TPLP **26** (1-2): 26–45, 2026. © The Author(s), 2026. 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/S1471068426100532 First published online 13 July 2026

*Experimental Evaluation of Optimal Abstract Operators for Sharing and Linearity Analysis**

GIANLUCA AMATO and FRANCESCA SCOZZARI

Laboratory of Computational Logic and Artificial Intelligence, "G. d'Annunzio" University of Chieti-Pescara, Italy,

(*e-mails:* gianluca.amato@unich.it, francesca.scozzari@unich.it)

submitted 27 May 2026 UTC; revised 27 May 2026 UTC; accepted 28 May 2026 UTC

Abstract

In the field of static analysis of logic programs, the optimality of abstract operators is a valuable theoretical property, as it provides insight into the structure of abstract domains and the maximum precision that can be achieved. However, implementing optimal operators is often complex and may significantly impact performance, giving rise to a trade-off between precision and efficiency. We experimentally investigate this trade-off in the context of sharing and linearity analysis of logic programs. Our experiments build on previous work that proposed several optimal operators for unification and matching. We have implemented these abstract operators and the corresponding abstract domains within the PLAI analyzer, part of the CiaoPP preprocessor, and we report the impact of increasing operator precision on the accuracy and performance of the overall analysis.

KEYWORDS: abstract interpretation, logic programming, sharing, linearity

1 Introduction

This paper is concerned with the static analysis of logic programs, in particular with *(set) sharing* analysis, whose aim is to identify sets of variables that share a common variable in a substitution. For example, consider the substitution $\{x/f(u,v), y/g(u,u,u), z/v\}$. We say that x and y share the variable u, while x and z share the variable v, and no single variable is shared by x, y, and z. Sharing has been extensively studied in many papers, such as those by Hans and Winkler (1992); Jacobs and Langen (1992); Muthukumar and Hermenegildo (1992); Codish *et al.* (1999); Bagnara *et al.* (2002); Jurjio-Rivas *et al.* (2024). It is well known that the precision of sharing analysis may be increased if we also record information about the linearity of terms. We say that a term is *linear* if it does not contain multiple occurrences of the same variable. For example, the term $f(x, f(y, z))$ is linear, while $f(x, f(y, x))$ is not, since x occurs twice. Integration of linearity and sharing information has been proposed by many authors, such as Codish *et al.* (1991); Hans and

* We thank Manuel Hermenegildo and José F. Morales for valuable discussions on the inner workings of CiaoPP.

https://doi.org/10.1017/S1471068426100532 Published online by Cambridge University Press

Winkler (1992); Muthukumar and Hermenegildo (1992); King (1994), and recently also in object-oriented languages by Amato *et al.* (2022). The works of Codish *et al.* (1995) and Bagnara *et al.* (2005) contain extensive comparative evaluations of static analyses of logic programs combining many different abstract properties.

In this paper, we focus on the following abstract domains:

- SHARING, the basic domain for (set) sharing analysis, including the improvements presented by Amato and Scozzari (2009).
- SHLIN, a simple domain combining sharing and linearity information (more precisely, it is the reduced product SHARING × LIN), including the optimal operators presented by Amato and Scozzari (2010, 2014, 2024).
- SHLIN^2, originally introduced by King (1994) and later refined by Amato and Scozzari (2010, 2024).

For each domain, we implemented both the standard (as described in the literature) and the optimal abstract operators. An operator is optimal if it is the most precise correct abstraction of its concrete counterpart. While optimality is well understood theoretically, its practical impact is less clear. Intuitively, increasing precision comes at a computational cost, since it often requires more complex operations. Therefore, one may expect a trade-off between precision and performance when adopting optimal operators. However, in some cases, higher precision may lead to smaller abstract objects, which can reduce the overall cost of the analysis.

All the domains have been implemented in the PLAI static analyzer (Muthukumar and Hermenegildo 1992), which is part of CiaoPP, the pre-processor (Bueno *et al.* 2025) of the Ciao Prolog system (Bueno *et al.* 1997). PLAI performs a goal-dependent analysis, where the program is analyzed with respect to a given goal. Goal-dependent analysis requires the implementation of two main operations called *forward* and *backward* unification[1] (see the papers by Cortesi *et al.* (1996) and Amato and Scozzari (2009)). Forward unification performs parameter passing by unifying the atom chosen by the selection rule and the *call substitution* with the head of the chosen clause. The result is an initial substitution for the clause, called *entry substitution*. Backward unification propagates back to the initial goal the *exit substitution* (i.e., the result of the analysis of the clause body), thus obtaining the *answer substitution*[2] for the initial goal. Forward and backward unification may be implemented using a common unification operator (*mgu*) together with other straightforward operations like projection and renaming of variables. Amato and Scozzari (2009) show that precision of backward unification may be improved by making use of an additional matching (*match*) operator.

The aim of this paper is: (1) to analyze the trade-off between standard and optimal operators for *mgu* and *match*; (2) to explore alternative implementations of backward unification, based either on *mgu* alone or on a combination of *mgu* and *match*; and (3) to experimentally evaluate the SHLIN^2 domain, which has not been implemented before.

The rest of the paper is structured as follows. Section 2 introduces the abstract domains considered in this work. Section 3 describes their implementation within the

[1] Bruynooghe (1991) and Hans and Winkler (1992) use *procedure entry* and *procedure exit*. Muthukumar and Hermenegildo (1991) use `call_to_entry` and `exit_to_success`.
[2] Also called *success substitution*.

https://doi.org/10.1017/S1471068426100532 Published online by Cambridge University Press

PLAI analyzer. Section 4 presents the experimental evaluation and discusses the impact of different operators on precision and performance. Finally, Section 5 concludes with directions for future work.

2 Preliminaries: Abstract domains for sharing analysis

In this section, we provide a formal yet intuitive presentation of the abstract domains involved in the experimental evaluation. Given two sets C and A of concrete objects and abstract properties, called the *concrete* and *abstract* domain, respectively, an *abstract interpretation* (Cousot and Cousot 1992) is given by an approximation relation $\rhd \subseteq A \times C$. When $a \rhd c$ holds, we say that a is a *correct abstraction* of c. We work in a framework where A is endowed with a partial order relation to compare properties by *precision*: $a_1 < a_2$ means that a_1 is more precise than a_2. The precision and approximation relations are related by the following: (1) $a \rhd c$ and $a \leq a'$ imply $a' \rhd c$ and (2) each c has a least (most precise) correct abstraction in A. Following the collecting semantics approach (Amato *et al.* 2020), abstract operators are defined as correct approximations of their concrete counterparts. Given a function $f : C \to C$, we say that $\tilde{f} : A \to A$ is a correct abstraction of f, and we write $\tilde{f} \rhd f$, when $a \rhd c \Rightarrow \tilde{f}(a) \rhd f(c)$. We say that $\tilde{f} : A \to A$ is the *optimal* abstraction of f when it is correct and, for each $f' : A \to A$, $f' \rhd f \Rightarrow \tilde{f} \leq f'$ with the pointwise ordering. In other words, the optimal abstraction of a concrete operator f is its most precise correct approximation.

2.1 The Sharing abstract domain

The pioneering abstract domain Sharing by Jacobs and Langen (1992) represents sharing information in a concise set representation. Let $\mathcal{V}$ be an infinite set of variables and $\mathcal{P}_f(\mathcal{V})$ be the set of its finite subsets. The domain Sharing can be defined as follows:

$$\text{Sharing} = \{[A, U] \mid A \subseteq \mathcal{P}(U), (A \neq \emptyset \Rightarrow \emptyset \in A), U \in \mathcal{P}_f(\mathcal{V})\}.$$

Intuitively, an abstract object $[A, U]$ describes the relationships among the variables in U: if $S \in A$, the variables in S are allowed to share a common variable. For example, $[\{\{x, y\}, \{z\}, \emptyset\}, \{w, x, y, z\}]$ represents the substitutions where x and y may possibly share, while z is independent from both x and y. Moreover, the variable w, which does not appear in the first component, is ground. Two examples of such substitutions are $\{x/y, w/a\}$ and $\{w/a\}$ while $\{x/z, w/a\}$ is not, because x and z do share. In addition, a special value $\bot$ is needed to denote the empty set of substitutions.

2.2 The shLin abstract domain

The abstract domain ShLin keeps track of the amount of linearity information in a substitution, coupling an object of Sharing with the set of linear variables. More formally, it is defined as the reduced product of Sharing and a basic linearity domain Lin. Each element of ShLin is a triple where the first component represents the sets of variables which may share, exactly as in Sharing; the second component records the set of variables that must be linear; the third component is, as before, the set of variables of interest. The abstract domain ShLin is defined as follows, where $vars(S)$ denotes the

https://doi.org/10.1017/S1471068426100532 Published online by Cambridge University Press

set of all the variables in S:

$$\textsc{ShLin} = \{[S, L, U] \mid S \subseteq \mathcal{P}(U), (S \neq \emptyset \Rightarrow \emptyset \in S), L \supseteq U \setminus vars(S), U \in \mathcal{P}_f(\mathcal{V})\} \ .$$

For instance, $[\{\{x, y\}, \emptyset\}, \{x, z\}, \{x, y, z\}]$ represents the substitutions where x may share with y, while x and z are (always) bound to a linear term (actually z must be ground). Two examples of such substitutions are $\{x/t(u, a), y/t(u, u), z/a\}$ and $\{x/y, z/a\}$, while $\{x/t(u, u), y/t(u, u), z/a\}$ is not since $t(u, u)$ is not linear.

2.3 Intuition on linearity and its impact on sharing precision

Linearity information plays a key role in improving the precision of sharing analysis. In fact, when a variable occurs multiple times in a term, unification may introduce additional sharing between variables.

For instance, consider the substitution $\theta_1 = \{x/f(u, v)\}$ over the set $U = \{x\}$ of variables of interest. This substitution is abstracted in Sharing as $S = [\{\{x\}, \emptyset\}, U]$. If we now compute the *mgu* between S and $\mathcal{E} = \{x = f(y, z)\}$, we obtain the abstract object $mgu(S, \mathcal{E}) = [\{\{x, y\}, \{x, z\}, \{x, y, z\}, \emptyset\}, \{x, y, z\}]$, which contains the sharing group $\{x, y, z\}$, due to the fact that the substitution $\theta_2 = \{x/f(u, u)\}$ is also abstracted as S in Sharing.

Linearity information mitigates this problem. Intuitively, linearity acts as a constraint that rules out combinations of sharing groups that would only arise due to repeated occurrences of variables. In fact, the substitutions θ_1 and θ_2 are abstracted in different objects in ShLin, namely $S_1 = [\{\{x\}, \emptyset\}, \{x\}, U]$ and $S_2 = [\{\{x\}, \emptyset\}, \emptyset, U]$. The abstract unification operator in ShLin can now exploit this information and obtain $mgu(S_1, \mathcal{E}) = [\{\{x, y\}, \{x, z\}, \emptyset\}, \{x, y, z\}, \{x, y, z\}]$ while $mgu(S_2, \mathcal{E}) = [\{\{x, y\}, \{x, z\}, \{x, y, z\}, \emptyset\}, \emptyset, \{x, y, z\}]$ which still contains the sharing group $\{x, y, z\}$.

2.4 The shLin^2 *abstract domain*

The idea behind ShLin^2 is to enhance Sharing by annotating each sharing group with linearity information on each variable. If a variable in a sharing group is (possibly) non-linear, it is denoted using ω as an exponent, otherwise the variable is linear. For instance, the object $xy^\omega z$ (called a 2-sharing group) represents the sharing group $\{x, y, z\}$ and the information that the common variable shared among x, y and z may occur more than once in y but only once in x and z. For example, this is the case for the variable u in the substitution $\{x/f(u, v, v), y/g(u, u, w), z/u\}$. The ShLin^2 abstract domain[3] can be defined as follows:

$$\textsc{ShLin}^2 = \{[S, U] \mid S \text{ is a set of 2-sharing groups}, U \in \mathcal{P}_f(\mathcal{V}), S \neq \emptyset \Rightarrow \emptyset \in S\}.$$

Linearity information may vary between 2-sharing groups. For instance, $[\{xy, y^\omega z, \emptyset\}, \{x, y, z\}]$ represents the substitutions θ where the variable y may share with x or z, x and z are linear, while y might be non-linear. However, the variable

[3] Usually the definition also requires that the set S is downward closed w.r.t. linearity information: for instance, the set $\{xy^\omega z\}$ is not downward closed, while $\{xyz, xy^\omega z\}$ is downward closed. For details, refer to the work by Amato and Scozzari (2010).

https://doi.org/10.1017/S1471068426100532 Published online by Cambridge University Press

in common between $\theta(x)$ and $\theta(y)$ may only appear once in $\theta(y)$. An example of such a substitution is $\{x/u, y/t(u,v,v), z/v\}$, while $\{x/u, y/t(u,u,v), z/v\}$ is not.

2.5 Optimal abstract operators

All the above abstract domains are equipped with the optimal abstract operators for unification, matching and the other required operations. Amato and Scozzari (2009, 2024) implement backward unification through matching, exploiting the property that the exit substitution is always more instantiated than the call substitution. Analyses based on matching have been shown to be strictly more precise than analyses that do not use matching. Moreover, Amato and Scozzari (2024) show that when using the optimal operators, the overall analysis is strictly more precise than the analysis performed with the standard operators. Surprisingly, the optimal operators are able to improve not only sharing and linearity information but also groundness.

3 Implementation

We have implemented the abstract domains Sharing, ShLin, ShLin^2, together with their operators, within the PLAI analyzer. All our code is publicly available on GitHub under the LGPL-3.0 license at https://github.com/CLAI-UdA/ciaopp.

3.1 Structure of the code

PLAI is written in Prolog and completely modular: each abstract domain is implemented in its own module and communicates with the analyzer engine through a well-defined interface. However, this interface is quite extensive and, to reduce the amount of code required for each abstract domain, we implemented a *compatibility layer* in the module `as_template` which translates the standard interface into more elementary operations.

For example, consider the `call_to_entry` predicate, part of the abstract domain interface, corresponding to what we have called forward unification. It is defined as follows:

```
:- pred call_to_entry(+Sv, +Sg, +Hv, +Head, +ClauseKey, +Fv, +Proj,
                      -Entry, -ExtraInfo)
```

where

- `Sg` is the goal we are going to analyze;
- `Sv` is the set of variables occurring in `Sg`;
- `ClauseKey` is an identifier for the clause we are using to analyze `Sg`;
- `Head` is the head of the clause;
- `Hv` is the set of variables occurring in `Head`;
- `Fv` are the variables in the clause that do not occur in `Head`;
- `Proj` is the call substitution projected onto the variables in `Sv`;
- `Entry` is the resulting entry substitution;
- `ExtraInfo` may contain additional information that is passed later to the backward unification phase.

https://doi.org/10.1017/S1471068426100532 Published online by Cambridge University Press

Our implementation of this predicate is the following:

```
call_to_entry(_Sv, Sg, Hv, Head, _ClauseKey, Fv, Proj, Entry, (Unif, Entry0)) :-
   unifiable_with_occurs_check(Sg, Head, Unifier),
   augment(Proj, Hv, ASub),
   mgu(ASub, Hv, Unifier, Entry0),
   project(Entry0, Hv, Entry1),
   augment(Entry1, Fv, Entry).
```

It combines the predicates `augment`, `mgu` and `project`, implemented in each of our domains, with `unifiable_with_occurs_check` which is a built-in predicate in Ciao.

The compatibility layer also avoids duplicating code that is syntactically identical across different domains. For example, the special value `$bottom`, which denotes an empty abstract substitution (corresponding to $\bot$ in Section 2.1), is handled largely within this layer. The same applies to most of the code for analyzing Prolog built-ins. Apart from the compatibility layer, there are two other modules that contain code shared by all our domains: `as_aux` contains generic utility code, while `as_bags` contains an implementation of finite multisets.

Finally, the three domains are implemented in the modules `as_sharing`, `as_shlin` and `as_shlin2`. Although CiaoPP already provides several implementations of SHARING, we have devised a new one to serve as a baseline for the other domains in our collection. The behavior of the analysis (such as choosing between matching or unification for backward unification, or between optimal and standard operators) can be configured through specific CiaoPP flags.

The built-in PLAI domains for sharing analysis are implemented natively in PROLOG, using sets (ordered lists) and a shallow encoding of logic variables. We adopt the same approach in our implementations. All code is fully annotated with *type* and *mode* information using the annotation mechanism provided by the Ciao system. For example, this is the complete specification of the predicate computing the lubs of abstract substitutions:

```
:- pred compute_lub(+ListASub, -LubASub)
   : list_nonempty(asub) * ivar => asub(LubASub)
   + (not_fails, is_det).
```

The specification states that `compute_lub` is always called with a non-empty list of abstract substitutions as its first argument, and an independent variable as a second argument, which is bound to an abstract substitution on exit. The predicate is also declared to be deterministic and not to fail (when called under these preconditions). These annotations may be checked at run-time and have proved very useful during development, both for catching bugs in our code and for ensuring a correct understanding of how the CiaoPP interface is expected to work. In addition, most modules include a comprehensive test suite.

3.2 From the algorithm to the implementation

Moving from the algorithmic descriptions given in our papers to their actual implementation has not always been straightforward, and has required careful consideration.

https://doi.org/10.1017/S1471068426100532 Published online by Cambridge University Press

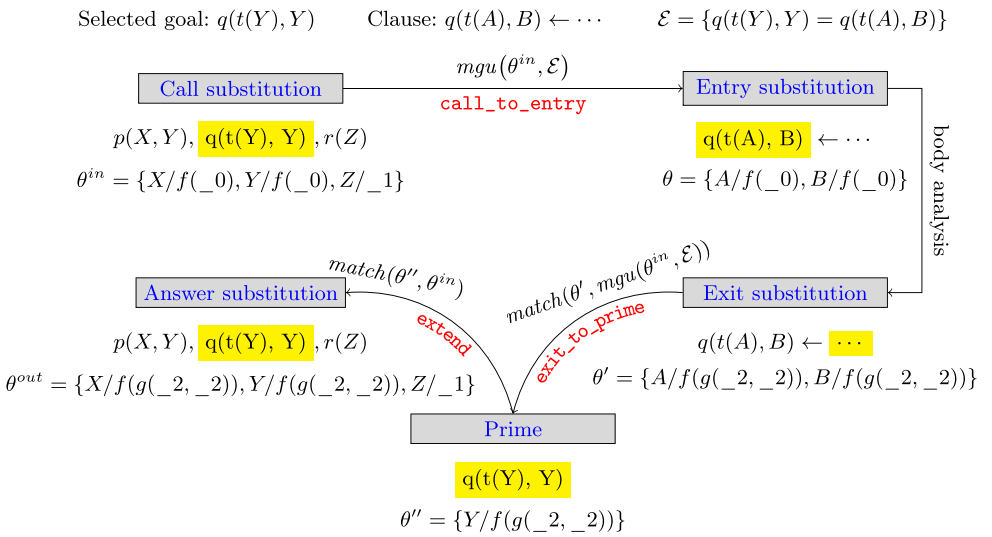


Fig. 1. Flow of the analysis in PLAI.

First of all, while PLAI has the predicate `call_to_entry` which corresponds directly to forward unification, it splits backward unification into two distinct steps. First, the exit substitution is projected onto the variables of the chosen atom by the predicate `exit_to_prime`, it is then extended to all the variables of the original goal by the predicate `extend`, incorporating the information on the other variables coming from the call substitution. This process is depicted in Figure 1, which we briefly describe here. The procedure `call_to_entry` computes the *mgu* between θ^{in} and $\mathcal{E}$, the equation connecting the selected atom $q(t(Y), Y)$ and the clause head $q(t(A), B)$, yielding $\bar{\theta} = mgu(\theta^{in}, \mathcal{E}) = \{X/f(_0), Y/f(_0), Z/_1, A/f(_0), B/f(_0)\}$. This is then projected onto the variables of the clause to obtain the entry substitution θ. After analyzing the body, θ is further instantiated, yielding the exit substitution θ'.

Now `exit_to_prime` computes the *mgu* between θ' and $\bar{\theta}$, and projects the result onto the variables of the selected goal, obtaining θ'', called the *prime*. Note that in this step we are sure that variables in θ' are not further instantiated, hence unification may be replaced by matching. Using matching instead of unification (which is irrelevant in the concrete) improves precision in the abstract. Finally, `extend` propagates the effect of the prime to the entire initial goal, by unifying it with the call substitution. Again, since variables in the prime are not going to be further instantiated, unification may be replaced by matching.

Two observations are worth noting. First, the `extend` step is conceptually redundant: the final projection in `exit_to_prime` could be performed directly over all variables of the initial goal, thus producing the answer substitution in a single step. However, this would require modifying the PLAI engine, which we do not intend to do. Second, our *mgu* and *match* operations are designed so that *mgu* takes an abstract substitution and a concrete set of equations, whereas *match* takes two abstract substitutions. This makes the implementation of `extend` using only *mgu* a bit cumbersome, since it should operate on two abstract substitutions.

https://doi.org/10.1017/S1471068426100532 Published online by Cambridge University Press

$$\mathsf{match}_{sl}([S_1, L_1, U_1], [S_2, L_2, U_2]) = \big[s(S_0' \cup S_0''), l(S_0' \cup S_0''), U\big]$$

$$U = U_1 \cup U_2 \qquad nl(X) = \{v \in \mathcal{V} \mid \exists B_1, B_2 \in X, B_1 \neq B_2, v \in B_1 \cap B_2\}$$

$$S_2' = \{B \in S_2 \mid B \cap U_1 = \emptyset\} \qquad s(H) = \{B \mid \langle B, L\rangle \in H\}$$

$$S_2'' = S_2 \setminus S_2' \qquad l(H) = \bigcap \{L_1 \cup L \cup (U \setminus B) \mid \langle B, L\rangle \in H\}$$

$$\bar{S} = \{B \in S_2'' \mid B \cap L_1 = \emptyset\} \qquad S_0' = \{\langle B, L_2\rangle \mid B \in S_2'\}$$

$$S_0'' = \Big\{ \Big\langle B \cup \bigcup X, L_2 \setminus nl(X) \setminus \bigcup (X \cap \bar{S}) \Big\rangle \;\Big|\; B \in S_1,$$

$$X \subseteq S_2'', B \cap U_2 = \Big(\bigcup X\Big) \cap U_1, L_1 \cap nl(X) = \emptyset \Big\}$$

Fig. 2. The definition of the match operator in ShLin, for $[S_1, L_1, U_1]$ and $[S_2, L_2, U_2] \in$ ShLin.

More importantly, the definitions of *mgu* and *match* given in our papers are expressed in a declarative style. It is relatively easy to implement these operators following the definitions using a generate-and-test approach, and we have adopted this strategy in several cases. However, there are some cases in which this approach is too slow, making it necessary to devise more carefully optimized implementations.

As an illustrative example, consider the definition of the matching operator for ShLin shown in Figure 2. Rather than discussing the details of the definition (for which we refer to Amato and Scozzari (2010)), we use it here to highlight some of the issues that arise in its implementation. The crucial step in the definition is the computation of S_0''. This involves selecting a sharing group $B \in S_1$ and a subset of sharing groups $X \subseteq S_2''$, and then checking whether the conditions $B \cap U_2 = (\bigcup X) \cap U_1$ and $L_1 \cap nl(X) = \emptyset$ are satisfied.

A straightforward implementation would follow a generate-and-test strategy that mirrors the formal definition. However, this approach requires enumerating a large number of candidate subsets X, whose number is exponential in the size of S_2'', which is itself potentially exponential in the number of program variables. To obtain a practically usable implementation, it is therefore necessary to reduce the search space.

First, once we fix $B \in S_1$, a necessary condition for satisfying $B \cap U_2 = (\bigcup X) \cap U_1$ is that every $C \in X$ satisfies $C \cap U_1 \subseteq B \cap U_2$. Hence, we can restrict the search to subsets $X \subseteq \{C \in S_2'' \mid C \cap U_1 \subseteq B \cap U_2\}$. Additionally, instead of generating X, checking constraints and then producing the pair $\big\langle B \cup \bigcup X, L_2 \setminus nl(X) \setminus \bigcup (X \cap \bar{S})\big\rangle$, we have devised a procedure that incrementally builds the resulting pairs, without explicitly generating the candidates X at all. In this way, although the theoretical complexity of the procedure does not decrease, the implementation is generally much faster.

4 Experimental evaluation

In order to benchmark both the precision and the efficiency of our domains, we used the SWI-Prolog benchmark suite.[4] It consists of 35 Prolog programs, collected by the SWI-Prolog organization but running on many Prolog systems. Two programs were removed from the suite: `det.pl`, which does not run in Ciao, and `queens_clpfd.pl`, which relies on CLP(FD) features that are not supported in our setting. As a result, we analyzed a total of 33 programs. They are listed in Table 1 together with some basic size metrics.

[4] https://github.com/SWI-Prolog/bench, commit f326964

https://doi.org/10.1017/S1471068426100532 Published online by Cambridge University Press

Table 1. *Benchmark programs and corresponding size metrics. The column max vars is the maximum number of variables in a clause*

Program	Size	#Predicates	#Clauses	Tot. Vars	Max Vars
boyer	11232	25	135	312	7
browse	2831	16	32	119	12
chat_parser	25824	158	516	1650	23
crypt	1834	9	27	64	19
derive	975	5	14	40	5
divide10	857	3	12	38	5
eval	639	5	6	11	4
fast_mu	3359	9	18	102	13
fib	1109	2	4	7	6
flatten	5773	27	58	221	13
log10	882	3	12	38	5
meta_qsort	2829	8	26	62	7
moded_path	2429	6	21	26	6
mu	945	9	17	34	6
nand	19927	42	138	714	44
nreverse	616	4	6	10	4
ops8	835	3	12	38	5
perfect	2061	9	14	40	5
pingpong	480	4	7	5	2
poly_10	3233	12	33	106	8
prover	2352	10	33	92	9
qsort	772	4	7	20	7
queens_8	1753	7	12	30	5
query	1754	6	55	11	6
reducer	11743	43	122	459	17
sendmore	1178	4	22	17	11
serialise	1108	8	14	45	7
sieve	1173	6	9	21	5
simple_analyzer	14154	67	143	662	48
tak	608	3	4	15	10
times10	853	3	12	38	5
unify	6945	15	63	291	14
zebra	1608	7	12	128	78

Each program was analyzed in 15 different configurations. These include the built-in SHARING domain provided by PLAI (`share_amgu`), the built-in SHARING × FREE × LIN (`shfrlin_amgu`), as well as our implementations of SHARING, SHLIN and SHLIN^2 with different options. In the following, we report only a representative subset of these configurations. The configuration referred to as *base* uses the standard (as described in the literature) *mgu* for both forward and backward unification. In particular, the standard operators used for SHLIN are the ones by Hans and Winkler (1992), including the independence checks that Howe and King (2003) and Hill *et al.* (2004) showed to be removable. The configuration *match* employs the standard *mgu* for forward unification but the (optimal) matching operator for backward unification. Finally, the configuration *optimal* uses optimal operators for both *mgu* and *match*.

https://doi.org/10.1017/S1471068426100532 Published online by Cambridge University Press

Program	ShLin^2			ShLin			Sharing			built-in	
	optimal	match	base	optimal	match	base	optimal	match	base	Sharing	ShFrLin
`chat_parser`	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
`flatten`	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓
`reducer`	✓	✓	✗	✗	✓	✗	✓	✓	✗	✗	✓
`simple_analyzer`	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
`unify`	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✓
`zebra`	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗

Fig. 3. Critical programs, causing either a timeout or an out-of-memory condition (both ✗).

The built-in domains `share_amgu` and `shfrlin_amgu` rely on specialized implementations that likely do not correspond to any of the configurations considered in this work. To the best of our knowledge, these implementations have not been fully described in the literature, at least in their current form. Note that `shfrlin_amgu` also tracks freeness information. This additional component may both improve precision and introduce extra computational overhead, which should be taken into account when comparing performance.

All benchmarks were executed on a machine equipped with an Intel i5-13600K CPU with 32 GB of RAM, running Ubuntu Linux 25.10 and Ciao `1.25.0-m1`. We measured both execution time and precision, according to the following procedure:

- CiaoPP was instructed to analyze the main predicate of each program;
- other predicates were analyzed only as a consequence of this initial call;
- a timeout of 2 minutes was imposed for each analysis;
- the virtual address space of the analyzer was limited to 24 GB of RAM.

The 24 GB memory limit was introduced to reliably detect out-of-memory conditions. In preliminary experiments, we relied on the operating system's OOM killer; however, this approach proved inconsistent, as it occasionally terminated not only the analyzer but also the script orchestrating the benchmark execution.

Following this procedure, we were able to fully analyze all benchmarks, with the exception of six critical programs, which failed under at least one configuration. These programs are summarized in Figure 3. There are two cases, namely `reducer` with ShLin and `flatten` with ShLin^2, where increasing precision of abstract operators leads to non-termination, and two cases, namely `chat_parser` with ShLin, `reducer` with ShLin^2 and `unify` with both Sharing and ShLin, where increasing precision allows the analysis to complete within the timeout.

As expected, the program metric most strongly correlated with analysis failures is the maximum number of variables in a clause (`max vars`), as it directly affects the size of abstract objects. In fact, the six programs in Figure 3 are all among the top nine programs with the highest values of the `max vars` metric. This observation is confirmed by computing the Pearson correlation coefficient between the indicator function of the timeout/out-of-memory condition and each metric. The highest correlation is consistently

https://doi.org/10.1017/S1471068426100532 Published online by Cambridge University Press

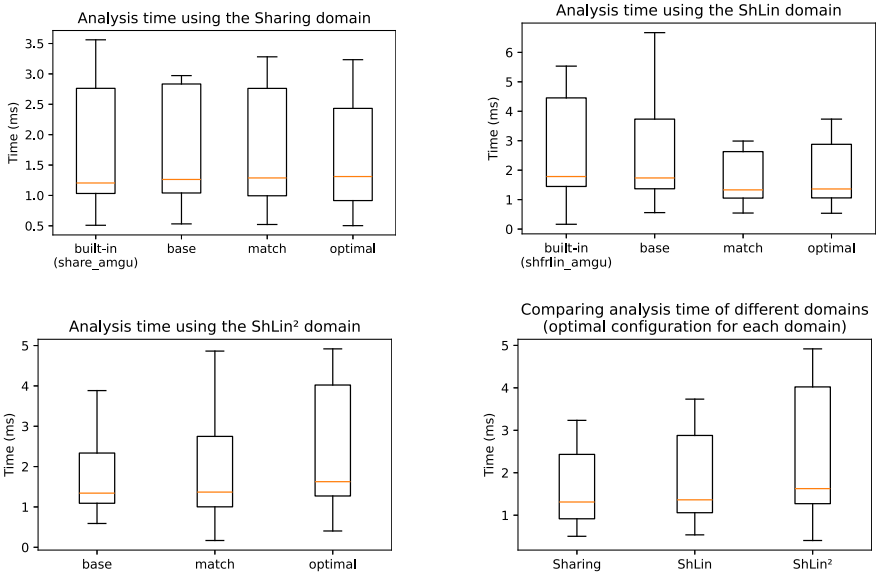


Fig. 4. Analysis time for non-critical programs.

obtained with `max vars` across all abstract domains. For instance, for ShLin with optimal operators, the correlation is 0.72 with `max vars`, while it remains below 0.30 for all other metrics.

Figure 4 reports the execution time for all non-critical programs, while Table 2 provides the complete set of results. For each domain we show a boxplot comparing different configurations, followed by a final boxplot comparing the three domains under optimal operators. As expected, more complex domains lead to higher execution times. However, the increase is negligible when comparing ShLin with Sharing, while it is much more visible for ShLin^2. This behavior is expected, since the size of abstract objects for the latter is much larger. Analyzing the boxplots for each domain confirms what we already observed for the critical programs: while more complex operators may increase analysis time, this effect does not occur systematically, the increase is generally negligible and sometimes the opposite happens.

If we exclude timeout and out-of-memory cases, the correlation between analysis time and the considered metrics becomes less clear. In particular, `max vars` generally shows a low correlation, while `predicates` tends to exhibit higher values. However, these trends vary significantly across the different abstract domains.

Figure 5 reports the precision of the three domains with respect to the sharing property. For each program and configuration, we counted the number of sharing groups in the annotations generated by CiaoPP. For program points where CiaoPP reports multiple analysis results, corresponding to different preconditions, we only counted the sharing groups appearing in the alternative with the smallest number of sharing groups. Although this is a coarse measure of precision, it provides a useful indication of overall trends. It shows that the use of matching and optimal operators leads to a clear increase in precision for the ShLin domain, while the improvement is more limited for Sharing and ShLin^2. Moreover, more expressive domains yield more precise results.

https://doi.org/10.1017/S1471068426100532 Published online by Cambridge University Press

https://doi.org/10.1017/S1471068426100532 Published online by Cambridge University Press

Table 2. *Analysis time for all programs with the different abstract domains and configurations. OOM denotes an out-of-memory condition, while TO denotes a timeout*

Program	as_ shlin2_ opt	as_ shlin2_ opt_mgu	as_ shlin2_ noopt	as_ shlin2_ noopt_ mgu	as_ shlin_ opt_ opt	as_ shlin_ opt	as_ shlin_ noindcheck	as_ shlin_ noopt	as_ shlin_ opt_ mgu	as_ shlin_ noindcheck_ mgu	as_ shlin_ noopt_ mgu	as_ sharing_ opt	as_ sharing_ noopt	as_ sharing_ opt_ mgu	as_ sharing_ noopt_ mgu	share	shfrlin
boyer	37.121	57.096	23.932	20.792	51.960	41.358	33.216	33.122	1242.125	253.599	254.620	28.371	31.557	242.264	248.382	25.141	28.793
browse	280.081	250.710	272.285	10.486	29.836	32.216	31.548	31.894	3369.263	257.422	256.255	29.768	586.998	249.751	259.373	15.459	16.814
chat_parser	OOM	TO	OOM	TO	62322.589	24305.137	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
crypt	3.972	5.354	4.863	3.094	2.849	3.043	2.891	2.989	2.802	3.209	3.311	2.438	3.019	1.819	2.821	2.541	4.522
derive	1.889	2.390	1.368	1.461	1.439	1.629	1.491	1.419	2.316	1.734	2.122	1.341	1.352	1.467	1.429	1.415	1.902
divide10	1.474	1.940	1.027	1.133	1.079	1.139	1.055	1.069	1.859	1.763	1.736	0.989	1.054	1.132	1.093	1.049	1.471
eval	0.776	0.853	0.633	0.689	0.689	0.707	0.676	0.709	0.810	0.694	0.688	0.637	0.673	0.677	0.697	0.674	0.785
fast_mu	4.353	5.777	1.626	2.135	2.697	2.969	2.859	1.909	3.549	3.254	3.316	2.341	2.408	2.602	2.312	2.800	4.580
fib	0.676	0.750	0.573	0.625	0.598	0.573	0.551	0.579	0.858	0.837	0.824	0.519	0.527	0.021	0.560	0.567	0.163
flatten	OOM	305.321	OOM	139.053	55.569	44.193	122.589	124.271	6029.845	1158.178	1164.468	104.683	112.588	990.709	1008.485	148.937	46.461
log10	1.342	1.772	0.986	1.087	1.038	1.069	1.064	1.049	1.845	1.662	1.754	0.976	1.002	1.055	1.084	1.016	1.423
meta_qsort	42.202	44.842	27.717	16.639	8.479	7.645	11.238	11.137	28.358	14.497	14.622	6.808	6.150	8.607	9.199	7.752	9.402
moded_path	1.626	1.957	1.216	1.310	1.264	1.294	1.260	1.217	1.387	1.290	1.326	1.088	1.143	1.184	1.181	1.175	1.777
mu	2.456	3.208	1.650	1.757	1.840	1.900	1.741	1.333	2.055	1.931	1.935	1.634	1.623	1.781	1.789	1.794	2.536
nand	58.464	68.381	22.331	25.946	31.262	32.417	93.743	93.841	88.356	76.390	82.523	85.971	86.652	48.461	48.871	43.730	39.258
nreverse	0.935	1.018	0.685	0.777	0.723	0.732	0.706	0.725	0.794	0.760	0.721	0.666	0.652	0.758	0.747	0.685	0.918
ops8	1.414	1.925	1.067	1.095	1.091	1.156	1.076	1.055	1.849	1.700	1.717	0.984	0.988	1.111	1.097	1.057	1.472
perfect	1.797	2.269	1.467	1.579	1.530	1.524	1.479	1.143	1.664	1.628	1.620	1.355	1.406	1.496	1.464	1.070	1.925
pingpong	0.594	0.627	0.168	0.590	0.537	0.551	0.530	0.545	0.549	0.536	0.559	0.503	0.522	0.521	0.532	0.509	0.554
poly_10	4.918	6.710	3.317	3.884	3.734	3.846	5.293	5.231	5.299	5.587	6.670	3.234	3.281	3.676	2.847	3.561	5.533

https://doi.org/10.1017/S1471068426100532 Published online by Cambridge University Press

Table 2. *Continued*

Program	as_shlin2_opt	as_shlin2_opt_mgu	as_shlin2_noopt	as_shlin2_noopt_mgu	as_shlin_opt_opt	as_shlin_opt	as_shlin_noindcheck	as_shlin_noopt	as_shlin_opt_mgu	as_shlin_noindcheck_mgu	as_shlin_noopt_mgu	as_sharing_opt	as_sharing_noopt	as_sharing_opt_mgu	as_sharing_noopt_mgu	share	shfrlin
prover	4.072	5.606	2.543	2.079	2.908	3.012	3.135	2.272	4.995	4.085	4.152	2.427	2.504	2.977	2.972	2.725	4.387
qsort	1.201	1.671	0.939	1.028	0.986	1.010	1.405	1.435	1.147	1.544	1.603	0.855	0.895	0.995	0.997	0.950	1.358
queens_8	1.502	1.587	1.164	1.251	1.246	1.284	1.271	1.281	1.424	1.406	1.426	1.121	1.157	1.218	1.230	1.205	1.680
query	1.968	2.075	1.372	1.342	1.362	1.395	1.314	1.331	1.735	1.412	1.411	1.329	1.296	1.248	0.988	1.280	1.784
reducer	32.608	TO	16.821	TO	TO	29509.867	70973.742	71091.251	TO	TO	TO	70327.287	71882.298	TO	TO	TO	44293.176
sendmore	0.885	2.041	1.411	1.655	1.482	1.504	1.465	1.436	1.812	1.748	1.766	1.311	1.288	1.437	1.406	1.277	1.813
serialise	3.093	4.728	2.954	2.537	3.664	4.280	16.780	16.348	134.471	43.438	43.098	12.953	13.844	37.129	40.664	13.206	3.471
sieve	1.450	1.694	1.218	1.278	1.274	1.295	1.231	1.253	1.381	1.329	1.325	1.179	1.185	1.241	1.263	1.205	1.550
simple_analyzer	TO	TO	OOM	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
tak	0.404	1.163	0.680	0.754	0.685	0.307	1.002	0.900	1.215	1.143	1.147	0.614	0.621	0.706	0.777	0.618	0.884
times10	1.390	1.856	1.019	1.114	1.108	1.215	1.068	1.048	1.893	1.724	1.724	0.677	1.001	1.100	1.118	1.074	1.478
unify	7536.688	588.872	7474.641	207.981	106.249	129.316	973.752	974.414	TO	TO	TO	949.814	977.158	TO	TO	1862.961	157.956
zebra	TO	TO	OOM	TO	OOM	OOM	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO

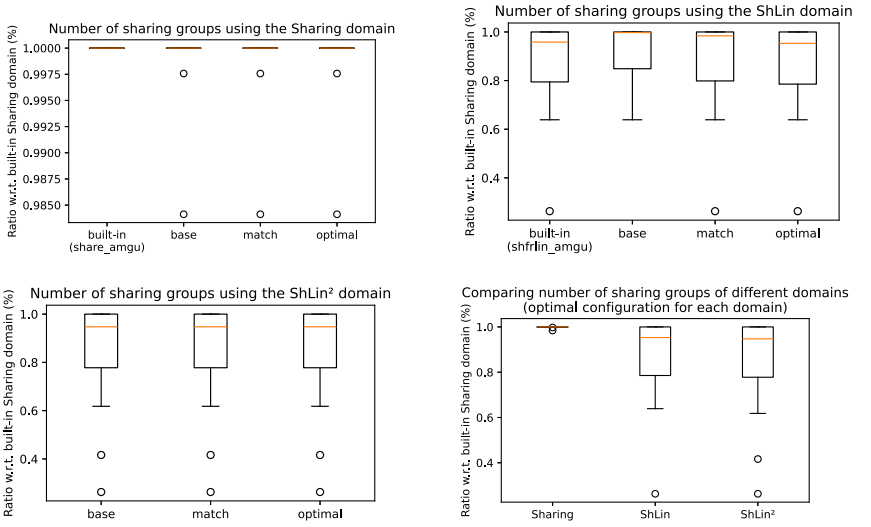


Fig. 5. Precision with respect to the sharing property.

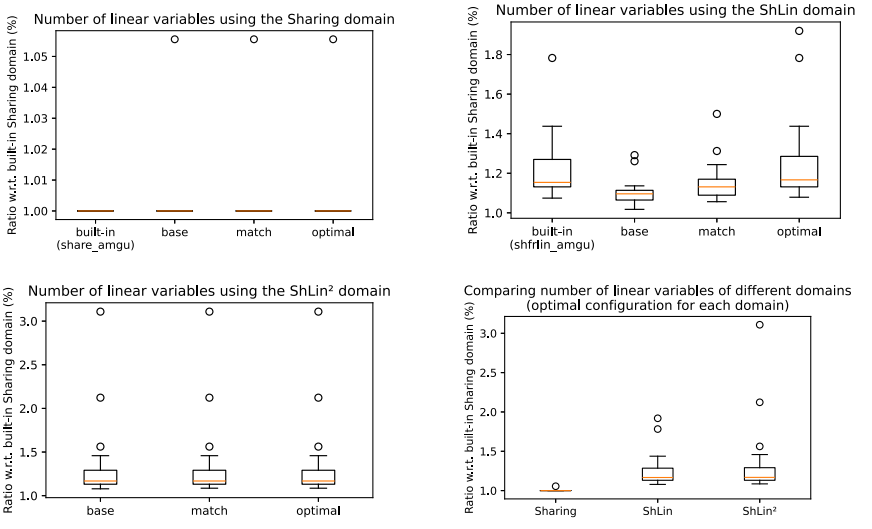


Fig. 6. Precision with respect to the linearity property.

Figure 6 reports the precision with respect to the linearity property, measured by counting the number of linear variables, and computing the maximum in case of multiple annotations. The results are similar to those obtained for the sharing property. We do not report boxplots for groundness, since almost all configurations yield identical results.

The complete set of detailed results is provided in Table 3, which shows that ShLin^2 with matching and optimal operators yields the most precise results among all the considered domains, with the only exception of the programs `chat_parser` and `flatten`, for which the analysis does not terminate. Since we measure sharing precision as the number of sharing groups, this also provides an indication of the size of abstract objects. As expected, in several cases, optimal operators lead to smaller abstract objects. This

https://doi.org/10.1017/S1471068426100532 Published online by Cambridge University Press

https://doi.org/10.1017/S1471068426100532 Published online by Cambridge University Press

Table 3. *Precision results for all analyzed programs and configurations. + indicates the use of optimal mgu, while (+) indicates identical results for standard and optimal mgu (which are therefore collapsed in a single column); M denotes the use of matching instead of mgu; ++ denotes the use of both optimal mgu and optimal matching. OOM denotes an out-of-memory condition, while TO denotes a timeout*

property	program	as_ shlin2 (+)	as_ shlin2_ opt_ mgu	as_ shlin2_ noopt_ mgu	as_ shlin_ opt_ opt	as_ shlin_ opt	as_ shlin_ noopt	as_ shlin_ opt_ mgu	as_ shlin_ noopt_ mgu	as_ sharing (+)	as_ sharing_ mgu (+)	share	shfrlin
mshare	boyer	296	296	296	321	321	437	437	437	479	479	479	437
mshare	browse	765	765	765	804	804	811	817	821	821	821	823	809
mshare	chat_parser	OOM	TO	TO	5804	5804	TO	TO	TO	TO	TO	TO	TO
mshare	crypt	224	224	224	224	224	235	235	235	235	235	235	235
mshare	derive	23	23	23	23	23	23	23	23	29	29	29	23
mshare	divide10	21	21	21	21	21	21	21	21	27	27	27	21
mshare	eval	8	8	8	8	8	8	8	8	8	8	8	8
mshare	fast_mu	97	97	97	97	97	116	116	116	116	116	116	97
mshare	fib	21	21	21	21	21	21	21	21	21	21	21	21
mshare	flatten	OOM	1335	1335	1343	1343	1511	1522	1524	1584	1584	1754	1495
mshare	log10	21	21	21	21	21	21	21	21	27	27	27	21
mshare	meta_qsort	62	62	62	62	62	62	62	62	62	62	63	62
mshare	moded_path	16	16	16	16	16	16	16	16	16	16	16	16
mshare	mu	13	13	13	13	13	13	13	13	16	16	16	13
mshare	nand	757	757	757	757	757	757	2879	2879	2879	2879	2879	757
mshare	nreverse	5	5	5	5	5	5	5	5	5	5	5	5
mshare	ops8	21	21	21	21	21	21	21	21	27	27	27	21
mshare	perfect	46	46	46	46	46	46	46	46	48	48	48	46
mshare	pingpong	9	9	9	9	9	9	9	9	9	9	9	9
mshare	poly_10	54	54	54	54	54	54	54	54	61	61	61	54
mshare	prover	36	36	36	36	36	36	36	36	38	38	38	36
mshare	qsort	14	14	14	14	14	14	14	14	14	14	14	14

https://doi.org/10.1017/S1471068426100532 Published online by Cambridge University Press

Table 3. *Continued*

property	program	as_shlin2 (+)	as_shlin2_opt_mgu	as_shlin2_noopt_mgu	as_shlin_opt_opt	as_shlin_opt	as_shlin_noopt	as_shlin_opt_mgu	as_shlin_noopt_mgu	as_sharing (+)	as_sharing_mgu (+)	share	shfrlin
mshare	queens_8	19	19	19	19	19	19	19	19	19	19	19	19
mshare	query	23	23	23	23	23	23	23	23	36	36	36	23
mshare	reducer	269	TO	TO	TO	15359	16736	TO	TO	16885	TO	TO	16919
mshare	sendmore	127	127	127	127	127	127	127	127	127	127	127	127
mshare	serialise	102	102	102	195	195	197	225	227	245	245	245	195
mshare	sieve	19	19	19	19	19	19	19	19	19	19	19	19
mshare	simple_analyzer	TO/OOM	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
mshare	tak	38	38	38	38	38	38	38	38	38	38	38	38
mshare	times10	21	21	21	21	21	21	21	21	27	27	27	21
mshare	unify	1287	1367	1367	1365	1368	2487	TO	TO	2602	TO	2602	1375
mshare	zebra	TO/OOM	TO	OOM	OOM	OOM	TO	TO	TO	TO	TO	TO	TO
linear	boyer	383	383	383	335	335	152	173	150	118	118	118	173
linear	browse	376	376	376	299	299	259	223	211	177	177	177	289
linear	chat_parser	OOM	TO	TO	4261	4261	TO	TO	TO	TO	TO	TO	TO
linear	crypt	620	620	620	620	620	589	429	429	396	396	396	620
linear	derive	105	105	105	105	105	105	101	101	82	82	82	105
linear	divide10	101	101	101	101	101	101	97	97	80	80	80	101
linear	eval	42	42	42	42	42	42	41	41	34	34	34	42
linear	fast_mu	358	355	355	355	355	275	300	264	246	246	246	353
linear	fib	45	45	45	45	45	39	42	38	24	24	24	45
linear	flatten	OOM	378	378	379	377	247	279	226	143	143	141	400
linear	log10	101	101	101	101	101	101	97	97	80	80	80	101
linear	meta_qsort	134	134	134	105	105	80	93	80	76	76	72	105
linear	moded_path	55	55	55	52	52	44	47	43	39	39	39	51
linear	mu	78	78	78	78	78	78	77	77	65	65	65	78

https://doi.org/10.1017/S1471068426100532 Published online by Cambridge University Press

Table 3. *Continued*

property	program	as_shlin2 (+)	as_shlin2_opt_mgu	as_shlin2_noopt_mgu	as_shlin_opt_opt	as_shlin_opt	as_shlin_noopt	as_shlin_opt_mgu	as_shlin_noopt_mgu	as_sharing (+)	as_sharing_mgu (+)	share	shfrlin
linear	nand	3885	3885	3885	3885	3885	3621	3341	3329	3132	3132	3132	3885
linear	nreverse	22	22	22	22	22	19	20	19	17	17	17	22
linear	ops8	101	101	101	101	101	101	97	97	80	80	80	101
linear	perfect	143	143	143	143	143	140	133	131	107	107	107	143
linear	pingpong	20	20	20	20	20	14	16	14	11	11	11	20
linear	poly_10	288	288	288	288	288	274	276	267	234	234	234	288
linear	prover	260	257	257	257	257	247	246	240	222	222	222	255
linear	qsort	55	55	55	55	55	49	52	48	41	41	41	55
linear	queens_8	89	89	89	89	89	83	85	83	70	70	70	89
linear	query	62	62	62	62	62	58	50	48	39	39	39	62
linear	reducer	1216	TO	TO	TO	672	563	TO	TO	418	TO	TO	710
linear	sendmore	469	469	469	469	469	469	354	354	342	342	342	469
linear	serialise	120	120	120	59	59	46	42	35	23	23	23	59
linear	sieve	85	85	85	85	85	82	84	81	68	68	68	82
linear	simple_analyzer	TO/OOM	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
linear	tak	104	104	104	104	104	94	89	84	66	66	66	104
linear	times10	101	101	101	101	101	101	97	97	80	80	80	101
linear	unify	1601	1395	1395	1414	1414	784	TO	TO	519	TO	519	1361
linear	zebra	OOM	TO	TO	4261	4261	TO	TO	TO	TO	TO	TO	TO

effect concerns the sharing component only, which is the dominant one, as the number of sharing groups may grow exponentially with the number of variables.

Finally, our implementation of the SHARING domain, when used with the most precise configuration, consistently outperforms the built-in SHARING implementation in CiaoPP; in some cases, it succeeds in terminating analyses for which the built-in implementation does not terminate.

5 Conclusions

The results of our experimental evaluation can be summarized as follows.

Optimality. The main outcome is that the use of optimal abstract operators does not compromise the feasibility of the analysis. Even if their definitions are more convoluted, optimal abstract operators can be implemented efficiently in practice. In many cases, they even lead to faster analyses. This behavior is likely a peculiarity of *may* abstract domains: the additional precision provided by optimal operators often results in smaller abstract objects. As a consequence, the effort required to compute the optimal operators is compensated by the reduced size of the resulting abstract objects. On the other hand, the benefits of optimal operators are not uniform and tend to be significant only in selected cases.

Matching. The use of matching consistently outperforms unification. It improves both precision and performance, and in several cases enables analyses that would otherwise time out when relying solely on unification. As a result, matching proves to be a particularly effective choice in practice.

As future work, we plan to extend the experimental evaluation by considering additional benchmark programs and improving the implementation of some abstract operators, in particular the optimal mgu for SHLIN and the matching operator for SHLIN^2, which still follow an unoptimized *generate-and-test* approach. We also plan to investigate the implementation in CiaoPP of numerical analyses, such as the parallelotope-based analysis by Amato *et al.* (2017), and more advanced fixpoint solvers, such as the ones implemented in ScalaFix (Amato and Scozzari 2023).

Competing interests

The authors declare none.

References

AMATO, G., MEO, M. C. and SCOZZARI, F. 2020. On collecting semantics for program analysis. *Theoretical Computer Science* 823, 1–25.

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., RUBINO, M. and SCOZZARI, F. 2017. Inferring linear invariants with parallelotopes. *Science of Computer Programming* 148, 161–188.

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

https://doi.org/10.1017/S1471068426100532 Published online by Cambridge University Press

Amato, G. and Scozzari, F. 2010. On the interaction between sharing and linearity. *Theory and Practice of Logic Programming* 10, 1, 49–112.

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

Amato, G. and Scozzari, F. 2023. The ScalaFix equation solver. In *25th International Symposium, FM 2023, Lübeck, Germany, March 6-10, 2023, Proceedings 2023*, M. Chechik, J.-P. Katoen and M. Leucker, Eds. Formal Methods, Vol. 14000 of Lecture Notes in Computer Science, Springer, Berlin Heidelberg, 142–159.

Amato, G. and Scozzari, F. 2024. Optimal matching for sharing and linearity analysis. *Theory and Practice of Logic Programming* 24, 5, 1–27.

Bagnara, R., Hill, P. M. and Zaffanella, E. 2002. Set-sharing is redundant for pair-sharing. *Theoretical Computer Science* 277, 3–46.

Bagnara, R., Zaffanella, E. and Hill, P. M. 2005. Enhanced sharing analysis techniques: A comprehensive evaluation. *Theory and Practice of Logic Programming* 5, 1–43.

Bruynooghe, M. 1991. A practical framework for the abstract interpretation of logic programs. *The Journal of Logic Programming* 1, 2/3, 91–124.

Bueno, F., Carro, M., Hermenegildo, M. V., López-García, P. and Morales, J. F. 1997. The Ciao System. Reference Manual. Technical Report CLIP-3/1997, Technical University of Madrid. Latest version available at https://ciao-lang.org/ciao/build/doc/ciao.html/.

Bueno, F., López-García, P., Morales, J. F., Puebla, G. and Hermenegildo, M. V. 2025. The Ciao Program Preprocessor. Technical Report CLIP-2/2025, Technical University of Madrid. Latest version available at https://ciao-lang.org/ciao/build/doc/ciaopp.html/.

Codish, M., Dams, D. and Yardeni, E. 1991. Derivation and safety of an abstract unification algorithm for groundness and aliasing analysis. In *Logic Programming, Proceedings of the Eighth International Conference 1991, Logic Programming*, K. Furukawa, Ed. The MIT Press, Cambridge, MA, USA, 79–93.

Codish, M., Mulkers, A., Bruynooghe, M., de la Banda, M. G. and Hermenegildo, M. 1995. Improving abstract interpretations by combining domains. *ACM Transactions on Programming Languages and Systems* 17, 1, 28–44.

Codish, M., Søndergaard, H. and Stuckey, P. J. 1999. Sharing and groundness dependencies in logic programs. *ACM Transactions on Programming Languages and Systems* 21, 5, 948–976.

Cortesi, A., Filé, G. and Winsborough, W. W. 1996. Optimal groundness analysis using propositional logic. *The Journal of Logic Programming* 27, 2, 137–167.

Cousot, P. and Cousot, R. 1992. Abstract interpretation frameworks. *Journal of Logic and Computation* 2, 4, 511–549.

Hans, W. and Winkler, S. 1992. Aliasing and groundness analysis of logic programs through abstract interpretation and its safety. Technical Report 92-27, RWTH Aachen. http://sunsite.informatik.rwth-aachen.de/Publications/AIB.

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.

Howe, J. M. and King, A. 2003. Three optimisations for sharing. *Theory and Practice of Logic Programming* 3, 2, 243–257.

Jacobs, D. and Langen, A. 1992. Static analysis of logic programs for independent and parallelism. *The Journal of Logic Programming* 13, 291–314.

Jurjio-Rivas, D., Morales, J. F., López-Gárcia, P. and Hermenegildo, M. V. 2024. Abstract environment trimming. *Theory and Practice of Logic Programming* 24, 4, 863–884.

King, A. 1994. A synergistic analysis for sharing and groundness which traces linearity. In *ESOP '94 Proceedings 1994*, Vol. 788 of LNCS, Springer, Berlin Heidelberg, 363–378.

https://doi.org/10.1017/S1471068426100532 Published online by Cambridge University Press

MUTHUKUMAR, K. and HERMENEGILDO, M. V. 1991. Combined determination of sharing and freeness of program variables through abstract interpretation. In *Logic Programming, Proceedings of the Eighth International Conference 1991, Logic Programming*, K. Furukawa, Ed. The MIT Press, Cambridge, MA, USA, 49–63.

MUTHUKUMAR, K. and HERMENEGILDO, M. V. 1992. Compile-time derivation of variable dependency using abstract interpretation. *The Journal of Logic Programming* 13, 2-3, 315–347.

https://doi.org/10.1017/S1471068426100532 Published online by Cambridge University Press