonable times.

KEYWORDS: logic programming methodology and applications, specification, analysis and verification of systems, security

* Partially funded by MICINN projects PID2019-108528RB-C21 ProCode, TED2021-132464B-I00 PRODIGY, and by the Tezos foundation. We also thank the anonymous reviewers for their very useful feedback.

https://doi.org/10.1017/S1471068424000358 Published online by Cambridge University Press

1 Introduction

Abstract Interpretation (Cousot and Cousot 1977) allows constructing sound program analysis tools that can extract properties of a program by safely approximating its semantics. Abstract interpretation-based analysis was proved practical and effective in the context of (Constraint) Logic Programming ((C)LP) (García de la Banda *et al.* 1996; Warren *et al.* 1988; Muthukumar and Hermenegildo 1990; Van Roy and Despain 1990; Le Charlier and Van Hentenryck 1994; Kelly *et al.* 1998; Warren *et al.* 1988), which was also one of its very first application areas (Giacobazzi and Ranzato 2022), and the techniques developed for CLP have also proved useful in the analysis and verification of other programming languages by using semantic translation into Constraint Horn Clauses (CHCs) (Henriksen and Gallagher 2006; De Angelis *et al.* 2021; Méndez-Lojo *et al.* 2007a). In the context of static analysis of (C)LP programs, variable sharing soon emerged as a fundamental property, which has led to very active and continuous development of variable sharing analysis domains. Sharing proved immediately necessary for ensuring correctness and precision while inferring most other useful program properties such as modes, determinacy, non-failure, and cost, among others. In fact, some early LP analyses were actually incorrect because variable sharing was not taken into account. Sharing analyses have also proven fundamental in the optimization of unification (Søndergaard 1986) and in automatic (and-)parallelization (Cabeza and Hermenegildo 1994; Hermenegildo and Rossi 1995; Pontelli *et al.* 1997; Bueno *et al.* 1999; García de la Banda *et al.* 2000). for example in parallelization it is crucial to precisely detect whether two variables are independent (i.e., they do not *share*), a variable is ground, etc. Sharing has also been used as the basis for more complex analyses for related properties such as linearity, paths, or freeness (Muthukumar and Hermenegildo 1991; Bruynooghe and Codish 1993; King and Soper 1994; Amato and Scozzari 2014; Amato *et al.* 2022). Furthermore, beyond (C)LP, sharing analysis is directly related to *aliasing* and *points-to* analyses in imperative programming, widely used in the context of languages with pointers and dynamic memory (Landi and Ryder 1992; Steensgaard 1996; Aiken *et al.* 2003; Bravenboer and Smaragdakis 2009; Navas *et al.* 2009; Rountev et al. 2001; Whaley and Lam 2002), sometimes applying directly domains stemming from (C)LP (Zanardini 2018; Méndez-Lojo and Hermenegildo 2008). In fact, (C)LP sharing analyses constituted some of the very first abstract interpretation-based pointer aliasing analyses for any programming language.

In this paper we concentrate on a popular abstract domain for variable sharing analysis: *set-sharing* analysis (Jacobs and Langen 1989; Muthukumar and Hermenegildo 1989). This domain captures which *sets* of program variables share run-time variables, encoding this information in *sharing sets*. For example, assume X, Y, and Z are the syntactic program variables that we need to track, and consider the substitution (run-time store) $\{X/f(M,K,a), Y/g(b,M), Z/g(a,b)\}$. This substitution is abstracted to the sharing set $\{\{X\},\{X,Y\}\}$, where $\{X,Y\}$ represents that there is (or, more precisely, may be) one or more variables shared in the terms to which X and Y are bound, and $\{X\}$ represents that there is one or more variables that appear only in X. Additionally, Z not appearing in any set means that it contains no variables, that is Z is bound to a ground term. Set-sharing encodes not only variable independence, that is the fact that substitutions

https://doi.org/10.1017/S1471068424000358 Published online by Cambridge University Press

that affect a given variable will not affect another one, but also groundness, grounding dependencies (e.g., the fact that if Y becomes ground then X becomes ground, but not the other way around), independence relationships, etc. This representation richness does come, however, at a price: the scalability of set-sharing domains to a wide set of realistic programs is challenging, since the number of sharing sets carried by the abstraction can be exponential in the number of variables of the clause being analyzed. This has prompted much work in improvements and alternative representations of set-sharing abstractions, with the objective of improving performance while maintaining precision as much as possible. In contrast, in this work, rather than defining new sharing abstract domains or modifying existing ones, we concentrate on developing techniques that can be incorporated in the analysis framework to address the root causes of the performance issues faced by the set-sharing domains, such as the number of variables, without affecting precision. We draw inspiration from techniques used in the context of compiler optimizations, which significantly reduce the number of variables presented in the abstractions.

The rest of the paper is structured as follows: First, Section 2 provides the necessary background, covering set-sharing abstract domains and top-down analysis. Section 3 presents our approach, offering first a program transformation that can provide an optimal solution; and second, an alternative solution based on *variable trimming* that can be applied during analysis without modifying the program. Section 4 reports our experimental evaluation and finally, Section 5 summarizes our conclusions and discusses some lines of future work.

2 Notation and preliminaries

We represent variables by uppercase letters (e.g.,: X, Y, Z, etc.) and atoms by lowercase letters (e.g.,: a, b, c, etc.). $\mathcal{P}(S)$ denotes the powerset of set S and $\mathcal{P}^0(S)$ the *proper* powerset of set S, that is $\mathcal{P}^0(S) = \mathcal{P}(S) \setminus \{\emptyset\}$. Given a term `T`, `vars(T)` denotes the set of its variables. A Constraint Logic Program (CLP) is a set of *clauses* of the form H :- $A_1, \ldots, A_n$ where $A_1, \ldots, A_n$ are *literals* that form the *body* and H is a positive literal said to be the *head* of the clause. A *substitution* is a set $\theta = \{V_1/t_1, \ldots, V_n/t_n\}$ with V_i distinct variables and t_i terms. We say that t_i is the *value* of V_i in θ. The set $\{V_1, \ldots, V_n\}$ is the *domain* of θ ($\mathtt{dom}(\theta)$).

The main idea behind *Abstract Interpretation* (Cousot and Cousot 1977) is to interpret the program over a special, abstract domain whose elements are finite representations of possibly infinite sets of actual substitutions in the concrete domain. We denote the concrete domain as D_γ, the abstract domain as D_α, and the functions that relate sets of concrete substitutions with abstract substitutions as the *abstraction* function $\alpha : D_\gamma \to D_\alpha$ and the *concretization* function $\gamma : D_\alpha \to D_\gamma$. The concrete domain is typically a complete lattice with the set inclusion order which induces an ordering relation in the abstract domain represented by $\sqsubseteq$. Under this relation the abstract domain is usually a complete lattice and $(D_\gamma, \alpha, D_\alpha, \gamma)$ is a Galois insertion/connection (Cousot and Cousot 1977). Several frameworks for abstract interpretation exist; this work focuses on *top-down frameworks*, discussed in Section 2.

https://doi.org/10.1017/S1471068424000358 Published online by Cambridge University Press

Set-Sharing Analyses. As mentioned in the introduction, in static analysis of logic programs, tracking of variables shared among terms is essential. A set of program variables $V_1, \ldots, V_m$ *share* if, in some execution of the program, they may respectively be bound to terms $\mathtt{T}_1, \ldots, \mathtt{T}_m$ such that $vars(\mathtt{T}_1) \cap \ldots \cap vars(\mathtt{T}_m) \neq \emptyset$. One of the most accurate abstract domains defined for tracking sharing information is *set-sharing* (Jacobs and Langen 1989; Muthukumar and Hermenegildo 1989). This domain captures whether at run-time there are variables sharing, condensing this information in a concise set representation. As an example, consider program variables `X`, `Y`, `Z`, `W`, `T`, and assume they are bound at run-time as follows: $\theta = \{$`X/f(M,a), Y=g(b,M,c), Z/[1,M,3], W/g(b), T/h(K,m)`$\}$. This substitution (run-time state) is represented in the *set-sharing* domain by the *sharing* abstraction $\{\{\mathtt{X,Y,Z}\}, \{\mathtt{T}\}\}$. The first element, $\{\mathtt{X,Y,Z}\}$, represents the fact that there is (at least one) variable (i.e., `M`) that appears in all of `X`, `Y`, `Z`, but not in `T` or `W`; and the second element, $\{\mathtt{T}\}$, represents that there is (at least one) variable that appears in `T` (i.e., `K`) but not in any of the others. We say that `X`, `Y` and `Z` "share," and that `T` does not "share" with `X`, `Y`, or `Z`. The fact that `W` does not appear in any set means it contains no variables and thus, it is ground. This representation also captures that there are no other variables in `X`, `Y`, or `Z` in addition to the one(s) they share, which has important implications with respect to grounding: after a program statement that grounds one of them (e.g., `Z=[1,2,3]`), we know both `X` and `Y` will also be grounded. However, grounding `T` does not ground any of `X`, `Y`, or `Z`. Other abstract domains have also been studied, notably the *pair-sharing*, which keeps track of *pairs* of variables that share. for example for the example above, its pair-sharing abstraction is: $\{(\mathtt{X},\mathtt{Y}), (\mathtt{Y},\mathtt{Z}), (\mathtt{X},\mathtt{Z})\}$. The intricacies of the relation and tradeoffs between set-sharing and pair-sharing are beyond the scope of this paper; the reader is referred to, for example, Bagnara *et al.* (1997) and Bueno and García de la Banda (2004) for further discussion of this topic. However, our subject of study in this work is set-sharing analyses.

The set-sharing domain has attracted a lot of attention in the literature and has been enhanced in different ways and extended with other kinds of information such as *freeness* or *linearity* (Muthukumar and Hermenegildo 1991; Bruynooghe *et al.* 1994; Filé 1994; King and Soper 1994; Codish *et al.* 1996; Fecht 1996; Zaffanella *et al.* 1999; Hill *et al.* 2004; Navas *et al.* 2006; Trias *et al.* 2008; Amato and Scozzari 2009; Amato *et al.* 2022).

However, the set representation, which allows the sharing domain to offer high precision, is also one of the reasons why this family of domains presents scalability challenges. A set-sharing abstraction is presented as a set of sets, each of them capturing a *possible* sharing that occurs at runtime. In cases where there is not much (or any) sharing information at runtime, more (or all the) sharing sets are possible. Given a set of variables appearing in a program being analyzed (V), the size of a set-sharing abstraction has an upper bound given by the abstraction which captures all the possible non-empty sharing sets ($\mathcal{P}^0(V)$). Thus, the size of an abstraction is, in the worst case, exponential in terms of the number of variables that appear in the program. To overcome these problems, various representations have been proposed, such as collapsing subsets of the abstraction into "cliques" (sets of variables that represent the proper powerset of those variables). These representations allow for a reduction in the size of the sharing abstraction and can improve performance, even more so when equipped with widenings (albeit then at the cost of losing precision) (Zaffanella *et al.* 1999; Navas *et al.* 2006). For example,

https://doi.org/10.1017/S1471068424000358 Published online by Cambridge University Press

the set-sharing abstraction $\{\{\mathtt{X}\},\{\mathtt{X,Y}\},\{\mathtt{Y}\},\{\mathtt{Z}\}\}$ can be represented using cliques as the tuple $(\{\{\mathtt{X,Y}\}\},\{\{\mathtt{Z}\}\})$ where the clique $\{\mathtt{X,Y}\}$ represents $\mathcal{P}^0(\{\mathtt{X,Y}\})$.

The PLAI Top-Down Analyzer. *Top-down* analyses are a family of static analyses that build an *analysis graph* starting from a series of program *entry points*. This approach was first used in analyzers such as MA3 and Ms (Warren *et al*. 1988), and matured in the PLAI analyzer (Muthukumar and Hermenegildo 1990, 1992) using an optimized fixpoint algorithm now also referred to as the *top-down algorithm* or *solver*. This algorithm was later applied to the analysis of CLP/CHCs (García de la Banda *et al*. 1996) and imperative programs (Henriksen and Gallagher 2006; De Angelis *et al*. 2021; Méndez-Lojo *et al*. b), and used in analyzers such as GAIA (Le Charlier and Van Hentenryck 1994), the CLP($\mathcal{R}$) analyzer (Kelly *et al*. 1998), or Goblint (Seidl and Vogler 2021; Tilscher *et al*. 2023).

The graph constructed by the PLAI algorithm during analysis is a finite, abstract object whose concretization approximates the (possibly infinite) set of (possibly infinite) maximal AND-trees of the concrete semantics. This approach separates the abstraction of the structure of the concrete trees (the paths through the program) from the abstraction of the *substitutions* at the nodes in those concrete trees (the program states in those paths). The first abstraction, T_α, is typically built-in, as an abstract domain of *analysis graphs*. The framework is *parametric* on a second abstract domain, D_α, whose elements appear as labels in the nodes of the analysis graph. A more detailed recent discussion can be found in De Angelis *et al*. (2021). Assume we are analyzing a literal `Goal` in the body of some clause in the program, and that `Head :- Body` is a clause in a predicate whose head unifies with `Goal`. Assume also that the substitution affecting `Goal` at the time of this call is approximated by the abstract substitution `Call` such that $\mathtt{vars(Goal)}\subseteq\mathtt{dom(Call)}$ and $\mathtt{vars(Call)}\cap(\mathtt{vars(Head)}\cup\mathtt{vars(Body)})=\emptyset$. The success (exit state) of `Goal` after having executed the above clause is represented by the abstract substitution `Success` given by:

$$\begin{aligned}
\mathtt{Success} &= \mathtt{extend}(\mathtt{Call},\mathtt{Goal},\mathtt{Prime})\\
\mathtt{Prime} &= \mathtt{exitToPrime}(\mathtt{project}(\mathtt{vars}(\mathtt{Head}),\mathtt{Exit}),\mathtt{Head},\mathtt{Goal})\\
\mathtt{Exit} &= \mathtt{entryToExit}(\mathtt{Entry},\mathtt{Head},\mathtt{Body})\\
\mathtt{Entry} &= \mathtt{augment}(\mathtt{vars}(\mathtt{Body})\backslash\mathtt{vars}(\mathtt{Head}),\mathtt{callToEntry}(\mathtt{Proj},\mathtt{Goal},\mathtt{Head}))\\
\mathtt{Proj} &= \mathtt{project}(\mathtt{vars}(\mathtt{Goal}),\mathtt{Call})
\end{aligned}$$

As an example, let `Goal` $=$ `p(A,f(B),E)`, `Call` $=\{\{\mathtt{A}\},\{\mathtt{B,C}\},\{\mathtt{A,C,D}\}\}$ (notice that `E` is ground, since it does not appear in any sharing set) and `Head :- Body` be the clause `p(f(X),f(Y),Z) :- [X|T1]=[Z,Y|T2]`, whose `Head` unifies with `Goal`. The success abstraction is computed as follows:

(i) First, the abstraction `Call` is *projected* onto the variables in `Goal` by means of the `project` function, obtaining `Proj` $=\{\{\mathtt{A}\},\{\mathtt{B}\}\}$.

(ii) Then, `callToEntry(Proj, Goal, Head)` yields an abstract substitution that represents the unification `p(A,f(B),E)=p(f(X),f(Y),Z)` (i.e., `Goal=Head`) in the context represented by `Proj`. In our example, such abstraction is $\{\{\mathtt{X}\},\{\mathtt{Y}\}\}$, where `Z` is becomes ground because it is unified with `E`, which is ground.

Algorithm 1 A schematic description of `entryToExit`

```
1: function entryToExit(Entry, Head, Body)
2:    Exit ← Entry
3:    for Literal ∈ Body do
4:      if RECURSIVE-CALL(Literal, Head) then
5:        Exit ← COMPUTE-FIXPOINT(Exit, Literal)
6:      else if PREDICATE-IN-SCOPE(Literal) then
7:        Exit ← ANALYZE-PRED(Exit, Literal)
8:      else
9:        Proj ← project(vars(Literal), Exit)
10:       MaybeAbs ← abstractLiteral(Literal, Proj)
11:       if MaybeAbs = fail then
12:         Abs_exit ← topmost(vars(Literal), Proj)
13:       else
14:         Abs_exit ← MaybeAbs
15:       Exit ← extend(Exit, Abs_exit)
16:   return Exit
```

(iii) Now, the domain of such abstraction is extended with the variables in `Body` that do not appear in `Head` (i.e., `T1` and `T2`), and the abstraction is updated accordingly by including safely approximated information. This is done by operation `augment`, which returns the `Entry` abstraction {{`X`}, {`Y`}, {`T1`}, {`T2`}}.

(iv) Then, `Body` is traversed so that each of its literals are (recursively) analyzed by procedure ENTRYTOEXIT, described in Algorithm 1. In our example, ENTRYTOEXIT(`Entry`, `Head`, `Body`) proceeds as follows:

  (a) First, the `Exit` abstraction is initialized with the current `Entry` abstraction (Line 2), and the first literal of the body is selected (Line 3), which in this case corresponds to the only literal in the body: `[X|T1]=[Z,Y|T2]`.

  (b) The PLAI framework proceeds differently depending on the kind of literal being analyzed (see Lines 4–15). Since the literal `[X|T1]=[Z,Y|T2]` is neither a recursive call nor a call to a predicate in the analysis scope, the steps in Lines 9–15 are performed as explained below.

  (c) First, the abstraction is projected onto the variables in the literal, returning {{`X`}, {`Y`}, {`T1`}, {`T2`}}) and the operation `abstractLiteral` is invoked, which captures the abstract behavior of the literal. In our example, it performs the abstract unification `[X|T1]=[Z,Y|T2]`. Since `Z` is ground, and the unification induces `X`=`Z`, the groundness information is propagated to `X`. Such unification also induces `T1`=`[Y|T2]`, which results in the creation of new sharing sets accordingly. The $\texttt{Abs}_{\texttt{exit}}$ abstraction obtained after these operations is {{`Y`, `T1`}, {`Y`, `T1`, `T2`}, {`Y`, `T2`}}.

  (d) Finally, such abstraction is used to update the previous exit abstraction by the `extend` operation (Line 15), yielding `Exit`={{`Y`, `T1`}, {`Y`, `T1`, `T2`}, {`Y`, `T2`}}.

(v) After the execution of ENTRYTOEXIT, the obtained `Exit` abstraction is projected over `vars`(`Head`) and represented in the context of the variables of `Call`.

https://doi.org/10.1017/S1471068424000358 Published online by Cambridge University Press

This is done by operation exitToPrime(project(vars(Head), Exit), Head, Goal), which captures the effects of the unification Head=Goal. In our example, it yields Prime={{B}}, since the groundness information has been propagated from X to A.

(vi) The analysis concludes with the extend(Call, Prime) operation, which *updates* the initial Call abstraction with the new inferred information in the Prime abstraction, obtaining the success abstraction: Success = {{B,C}}, where the sharing-set {A,C,D} has been deleted because A is ground, and such information is propagated to D.

As some final remarks, if no predicate head can be unified with the goal under analysis, a bottom abstraction ($\perp$) is returned (representing that the exit state is unreachable). If several clauses are available, all of them are analyzed, and a collection of *prime* abstractions $\texttt{Prime}_1, \ldots, \texttt{Prime}_m$ is obtained, one abstraction per clause, where m is the number of clauses. Then, the success abstraction is computed as Success=extend(Call,computeLub($\texttt{Prime}_1, \ldots, \texttt{Prime}_m$)), where computeLub yields the *least upper bound* of the collection of abstractions (other operators, including disjunction and widenings, are possible).

In the ENTRYTOEXIT loop (Lines 3–5), when the current literal under analysis corresponds to a recursive call (Lines 4–5), the analyzer computes a fixpoint for the associated call pattern. Such call pattern is determined by the current literal Goal and the abstraction Call representing the environment under which Goal is executed (this may require the use of a widening operation to ensure termination). A detailed discussion on the different fixpoint computation methods is outside the scope of this work and we believe that it is not necessary for understanding our approach and contribution. The reader is referred to, for example Muthukumar and Hermenegildo (1990, 1992) for more details.

In the case that the current literal is not recursive but corresponds to a call to a predicate within the analysis scope, the associated call pattern is analyzed (Lines 6–7) following steps (i) to (vi) illustrated above using our example, with the clauses that unify with the literal.

Finally, if the literal to be analyzed does not correspond to any of the two cases discussed above and the analysis domain does not know how to abstract it either (Lines 9–11), the invocation to the abstractLiteral operation returns a fail atom. The analyzer then computes the *top-most* information for vars(Literal) by calling the top-most function (Line 12). Then, the original Call abstraction is extended with such top-most information. In our example, if the abstract domain did not implement how to abstract the unification [X|T1]=[Z,Y|T2], the top-most abstraction would be $\mathcal{P}^0$({X,T1,T2,Y,Z}). Notice that this can be quite common when, for example, the analyzer has to process a call to a predicate which is imported from a library whose source code is not available, is not in the analysis scope, etc. In this case, the top-most abstraction has to be assumed to ensure correctness.

3 Environment reassociation and abstract environment trimming

Given a clause H :- $B_1, \ldots, B_n$, its *environment* is the set of all the variables appearing in theclause, defined as $\texttt{vars}(H) \cup \bigcup_{i=1}^{n} \texttt{vars}(B_i)$. As mentioned in Section 2, a set-sharing abstraction at any analysis point contains, at most, $2^V - 1$ sharing sets, where V is the

https://doi.org/10.1017/S1471068424000358 Published online by Cambridge University Press

(a) qplan/3 definition (with annotations on live variables).

```
1  qplan(X0,P0,X,P) :-             % #Abs < 2^12
2     numbervars(X0,0,I),          % I alive
3     variables(X0,0,Vg),          % I,Vg alive
4     numbervars(P0,I,N),          % I,Vg,N alive
5     mark(P0,L,0,V1),             % Vg,N,L,V1 alive and I dies
6     schedule(L,Vg,P1),           % Vg,N,L,V1,P1 alive
7     quantificate(V1,0,P1,P2),    % N,V1,P1,P2 alive and Vg,L die
8     functor(VA,f,N),             % N,P2,VA alive and V1,P1 die
9     melt(X0,VA,X),               % P2,VA alive and N dies
10    melt(P2,VA,P).               % P2,VA alive
```

```
1  qplan(X0,P0,X,P) :- % #Abs < 2^6
2      qplan_aux1(X0,P0,P2,VA),
3      qplan_aux2(X0,X,P,P2,VA).
4
5  qplan_aux1(X0,P0,P2,VA) :- % #Abs < 2^5
6      qplan_aux11(X0,P0,P2,N),
7      functor(VA,f,N).
8
9  qplan_aux11(X0,P0,P2,N) :- % #Abs < 2^5
10     qplan_aux111(X0,P0,N,Vg),
11     qplan_aux112(P0,P2,Vg).
12
13 qplan_aux111(X0,P0,N,Vg) :- % #Abs < 2^5
14     qplan_aux1111(X0,Vg,I),
15     numbervars(P0,I,N).
16 qplan_aux1111(X0,Vg,I) :- % #Abs < 2^3
17     numbervars(X0,0,I),
18     variables(X0,0,Vg).
19
20 qplan_aux112(P0,P2,Vg) :- % #Abs < 2^5
21     mark(P0,L,0,V1),
22     qplan_aux1121(P2,Vg,L,V1).
23
24 qplan_aux1121(P2,Vg,L,V1) :- % #Abs < 2^5
25     schedule(L,Vg,P1),
26     quantificate(V1,0,P1,P2).
27
28 qplan_aux2(X0,X,P,P2,VA) :- % #Abs < 2^5
29     melt(X0,VA,X),
30     melt(P2,VA,P).
```

(b) Optimal transformation of qplan/3 with the minimal number of non-existential variables in environments. The maximum size of the set-sharing abstractions is included as a comment.

Fig 1. qplan/3 predicate and its environment trimming-based transformation.

size of the environment. Intuitively, a clause should be faster to analyze if it has fewer variables. Since it is not possible to artificially reduce the number of variables present in a clause, we propose two techniques: rearranging the literals of the body into new predicates, and modifying the domain of the abstraction without altering the clause being analyzed.

3.1 Environment reassociation

Expression reassociation (Briggs and Cooper 1994), also known as reordering or restructuring, is a technique that involves changing the grouping of terms in an expression without altering its overall value. It is used for optimization purposes, such as improving performance, reducing floating-point errors, eliminating redundancies, etc.

Given a clause H :- $B_1, \ldots, B_n$, a *partition* is a collection $P_1, \ldots, P_s$ such that each P_j is a subsequence of *consecutive* literals, $P_j = B_i, B_{i+1}, \ldots, B_k$ with $1 \leq i < k \leq n$, such that given P_j,P_{j+1} with $j \in \{1, s-1\}$: a) if the first element of P_{j+1} is B_k, then the last element of P_j is B_{k-1}, b) $B_1 \in P_1$ and c) $B_n \in P_s$.

Folding each of the literals encapsulated by each P_i generates new auxiliary predicates whose environments are smaller (or equal) than the environment of the original predicate. This procedure can be repeated recursively over the auxiliary predicates obtaining a new collection of predicates with reduced environments. Finally, our focus is to find an *optimal* partition. An optimal partition is a (possibly recursive) partition where the number of variables in the environments of each of the auxiliary predicates generated is minimal.

Consider the clause of predicate qplan/3 shown in Figure 1a (the meaning of the comments will be explained later). We refer to body literals by $\mathtt{L}_i$, where i is a position in the body of the clause. For example, $\mathtt{L}_4$=mark(P0,L,0,V1).

https://doi.org/10.1017/S1471068424000358 Published online by Cambridge University Press

Algorithm 2 Functions to detect live and dead variables.

1: **function** LIVE-VARS(LiveVars, B)
2: $\quad$ LitVars $\leftarrow$ vars(B)
3: $\quad$ **return** $\{X \in \texttt{LitVars}\ s.t.\ X \notin \texttt{LiveVars}\}$
4: **function** DEAD-VARS(LiveVars, HVars, $\{\texttt{B}_i, \ldots, \texttt{B}_n\}$)
5: $\quad$ FutVars $\leftarrow \bigcup_{j=1}^{n} \texttt{vars}(\texttt{B}_j)$
6: $\quad$ **return** $\{X \in \texttt{LiveVars}\ s.t.\ X \notin \texttt{FutVars} \wedge X \notin \texttt{HVars}\}$

The collections P_1=$\texttt{L}_1, \ldots, \texttt{L}_3$, P_2=$\texttt{L}_4, \ldots, \texttt{L}_6$, and P_3=$\texttt{L}_7, \ldots, \texttt{L}_9$ define a partition of the predicate `qplan`. This partition, when folded, generates three auxiliary predicates: `aux1(P0,X0,Vg,N)` :- $\texttt{L}_1, \ldots, \texttt{L}_3$, `aux2(P0,Vg,P2)` :- $\texttt{L}_4, \ldots, \texttt{L}_6$, and `aux3(N,X0,P2,X,P)` :- $\texttt{L}_7, \ldots, \texttt{L}_9$, with environments containing 5, 6, and 6 variables respectively. Finally, Figure 1b presents a transformation of `qplan` obtained by recursively reassociating the predicate. Each auxiliary clause is annotated with the worst case size for any set-sharing abstraction traversing it. While in the original program the maximum size is 2^{12}-1, it is reduced to 2^6-1 in the transformed program.

3.2 Abstract environment trimming

The technique of environment reassociation described before, allows obtaining, given a clause, a collection of clauses where the number of variables appearing in each one is reduced with respect to the original one. However, applying transformations over the program under analysis may not always be desirable. In this section we provide an alternative approach, where the domain of abstractions is dynamically modified as the analysis of a clause processes each body literal. Although the resulting abstraction domains might not be optimal, this technique avoids the transformations, since such dynamic domain modifications are performed as part of the abstract operations. Given a clause Head :- $\texttt{B}_1, \ldots, \texttt{B}_n$ a variable X is a *live variable* while analyzing $\texttt{B}_i$ (that we refer to as the analysis point $\texttt{B}_i$) if $\texttt{X} \in \texttt{vars}(\texttt{Head})$ or there exists j, k $1 \leq j \leq i \leq k \leq n$ such that X belongs to both $\texttt{vars}(\texttt{B}_j)$ and $\texttt{vars}(\texttt{B}_k)$. Conversely, a variable X is a *dead variable* at analysis point $\texttt{B}_i$ if it does not appear in the body after such point, that is $\texttt{X} \notin \bigcup_{j=i}^{n} \texttt{vars}(\texttt{B}_j)$. Our definition of liveness is similar to imperative programming, with the difference that variables are not reassigned and that variables become live on the first appearance, since logic variables do not need to be declared in the body of a clause (Aho *et al.* 2006, pp. 608–610). Figure 1a shows, at each program point, which body variable lives or dies. In that sense it is reminiscent of the concept of *environment trimming* used in the Warren Abstract Machine (Warren 1987; Ait-Kaci 1991), but more general, since variables also come in instead of only out, and the objective is of course different: optimizing abstract operations *versus* saving stack space.

Algorithm 2 presents the operations $LIVE-VARS$ and $DEAD-VARS$ to obtain the variables becoming alive and the ones which are dead at a given analysis point. $LIVE-VARS$ receives the set of current live variables (LiveVars), and the literal that is going to be analyzed (B), and returns the set of new live variables at that analysis

https://doi.org/10.1017/S1471068424000358 Published online by Cambridge University Press

Algorithm 3 Version of `entryToExit` dynamically modifying the abstraction domain.

```
 1: function entryToExit(Entry, Head, {B_1, ..., B_m})
 2:   HVars ← vars(Head)                                        ▷ Obtain the Head Variables
 3:   LiveVars ← HVars                                          ▷ Initialize the Live Variables
 4:   Exit ← Entry
 5:   for i ∈ {1, ..., m} do
 6:     NLiveVars ← LIVE-VARS(dom(Entry), B_i)                  ▷ Get the New Live Vars.
 7:     LiveVars ← LiveVars ∪ NLiveVars                          ▷ Update the Live Variables
 8:     Entry^aug ← augment(NLiveVars, Entry)                    ▷ Augment the Abstraction
 9:     if RECURSIVE-CALL(Literal, Head) then
10:       Exit ← COMPUTE-FIXPOINT(Exit, B_i)
11:     else if PREDICATE-IN-SCOPE(Literal) then
12:       Exit ← ANALYZE-PRED(Exit, B_i)
13:     else
14:       Proj ← project(vars(B_i), Exit)
15:       MaybeAbs ← abstractLiteral(B_i, Proj)
16:       if MaybeAbs ≠ fail then
17:         Abs_exit ← topmost(vars(B_i), Proj)
18:       else
19:         Abs_exit ← MaybeAbs
20:       Exit ← extend(Exit, Abs_exit)
21:       DeadVars ← DEAD-VARS(LiveVars, HVars, {B_j}_{j=i+1}^m)    ▷ Get Dead Vars.
22:       LiveVars ← LiveVars\DeadVars                           ▷ Update Live Variables
23:       Exit ← project(LiveVars, Exit)                         ▷ Restrict to the Live Variables
24:   return Exit
```

point, that is the variables that appear in the literal but were not in `LiveVars`. The other operation, $DEAD-VARS$, takes as input the set of current live variables (`LiveVars`), the variables of the head (`HVars`), and the set of literals that have not been analyzed yet ($\{\mathtt{B}_i, \ldots, \mathtt{B}_n\}$), and produces as output a set containing the variables of `LiveVars` that do not appear in any of the literals nor belong to the clause head. More efficient procedures to determine the liveness of variables are possible. However, we checked experimentally that they do not offer substantial improvements, and thus we decided to keep these simpler definitions. With these auxiliary operations, the analyzer can determine, at each analysis point, whether a variable is live or dead. With this information, it is possible to restrict the domain of the abstractions to the set of live variables. To do so, the computation of the `Success` abstraction is modified slightly:

$$\begin{aligned}
\mathtt{Success} &= \mathtt{extend}(\mathtt{Call}, \mathtt{Goal}, \mathtt{Prime})\\
\mathtt{Prime} &= \mathtt{exitToPrime}(\mathtt{project}(\mathtt{vars}(\mathtt{Head}), \mathtt{Exit}), \mathtt{Head}, \mathtt{Goal})\\
\mathtt{Exit} &= \mathtt{entryToExit}(\mathtt{Entry}, \mathtt{Head}, \mathtt{Body})\\
\mathtt{Entry} &= \mathtt{callToEntry}(\mathtt{Proj}, \mathtt{Goal}, \mathtt{Head})\\
\mathtt{Proj} &= \mathtt{project}(\mathtt{vars}(\mathtt{Goal}), \mathtt{Call}).
\end{aligned}$$

In this case, the `Entry` abstraction is obtained by directly computing `callToEntry`, instead of by augmenting the result of the `callToEntry` invocation, as was done in the schema presented in Section 2. Thus, in this approach, the domain of the `Entry`

https://doi.org/10.1017/S1471068424000358 Published online by Cambridge University Press

abstraction is exactly the set of head variables (which are the only variables alive at that analysis point). Finally, the function `entryToExit` presented in Algorithm 1 is modified so that it keeps the abstraction defined only over the live variables. Such a modified version is described by Algorithm 3. There, a set containing the live variables is carried around while analyzing a clause body ($\{B_1, \ldots, B_m\}$). Such set is initialized with the variables of the clause head (Line 3), and updated by adding new variables to it as they become alive (Lines 6-7) and by removing the dead variables (Lines 21-22). Accordingly, the abstraction is augmented with the new live variables (Line 8) and reduced when some of them die (Line 23).

4 Experimental evaluation

We have conducted an extensive experimental study to assess the benefits of the techniques proposed, to which we will refer to here as *reassociation* and *trimming* for short. In particular, we measured, for a variety of programs, the effects of applying both techniques to a number of abstract domains that use set sharing: the classical set-sharing domain, `share` (Muthukumar and Hermenegildo 1989), the combination of sharing and freeness, `shfr` (Muthukumar and Hermenegildo 1991), and the combination of sharing and freeness including cliques, `shfr-clique` (Navas *et al*. 2006); the latter is the most efficient of the three, while `share` and `shfr` are, in general, more precise but slower. All experiments were performed on Debian/GNU Linux 12 (bookworm) 64bit (amd64) with 128 GB RAM, and 800 GB of disk space. We focus on analysis times since precision is unchanged.

We start by showing in Table 1 the detailed results for one of our benchmarks, the `LPdoc` documenter (Hermenegildo 2000; Hermenegildo and Morales 2011), which is a relatively large, real-world application, and whose results are typical. The `LPdoc` source code is composed of 26 modules that exhibit different challenges: for some of them analysis terminates in times that range from a few milliseconds to several minutes, while others cannot be analyzed, either because of a timeout, set for these experiments at 5 min per module, or by running out of memory. In either of these cases we will say that the analysis fails. For each abstract domain, Column **TC** shows the *total* time that the classical domain requires to analyze the modules, columns **TR** and **TT** show the *total* analysis times when applying reassociation and trimming respectively to the corresponding classical domain, and columns ρ**R** and ρ**T** present the relative speedup computed as **TC**/**TT** and **TC**/**TR**. *Total* analysis times are the addition of the times for the loading of the module, the pre-processing (including the transformation required by the reassociation method), and the actual analysis time. Some statistics are included at the bottom of the table: the total number of modules (**Mods**) and the number of modules for which the different analyses fail, for the classical approach (**FC**), when applying reassociation (**FR**), and when applying trimming (**FT**). Finally, μ**T** and μ**R** show the mean of the speedups obtained. We can see that applying reassociation and trimming allows analyzing a significant number of modules that could not be analyzed with the classical techniques. In particular, when applied to the `shfr-clique` domain they achieve a full analysis of the `LPdoc` source code. The mean speedups show that trimming outperforms

https://doi.org/10.1017/S1471068424000358 Published online by Cambridge University Press

https://doi.org/10.1017/S1471068424000358 Published online by Cambridge University Press

Table 1. *Analysis times and statistics for the source code of* `LPdoc`*. Times are in mS*

	share					shfr					shfr-clique				
module	TC	TR	ρR	TT	ρT	TC	TR	ρR	TT	ρT	TC	TR	ρR	TT	ρT
html	23,485	1377	17.04	2646	8.87	20,858	1373	15.19	2516	8.29	1522	1293	1.18	1532	0.99
html˙assets	6.20	8.20	0.76	6.75	0.92	7.25	6.46	1.12	7.79	0.93	13.70	12.42	1.10	14.25	0.96
aux	4.12	4.60	0.90	4.09	1.01	4.51	5.17	0.87	4.58	0.98	5.57	6.75	0.83	5.82	0.96
man	timeout	32,944	-	64,490	-	timeout	29,235	-	60,521	-	23.74	27.23	0.87	23.32	1.02
doctree	10617	7095	1.50	5807	1.83	6426	5443	1.18	1898	3.39	4485	555.77	8.07	568.98	7.88
docmod	0.82	0.76	1.07	0.85	0.97	0.75	0.88	0.85	0.77	0.97	0.75	0.97	0.87	0.70	1.08
images	132,364	134,906	0.98	134.10	987.03	136,011	130,448	1.04	123.82	1098	16.95	14.83	1.14	52.66	0.32
messages	3.75	3.58	1.05	4.29	0.87	4.22	4.39	0.96	4.32	0.98	6.08	5.47	1.06	5.96	1.02
structure	15.59	12.63	1.23	11.54	1.35	17.77	14.07	1.26	13.27	1.34	21.44	16.34	1.31	18.24	1.18
lpdoc˙sing.	342.13	17.53	19.51	12.58	27.19	331.32	19.85	16.96	14.21	23.31	533.04	29.56	18.03	22.26	23.94
docmaker	40.49	34.69	1.17	33.97	1.19	41.71	46.96	0.89	33.44	1.25	60.92	76.91	0.79	69.32	0.88
bibrefs	119,737	64,348	1.86	91,029	1.32	74,779	12,024	6.22	40,949	1.83	1560	590.09	2.64	780.31	2.00
html˙templ.	124.43	34.46	3.61	26.27	4.74	121.22	25.99	4.66	22.03	5.50	41.23	42.65	0.97	46.61	0.88
lpdoc˙help	3.01	3.56	0.85	2.73	1.10	3.84	4.22	0.91	3.01	1.28	5.34	5.82	0.92	4.43	1.21
texinfo	timeout	timeout	-	timeout	-	timeout	timeout	-	timeout	-	timeout	171.50	-	1440	-
comments	35.68	38.58	0.93	22.50	1.59	25.90	24.91	1.04	23.46	1.10	44.29	49.59	0.89	36.56	1.21
errors	0.79	0.95	0.83	0.74	1.07	0.97	0.76	1.29	0.76	1.29	0.00	1.06	0.09	1.14	0.09
parse	55,513	54,284	1.02	18,022	3.08	49,511	49,577	1.00	15,353	3.22	2381	2404	0.99	2492	0.96
autodoc	timeout	timeout	-	timeout	-	timeout	timeout	-	timeout	-	timeout	6242	-	6864	-
index	1856	395.24	4.70	685.04	2.71	1298	344.32	3.77	562.93	2.31	93.31	66.97	1.39	95.55	0.98
filesystem	80.38	41.11	1.96	53.51	1.50	80.32	51.71	1.55	51.93	1.55	59.12	65.75	0.90	63.98	0.92

https://doi.org/10.1017/S1471068424000358 Published online by Cambridge University Press

Table 1. *Continued*

	share					shfr					shfr-clique				
module	TC	TR	ρR	TT	ρT	TC	TR	ρR	TT	ρT	TC	TR	ρR	TT	ρT
doccfg	3.06	3.53	0.87	2.18	1.40	4.03	3.75	1.08	2.49	1.62	3.03	3.54	0.85	0.44	6.82
refsdb	timeout	1507	-	2991	-	97,871	1093	89.54	572.88	170.84	26,853	1433	18.73	393.76	68.20
lookup	11.68	12.04	0.97	6.98	1.67	14.44	12.54	1.15	4.01	3.60	25.30	18.14	1.39	13.09	1.93
version	0.28	0.23	1.23	0.30	0.92	0.32	0.30	1.05	0.31	1.04	0.28	0.32	0.88	0.30	0.93
settings	15.36	17.97	0.85	17.19	0.89	19.17	22.26	0.86	20.03	0.96	37.77	42.99	0.88	41.01	0.92
nil	2.11	2.60	0.81	2.27	0.93	3.52	2.26	1.56	2.43	1.45	1.56	1.63	0.96	1.69	0.92
state	38,271	4745	8.06	4782	8.00	38,006	4647	8.25	4733	8.03	1177	1103	1.07	1103	1.07

	share	shfr	shfr-clique
Mods	27	27	27
FC	4	5	2
FR	2	2	0
FT	2	2	0
μ**R**	3.07	8.13	2.64
μ**T**	44.26	53.82	4.97

reassociation in this application. This is due to the significant improvements that trimming brings to the modules **images** and **refsdb**. Specifically, in the case of **images**, the trimming approach significantly reduces analysis times for the `share` and `shfr` domains, resulting in speedups comparable to those achieved by `shfr-clique`. However, trimming introduces a slow-down when applied to the `shfr-clique` domain. Conversely, reassociation does not provide any significant changes. In the **texinfo** module, reassociation performs 10 times faster than trimming, while in the remaining modules, both methods behave similarly. Our hypothesis is that the overhead introduced by the transformation is what causes reassociation to slightly under-perform in some cases.

The proposed methods were also evaluated across a set of classic benchmarks (see Table C.1 in the appendices). The benchmarks include a variety of examples, ranging from simple predicates that feature only direct recursions, such as **qsort** and **append**, to more complex cases with mutual recursions and elaborate aliasing. Some benchmarks are extracted from real-world programs. For example, **aikal** is part of an analyzer for the AKL language, while **read** and **rdtok** are Prolog parsers. Additionally, parts of actual programs are also included, such as **ann** (the &-Prolog parallelizer), **qplan** (the core of the Chat-80 application), and **witt** (a conceptual clustering application). As expected, in these comparatively less challenging programs the advantages obtained are smaller, but they are still significant. For instance, using trimming and reassociation bring mean relative speedups of 1.1 and 1.06 respectively for the `share` domain; 0.95 and 1.5 for `shfr`; and 2.2 and 1.97 for `shfr-clique`.

Most of these modules already required small analysis times (less than 0.1 s, which may indicate they offer fewer opportunities for optimization). The improvements are most significant for `qplan`, which had the largest analysis times to begin with: 5.7 s (`share`), 1 s (`shfr`), and 5 s (`shfr-clique`). In this case, trimming and reassociation bring analysis times of 1.1 and 0.5 s, respectively, with `share`; and 0.3 and 0.2 s with `shfr`. Moreover, with `shfr-clique` both techniques bring the time down to 0.1 s (50× speedup, the largest one). The full experimental results can be found in Appendix C.1.

Given the positive results, in a second phase of our experimentation we decided to greatly expand the code base used to measure the impact of applying abstract environment trimming, in order to explore whether the improvements obtained for the classic benchmarks and the `LPdoc` application would translate much more widely. All in all we have analyzed around 1200 program modules. These include, in addition to the previously mentioned `LPdoc` documenter (28 modules) and classic benchmarks (26 modules): all the libraries distributed with the `Ciao` system (Hermenegildo *et al*. 2012) (comprising more than 1000 modules), `CiaoPP`'s program analyzer (Hermenegildo *et al*. 2003) PLAI (56 modules), s(CASP) (Arias *et al*. 2018), a top-down interpreter for ASP programs with constraints (44 modules), and Spectector (Guarnieri *et al*. 2020), a tool for automatically detecting leaks caused by speculatively executed instructions in x64 assembly programs (15 modules). The ≈1000 library modules in the standard `Ciao` distribution come from many sources including libraries ported from SICStus (Carlsson and Mildner 2012); libraries common with Yap (Santos Costa *et al*. 2012), XSB (Swift and Warren 2012), SWI (Wielemaker *et al*. 2012), etc., including those from the Prolog Commons project (https://prolog-commons.org/); the classic libraries from O'Keefe and those for Constraint Logic Programming over Rationals and Reals (Holzbaur 1995); libraries for

https://doi.org/10.1017/S1471068424000358 Published online by Cambridge University Press

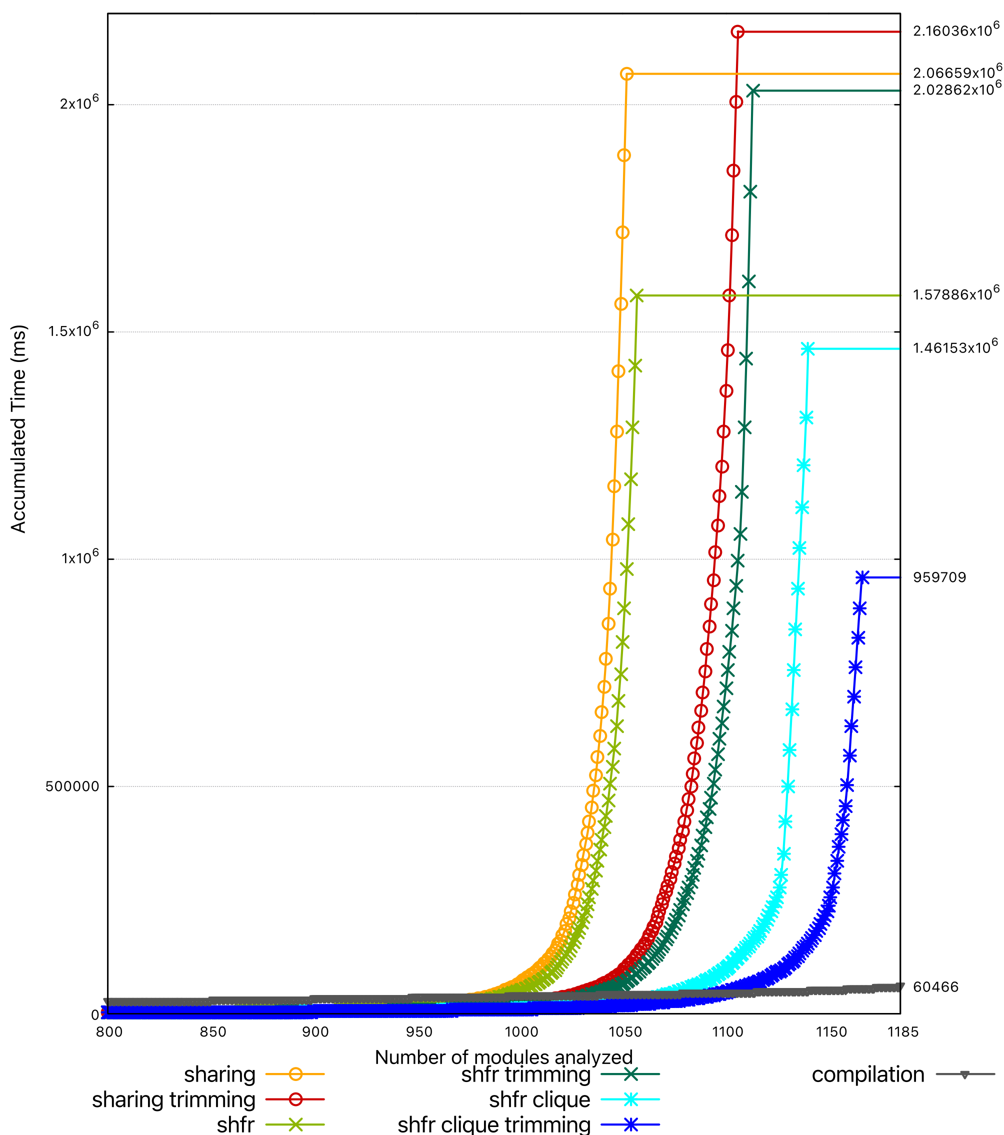

Fig 2. Cactus plot aggregating all the benchmarks (1185 modules). Times are in mS.

implementing language extensions such as functional syntax or the `Ciao` assertion language; etc., developed by around 100 different programmers. We argue that this selection of benchmarks constitutes a good representation of real-life code written in Prolog. The detailed results of these experiments can be found in Appendix C, while here we present them in a more manageable aggregated form.

We first present in Figure 2 a *cactus plot* of the results. Cactus plots display, on the X-axis, the number of benchmarks that can be analyzed, that is those for which the analysis does not time out or run out of memory, and, on the Y-axis, the *accumulated* analysis time. The plots for each abstract domain and analysis technique are generated

https://doi.org/10.1017/S1471068424000358 Published online by Cambridge University Press

by taking all the analysis times, *sorting them in increasing order*, and then plotting them following the formula (i, t_i) where t_i is the *accumulated* time in the i-th position. This way, the analysis time of the benchmark corresponding to the $i+1$-th position is $t_{i+1} - t_i$. Darker colors represent abstract domains with trimming, while lighter colors represent domains with classic analyses. Empty circles correspond to the classic set-sharing domain, crosses to `shfr`, and stars to `shfr-clique`. The module compilation times are given for comparison, represented by gray triangles –resulting in the gray vertical line. The plot has been zoomed in to exclude points with X-values (numbers of benchmarks) less than 800, as the most interesting information is in the more challenging benchmarks beyond, that require the highest analysis times. Also, plots corresponding to modules for which the analysis fails are excluded. This figure allows us to observe that when the domains are used with abstract trimming, they perform better than their classical counterparts. Specifically, `share` and `shfr` reduce the number of modules they are unable to analyze from 134 and 130 to 81 and 74, respectively, corresponding to a 39.55% and 43.07% improvement. For `shfr-clique` applying abstract environment trimming results in failure to analyze only 21 modules versus the 47 that the classical approach failed to analyze, while reducing the accumulated time by 8.36 min. This translates to a reduction in timeouts by 55.32% while reducing the accumulated time by 34.3%. We have also obtained mean speedups of 5.82, 5.91, and 22.08 when analyzing with `share`, `shfr` and `shfr-clique` respectively. Moreover, the number of modules that can be analyzed in a time lower than the compilation time also increases.

The cactus plot shows how the analysis results accumulate. In order to show how these results relate individually, Figure 3 presents a scatter plot displaying the time required to analyze each module. Given a point (x, y), the value in x corresponds to the time required by the classical analysis to analyze a given module, while y corresponds to the time required to analyze that same module using abstract trimming. Modules that are not analyzed with the classic approach or with both are not displayed. If the points are close to the orange line, it means that the times are very similar; if they are above the line, it means that abstract trimming introduces an overhead; if they are below, abstract trimming speeds up the analysis. To complement this information, Figure 4 presents a scatter plot displaying a comparison between the time required to analyze each module with the classical analysis (X-axis) and the speedup obtained by applying the abstract trimming technique (Y-axis). The speedup values range from very close to zero to significantly larger numbers (see the 0.09 and the 1098 speedup obtained by abstract trimming when analyzing the "errors" module with sharefree-clique and the "images" module with shfr, as shown in Table 1). To better represent these values, we have applied a base 10 logarithm to the resulting speedups. Thus, values between 0 and 1 become negative (with larger absolute values the closer they are to 0), while values greater than 1 are scaled in the positive plane. The most significant performance improvements are observed in the right-most side of the figures, corresponding to the modules where the classical approach takes more time. Conversely, in cases where the classical approach is very fast (left-most part of the figures), the technique of abstract trimming does not yield many speedups but introduces some overheads, which are however small. Another observation is that in most benchmarks, the analysis times are quite low, but of course our target has been the rest that present significant challenges.

https://doi.org/10.1017/S1471068424000358 Published online by Cambridge University Press

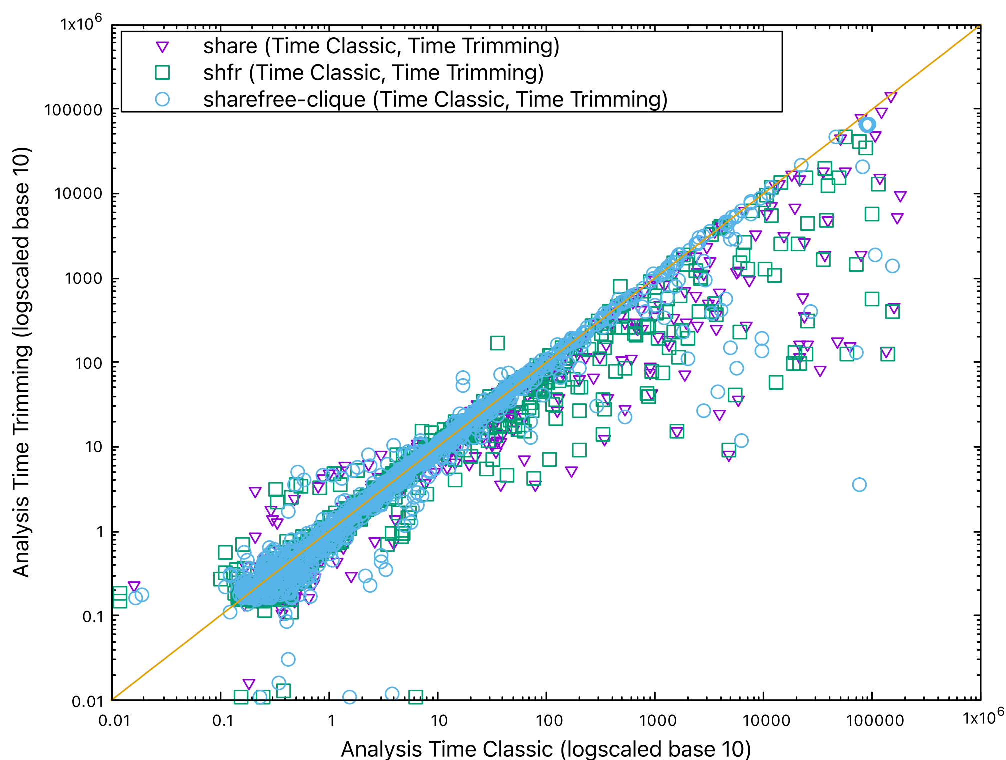

Fig 3. Scatter plot comparing absolute analysis times (in mS).

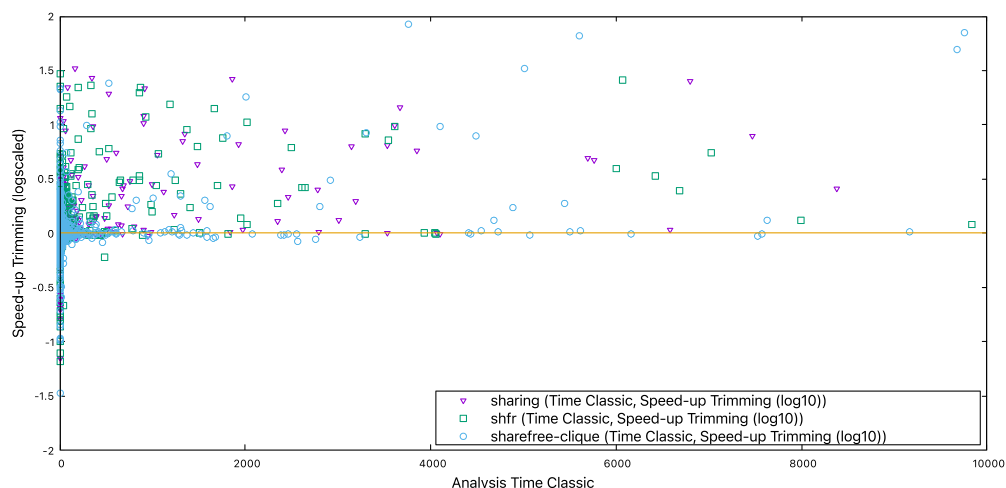

Fig 4. Scatter plot showing classic analysis time (in mS) vs. speedup obtained (logscaled base 10).

To concentrate on this set we have collected the speedup results considering only the benchmarks taking more than 0.1, 0.5, and 1 s when analyzed with `share` (which represents the slower domain). These results are presented in Table 2. For example, the

https://doi.org/10.1017/S1471068424000358 Published online by Cambridge University Press

Table 2. *Complete statistics*

	share	shfr	shfr-clique
Modules	1186	1186	1186
FC	134	130	47
FT	81	74	21
μT	5.82	5.91	22.08
μT>1s (70-74-160)	62.13	57.64	146.15
μT>0.5s (93-97-180)	47.86	45.05	130.66
μT>0.1s (143-147-230)	32.11	30.60	102.65
μT<0.1s (909-913-999)	1.68	2.10	24.12

row starting with "μ**T**>1s (70-74-160)" shows the mean speedup for the benchmarks successfully analyzed such that the time required to analyze them with `share` is greater than 1 s or `share` times out. The results show that in the case of `share`, 70 of these more challenging modules are successfully analyzed with both trimming and reassociation (note that both approaches need to succeed in order to compute the means), 74 in the case of `shfr`, and 160 in the case of `shfr-clique`. Similar very positive results are obtained for the other cases.

5 Conclusions

We have proposed a number of techniques for addressing the scalability problems inherent to set-sharing analyses. We have focused on the root of the problem: the potentially exponential dependency of the size of the abstractions on the number of variables. We have cast this problem as an instance of expression reassociation and provided an optimal solution using program transformations. Additionally, we have proposed a practical solution that can be integrated into top-down analyzers, based on the liveness of variables in the body of the clause being analyzed. We have conducted an extensive experimental evaluation of over 1100 program modules taken from both production code and classical benchmarks. We have obtained significant speedups, and, more importantly, the number of modules that require a timeout was cut in half. As a result, many more programs can be analyzed precisely in reasonable times. We believe that the results obtained suggest that the proposed local technique improves significantly the scalability of set-sharing analyses, and can thus enhance the practicality of top-down set-sharing analysis for production code. As a possible avenue for future work, note that the definition of live variables used in this work is local to each clause of the predicate being analyzed. Future lines of work could explore a more global notion that also considers the calls to predicates within the clause under analysis. This will presumably incur additional cost but could also possibly allow further reduction in the size of the domains of the sharing abstractions.

Competing interests

The authors declare that they have no competing interests.

https://doi.org/10.1017/S1471068424000358 Published online by Cambridge University Press

Supplementary material

To view supplementary material for this article, please visit https://doi.org/10.1017/S1471068424000358

References

Aho, A. V., Lam, M. S., Sethi, R. and Ullman, J. D. 2006. *Compilers: Principles, Techniques, and Tools*, 2nd ed. Addison-Wesley Longman Publishing Co., Inc., USA.

Aiken, A., Foster, J. S., Kodumal, J. and Terauchi, T. 2003. Checking and inferring local non-aliasing. In *Proceedings of the ACM SIGPLAN. 2003 Conference on Programming Language Design and Implementation 2003*, San Diego, California, USA. ACM, 129–140.

Ait-Kaci, H. 1991. *Warren's Abstract Machine, A Tutorial Reconstruction*. MIT Press.

Amato, G., Meo, M. C. and Scozzari, F. 2022. The role of linearity in sharing analysis. *Mathematical Structures in Computer Science* 32, 1, 44–110.

Amato, G. and Scozzari, F. 2009. Optimality in goal-dependent analysis of sharing. *Theory and Practice of Logic Programming* 9, 5, 617–689.

Amato, G. and Scozzari, F. 2014. Optimal multibinding unification for sharing and linearity analysis. *Theory and Practice of Logic Programming* 14, 3, 379–400.

Arias, J., Carro, M., Salazar, E., Marple, K. and Gupta, G. 2018. Constraint answer set programming without grounding. *Theory and Practice of Logic Programming* 18, 3-4, 337–354.

Bagnara, R., Hill, P. M. and Zaffanella, E. (1997) Set-sharing is redundant for pair-sharing. In *Static Analysis Symposium*. Springer-Verlag, 53–67.

Bravenboer, M. and Smaragdakis, Y. 2009. Strictly declarative specification of sophisticated points-to analyses. *SIGPLAN Not* 44, 10, 243–262.

Briggs, P. and Cooper, K. D. 1994. Effective partial redundancy elimination. In *ACM-SIGPLAN Symposium on Programming Language Design and Implementation*.

Bruynooghe, M. and Codish, M. 1993. Freeness, sharing, linearity and correctness–all at once. In *Proc. Third International Workshop on Static Analysis*. Springer, vol. 724, 153–164, LNCS.

Bruynooghe, M., Codish, M. and Mulkers, A. (1994) Abstract unification for a composite domain deriving sharing and freeness properties of program variables. In *Verification and Analysis of Logic Languages*, F. de Boer and M. Gabbrielli, Eds., 213–230.

Bueno, F. and García de la Banda, M. 2004. Set-sharing is not always redundant for pair-sharing. In *7th International Symposium on Functional and Logic Programming (FLOPS 2004)*, Heidelberg, Germany. Springer-Verlag, vol. 2998, LNCS.

Bueno, F., García de la Banda, M. and Hermenegildo, M. V. 1999. Effectiveness of abstract interpretation in automatic parallelization: A case study in logic programming. *ACM TOPLAS* 21, 2, 189–238.

Cabeza, D. and Hermenegildo, M. 1994. Extracting non-strict independent and-parallelism using sharing and freeness information. In *1994 International Static Analysis Symposium*, Namur, Belgium. Springer-Verlag, vol. 864, 297–313, LNCS.

Carlsson, M. and Mildner, P. 2012. SICStus prolog – the first 25 years. *Theory and Practice of Logic Programming* 12, 1-2, 35–66.

Codish, M., Dams, D., Filé, G. and Bruynooghe, M. 1996. On the design of a correct freeness analysis for logic programs. *The Journal of Logic Programming* 28, 3, 181–206.

Cousot, P. and Cousot, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of POPL'77*. ACM Press, 238–252.

https://doi.org/10.1017/S1471068424000358 Published online by Cambridge University Press

De Angelis, E., Fioravanti, F., Gallagher, J. P., Hermenegildo, M. V., Pettorossi, A. and Proietti, M. 2021. Analysis and Transformation of Constrained Horn Clauses for Program Verification. TPLP.

Fecht, C. 1996. An efficient and precise sharing domain for logic programs. In PLILP 1996, H. Kuchen and S. D. Swierstra, Eds., vol. 1140. Springer, 469–470, Lecture Notes in Computer Science.

Filé, G. 1994. Share x Free: Simple and correct. Technical Report 15, Dipartamento di Matematica, Universita di Padova.

García de la Banda, M., Hermenegildo, M. V., Bruynooghe, M., Dumortier, V., Janssens, G. and Simoens, W. 1996. Global analysis of constraint logic programs. *ACM Trans. on Programming Languages and Systems* 18, 5, 564–615.

García de la Banda, M., Hermenegildo, M. V. and Marriott, K. 2000. Independence in CLP languages. *ACM Transactions on Programming Languages and Systems* 22, 2, 269–339.

Giacobazzi, R. and Ranzato, F. 2022. History of abstract interpretation. *IEEE Annals of The History of Computing* 44, 2, 33–43.

Guarnieri, M., Köpf, B., Morales, J. F., Reineke, J. and Sánchez, A. 2020. Spectector: Principled detection of speculative information flows. In *2020 IEEE Symposium on Security and Privacy (SP)*, 1–19

Henriksen, K. S. and Gallagher, J. P. 2006. Abstract interpretation of PIC programs through logic programming. In *SCAM'06*. IEEE Computer Society, 184–196.

Hermenegildo, M. and Rossi, F. 1995. Strict and non-strict independent and-parallelism in logic programs: Correctness, efficiency, and compile-time conditions. *Journal of Logic Programming* 22, 1, 1–45.

Hermenegildo, M. V. 2000. A documentation generator for (C)LP systems. In *Int'l. Conf. CL*. Springer-Verlag, vol. 1861, 1345–1361.

Hermenegildo, M. V., Bueno, F., Carro, M., Lopez-Garcia, P., Mera, E., Morales, J. and Puebla, G. 2012. An overview of ciao and its design philosophy. *Theory and Practice of Logic Programming* 12, 1-2, 219–252.

Hermenegildo, M. V. and Morales, J. 2011. The LPdoc Documentation Generator. Ref. Manual (v3.0). Technical report, UPM. URL: http://ciao-lang.org.

Hermenegildo, M. V., Puebla, G., Bueno, F. and Lopez-Garcia, P. 2003. Program development using abstract interpretation (and the Ciao system preprocessor). In *10th International Static Analysis Symposium (SAS'03)*. Springer-Verlag, vol. 2694, 127–152.

Hill, P. M., Zaffanella, E. and Bagnara, R. 2004. A correct, precise and efficient integration of set-sharing, freeness and linearity for the analysis of finite and rational tree languages. *Theory and Practice of Logic Programming* 4, 3, 289–323.

Holzbaur, C. 1995. OFAI CLP(Q,R) Manual, Edition 1.3.3. Technical Report TR-95-09, Austrian Research Institute for Artificial Intelligence, Vienna.

Jacobs, D. and Langen, A. 1989. Accurate and efficient approximation of variable aliasing in logic programs. In *North American Conference on Logic Programming*.

Kelly, A., Marriott, K., Søndergaard, H. and Stuckey, P. 1998. A practical object-oriented analysis engine for CLP. *Software: Practice and Experience* 28, 2, 188–224.

King, A. and Soper, P. 1994. Depth-k sharing and freeness. In *International Conference on Logic Programming*. MIT Press.

Landi, W. and Ryder, B. G. 1992. A safe approximate algorithm for interprocedural pointer aliasing (with retrospective). In *Best of PLDI*, K. S. McKinley, Ed. ACM, 473–489.

Le Charlier, B. and Van Hentenryck, P. 1994. Experimental evaluation of a generic abstract interpretation algorithm for prolog. *ACM TOPLAS* 16, 1, 35–101.

https://doi.org/10.1017/S1471068424000358 Published online by Cambridge University Press

Méndez-Lojo, M. and Hermenegildo, M. 2008. Precise set sharing analysis for Java-style programs. In LNCS.*9th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'08)*. Springer-Verlag, vol. 4905, 172–187,

Méndez-Lojo, M., Navas, J. and Hermenegildo, M. 2007a. A flexible (C)LP-based approach to the analysis of object-oriented programs. In *LOPSTR 2007*, vol. 4915. Springer-Verlag, 154–168, LNCS.

Méndez-Lojo, M., Navas, J. and Hermenegildo, M. V. 2007b. Parametric fixpoint algorithm for analysis of java bytecode. In *ETAPS Workshop On Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'07)*.

Muthukumar, K. and Hermenegildo, M. 1989. Determination of variable dependence information at compile-time through abstract interpretation. In *NACLP'89*. MIT Press, 166–189.

Muthukumar, K. and Hermenegildo, M. 1990. Deriving A Fixpoint Computation Algorithm for Top-down Abstract Interpretation of Logic Programs. Technical Report ACT-DC-153-90, Microelectronics and Comp. Tech. Corp. (MCC).

Muthukumar, K. and Hermenegildo, M. 1991. Combined determination of sharing and freeness of program variables through abstract interpretation. In *8th Int'l. Conference on Logic Programming*. MIT Press, 49–63.

Muthukumar, K. and Hermenegildo, M. 1992. Compile-time derivation of variable dependency using abstract interpretation. *JLP* 13, 2/3, 315–347.

Navas, J., Bueno, F. and Hermenegildo, M. V. 2006. Efficient top-down set-sharing analysis using cliques. In LNCS.*8th Int'l. Symp. on Practical Aspects of Declarative Languages (PADL'06)* . Springer, vol. 2819, 183–198,

Navas, J., Méndez-Lojo, M. and Hermenegildo, M. V. 2009. User-definable resource usage bounds analysis for Java bytecode. In*Proceedings of the Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'09) 2009*. Elsevier, North Holland, vol. 253, 65–82, Electronic Notes in Theoretical Computer Science.

Pontelli, E., Gupta, G., Pulvirenti, F. and Ferro, A. 1997. Automatic compile-time parallelization of prolog programs for dependent and-parallelism. In Naish, L., Ed. *Proc. of the Fourteenth International Conference on Logic Programming*. MIT Press, 108–122.

Rountev, A., Milanova, A. and Ryder, B. G. 2001. Points-to analysis for Java using annotated constraints. In *Conference on Object-Oriented*, 43–55.

Santos Costa, V., Rocha, R. and Damas, L. 2012. The YAP prolog system. *Theory and Practice of Logic Programming*, 1-2, 5–34.

Seidl, H. and Vogler, R. 2021. Three improvements to the top-down solver. *Mathematical Structures in Computer Science* 31, 9, 1090–1134.

Søndergaard, H. 1986. An application of abstract interpretation of logic programs: Occur check reduction. In LNCS, *European Symposium on Programming*. Springer-Verlag, vol. 123, 327–338.

Steensgaard, B. 1996. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages*, 32–41.

Swift, T. and Warren, D. 2012. XSB: Extending prolog with tabled logic programming. *TPLP* 12, 1-2, 157–187.

Tilscher, S., Stade, Y., Schwarz, M., Vogler, R. and Seidl, H. 2023. The top-down solver–An exercise in A2I. In *Challenges of Software Verification, chapter 9*, V. Arceri, A. Cortesi, P. Ferrara and M. Olliaro, Eds., vol. ISRL 238, Singapore, Springer, 157–179.

Trias, E., Navas, J., Ackley, E. S., Forrest, S. and Hermenegildo, M. V. 2008. Negative ternary set-sharing. In LNCS, *International Conference on Logic Programming, ICLP 2008*, Udine (Italy). Springer-Verlag, vol. 5366, 301–316,

https://doi.org/10.1017/S1471068424000358 Published online by Cambridge University Press

Van Roy, P. and Despain, A. M. 1990. The benefits of global dataflow analysis for an optimizing prolog compiler. In *North American Conf. on Logic Programming*. MIT Press, 501–515.

Warren, D. H. D. 1987. The SRI model for OR-parallel execution of prolog—Abstract design and implementation. In *Symp. on Logic Prog.*, 92–102.

Warren, R., Hermenegildo, M. and Debray, S. K. 1988. On the practicality of global flow analysis of logic programs. In, *JICSLP 1988*. MIT Press, 684–699.

Whaley, J. and Lam, M. S. 2002. An efficient inclusion-based points-to analysis for strictly-typed languages. In Lecture Notes in Computer Science. *SAS 2002*, vol. 2477, 180–195,

Wielemaker, J., Schrijvers, T., Triska, M. and Lager, T. 2012. SWI-prolog. *TPLP* 12, 1-2, 67–96.

Zaffanella, E., Bagnara, R. and Hill, P. M. 1999. Widening sharing. In LNCS. *PPDP 1999*. Berlin, vol. 1702, Springer-Verlag, 414–432,

Zanardini, D. 2018. Field-sensitive sharing. *Journal of Logical and Algebraic Methods in Programming* 95, 103–127.

https://doi.org/10.1017/S1471068424000358 Published online by Cambridge University Press