Demonstrating the Top-Down Solver in a Box

(Extended Abstract)

Marco Ciccalè
Universidad Politécnica de Madrid (UPM) & IMDEA Software Institute, Madrid, Spain
m.ciccale@alumnos.upm.es

José F. Morales
Universidad Politécnica de Madrid (UPM) & IMDEA Software Institute, Madrid, Spain
josef.morales@imdea.org

Pedro López-García
Spanish Council for Scientific Research (CSIC) & IMDEA Software Institute, Madrid, Spain
pedro.lopez@imdea.org

Manuel V. Hermenegildo
Universidad Politécnica de Madrid (UPM) & IMDEA Software Institute, Madrid, Spain
manuel.hermenegildo@imdea.org

Acknowledgments. Partially funded by MICIU program CEX2024-001471-M *María de Maeztu*. Thanks also to Daniel Jurjo and the anonymous reviewers for their useful feedback.

1 Introduction

The need to inspect and monitor the results and internal operation of complex solvers is well recognized, and static analyzers based on abstract interpretation [3] are no exception. When an analyzer fails to prove a desired property, understanding why may require inspecting abstract states at particular program points, monitoring the fixpoint computation, or debugging the implementation of the abstract domain operations. This is particularly challenging for demand-driven analyzers, where the computation depends on the paths and invocation contexts encountered during analysis.

The PLAI/TD analyzer in CiaoPP [7, 16] classically includes several tracing and visualization facilities, but these were only partially interactive previously to the work we present here. GobPie [8] and the Frama-C/EVA [12] abstract debugger provide source-level navigation but limited to the analysis results and inspection of abstract states, with the former operating after the analysis terminates and the latter incrementally during analysis. Mopsa's abstract debugger [14], in contrast, provides tracing and abstract debugging facilities but primarily for inspecting the analyzer itself.

We present and demonstrate recent extensions to the tracing facilities in the PLAI/TD analyzer in CiaoPP which together constitute a unified, configurable debugging framework for *top-down* static analyzers, combining source-level navigation with online observation of several different aspects of the analysis. Our contributions are: (§2) a procedure-box model of the top-down solver, exposing observables at multiple solver layers; (§3) an interactive debugger built on this model, integrated with local and web IDEs (including GNU Emacs, VS Code, and the online Playground); (§4) three complementary debugging scenarios: diagnosing faulty abstract domains, monitoring fixpoint computation, and abstract debugging. The discussion in this demonstration/short paper is necessarily brief; the reader is referred to [2] and the CiaoPP manuals [1] for details and links to the debugging framework, the intermediate representation, examples of analysis of imperative code, etc.

2 Procedure boxes for the top-down solver

Top-down analyzers are context-sensitive, demand-driven analyzers that build an *analysis graph* starting from a set of program *entry points* and analyze procedures and statements as they are encountered. Many top-down analyzers such as Goblint [21] and CiaoPP [6] are based on the *top-down solver*, or *top-down algorithm*, PLAI/TD.[1] First introduced in [15, 16] for the analysis of logic programs, it has also proved useful for other paradigms such as object-oriented programming [13], multi-threaded programs [21], and smart contracts [18].

Intermediate representation. To abstract from the programming model, we assume programs are translated to a *block-based* intermediate representation [4, 13], where the body of each block is a sequence of *statements*: basic commands and procedure calls. All statements are seen as procedure calls, with basic commands treated as calls to "built-in" procedures. Blocks are represented by *rules* (*clauses*), where the statements / procedure calls are the *literals* of these rules and loops are encoded as recursive calls among such rules. Conditionals, cases, etc. are encoded by several conditional rules. For each statement (*i.e.*, program point) and each abstraction of its calling context — each *call abstraction* — the analysis computes an abstraction of its resulting context, the *success abstraction*.

Procedure boxes. We model PLAI/TD using nested *procedure boxes*, one per layer of the algorithm, each exposing a set of observable *ports*. Figure 1 depicts the model, which we illustrate next by walking through the analysis of a literal (i.e., a procedure call or basic command). A Call event fires at the **Literal** box carrying the call abstraction, and control passes to the **Procedure** box, where the fixpoint computation takes place. FixpStart, FixpIter, and FixpEnd mark the beginning, each iteration, and end of the fixpoint computation.WidenCall and WidenSucc may fire to widen the call

[1] The original name of the algorithm is PLAI, but we use PLAI/TD to recall that it is the "top-down analysis algorithm."

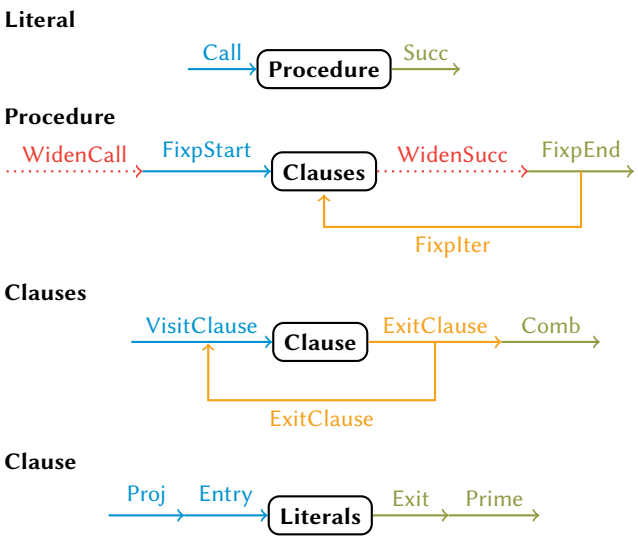

Figure 1. Procedure-box model for PLAI/TD.

and success abstractions.[2] Once stabilized, Succ fires carrying the corresponding success abstraction. Inside the **Clauses** box, all clauses are analyzed unconditionally. For each clause, VisitClause fires and control passes to the **Clause** box, where Proj and Entry project the call abstraction onto the goal and clause variables respectively. The **Literals** box then processes each literal in the clause body left to right: built-ins via the abstract transfer function, and procedure calls by recursively re-entering the **Literal** box. Once all literals are processed, Exit fires and Prime projects the exit abstraction back into the calling context. After all clauses have been analyzed, Comb combines all prime abstractions into the procedure's success abstraction.

3 Debugger for the top-down solver

We use the ports of the procedure-box model of figure 1 to define a concrete vocabulary of observable events. Each time a port event fires, a message is emitted carrying information specific to that event, of the form:

['+'] ⟨*inv-id*⟩ ⟨*node-id*⟩ ⟨*port*⟩ ':' ⟨*lit*⟩ ':' ⟨*abs*⟩ ⟨*info*⟩ ['?']

The fields common to all messages are as follows: ⟨*port*⟩, identifying the event; ⟨*inv-id*⟩, a unique invocation identifier; ⟨*node-id*⟩, the current node in the analysis graph; ⟨*lit*⟩, the literal under analysis; ⟨*abs*⟩, its current abstraction; and the optional + and ?, marking break-points and interactive prompt, respectively. The ⟨*info*⟩ field carries port-specific observables such as variable bindings and auxiliary information for abstraction ports; the fixpoint iteration count for fixpoint ports; the widening operands for widening ports; and the clause identifier for clause ports.

As the analysis progresses, the debugger presents emitted messages as source-level debugging events, controllable at three levels of granularity. At the event level, users can set *breakpoints* at program locations of interest and control execution one event at a time (*continue*/*creep*), skip over a procedure call's internal events (*step-over*/*skip*), or jump directly to the next breakpoint (*leap*). At the trace level, users can *hide* selected ports, filtering the trace to events of interest — *e.g.*, those of the abstract domain, the fixpoint computation, or abstract debugging. At the interaction level, users can *unleash* selected ports, turning their events non-interactive, so the debugger can also act as a tracer on the ports of interest, while the remaining, leashed ports stay interactive.

The debugger is implemented as a thin instrumentation layer over PLAI/TD in the Ciao system, intercepting events via hooks at each port without modifying the solver or the abstract domains. New events can be added by extending this layer alone. Together with the messages themselves, source location information is emitted for source highlighting within the host IDE. Users can configure and control the debugger on the fly through a simple read-eval loop.

4 Case studies

Abstract domain implementation. Consider a simple, non-relational types domain: $\bot \leqslant \mathsf{int}, \mathsf{atm} \leqslant \mathsf{gnd} \leqslant \top$, where int abstracts integers, atm abstracts atoms, and gnd abstracts any ground term (*i.e.*, data structure with no pointers).[3] Consider **lookup_name**/2, which looks up a name in a database, returning its integer identifier or no if not found:

```
lookup_name(Name, Id) :- db(Id, Name, ...).
lookup_name(Name, no).
```

We want to verify that **lookup_name**/2, when called with Name bound to an atom, always bounds Id to a ground term (either an integer or an atom), as expressed by the assertion:[4]

```
:- pred lookup_name/2 : atm * term => atm * gnd.
```

We see that, when analyzing the procedure with our simple types domain, the assertion cannot be verified: the analysis abstracts Id to ⊤ instead of gnd. To diagnose the issue, we set a break-point on **lookup_name**/2, hide all ports except for Call, Prime, and Comb, and only leash Comb, since we suspect the issue lies in the combination of the clause results rather than in the analysis of individual literals. Figure 2 shows the interactive debugging session. After analyzing both clauses, the Prime messages show the prime abstraction of each clause, together with the variable bindings that map clause variables back to the caller's variables. The abstraction of Id1 is int for the first clause, since **db**/n bounds Id1 to an integer; and atm for the second clause, since Id1 is bound to the atom no. The Comb message then shows that their combination, via the least upper bound (lub), yields ⊤ for Id1 instead of gnd, pointing directly to the implementation:

[2] We do not mention narrowing steps for brevity, treated similarly.

[3] The implementation of this abstract domain following the abstract domain interface of the Ciao system is given in appendix A.

[4] Types atm and gnd correspond to elements in the abstract domain lattice. See [19] for a detailed presentation of the assertion framework.

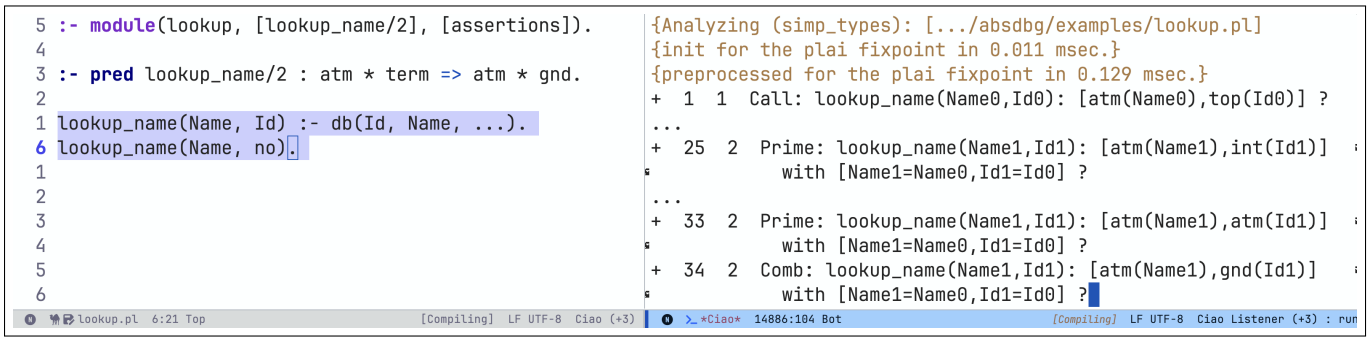

Figure 2. Screenshot of the interactive debugger for PLAI/TD running in the GNU Emacs editor. The left panel shows the source code and the right panel shows the debugger running in the interactive top-level.

```
lub_el(X, Y,   Y) :- less_or_equal_el(X, Y), !.
lub_el(X, Y,   X) :- less_or_equal_el(Y, X), !.
lub_el(_, _, top).
```

Since neither int ⩽ atm nor atm ⩽ int, **lub_el**/3 falls back to ⊤, which is imprecise. The fix is prepending:

```
lub_el(int, atm, gnd) :- !.
lub_el(atm, int, gnd) :- !.
```

Reanalyzing shows that the lub combination is now precise when combining both clauses:

```
+ 22 1 Comb: lookup_name(Name1,Id1): [atm(Name1),gnd(Id1)] ?
```

and the assertion is now verified.

Fixpoint computation. Consider **qplan**, a query planner program with 330 lines of code and 37 procedures whose clauses involve many variables with complex sharing patterns. Analyzing **qplan** with the Share+Cliques set-sharing abstract domain [9, 15, 17] takes around 7.6 seconds. To diagnose whether the solver is performing an excessive number of iterations, we run the debugger in non-interactive mode (*i.e.*, all ports unleashed), showing only fixpoint events. A representative selection of FixpEnd messages shows that the solver performs at most four iterations per procedure:

```
1466 21 FixpEnd: mark(P,L,Q0,Q1): ⟨abs⟩ (iter 3)
3427 46 FixpEnd: plan(Q2,V,C,G,Q3): ⟨abs⟩ (iter 2)
3875 98 FixpEnd: variablise(T,V0,V1): ⟨abs⟩ (iter 4)
```

The bottleneck, therefore, lies elsewhere.

We reconfigure the debugger to also show clause-level events, and we find abstractions of rather alarming size. For example, the success abstraction of **strip_keys**/2:

```
3421 80 Succ: strip_keys(L0,L1):
       [clique([[L0,L1,L2,L3,L4,L5,L6,L7,L8]]),
        sharing([[P,L],[P,L,Q0],[P,L,Q0,Q1],
                 [P,L,Q0,Q1,Q2],[P,L,Q0,Q1,Q2,Q3],
                 [P,L,Q0,Q1,Q2,Q3,V],...])
```

contains a set of 1024 sharing sets, each involving up to 12 variables. It is well known that set-sharing representations present scalability challenges despite their high precision. In the worst case, their size grows exponentially in the number of variables, when no sharing information can be discarded. To address this, *cliques* [17] collapse subsets of the abstraction into compact representations. However, the number of sharing sets can still grow large, as witnessed before. Another approach is *abstract environment trimming* [10], which, at the cost of a little overhead, dynamically modifies the domain of the abstractions as the analysis progresses reducing the size of the abstraction. Reanalyzing with abstract environment trimming enabled and the same conditions we have:

```
3421 80 Succ: strip_keys(L0,L1):
        [sharing([[L0],[L1],[L0,L1],[L0,L1,P],...])
```

The abstraction shrinks from 1024 sharing sets to 64, and the analysis time from 7.6 seconds to 800 milliseconds.

Abstract debugging. Consider the classic naive list reverse procedure **nrev**/2 using list concatenation:

```
nrev(    [], []).
nrev([H|L0], L2) :- nrev(L0, L1), conc(L1, [H], L2).
```

We want to verify that **nrev**/2, when called with a list of numbers and a free variable, yields a list of numbers and is deterministic, *i.e.*, it produces exactly one solution, as expressed by the following assertion:

```
:- pred nrev/2 : list(num) * var
              => list(num) * list(num) + det.
```

which involves properties of three different abstract domains: Share+Cliques to abstract sharing properties; ETerms [20], a weakly relational domain to abstract (parametric) types; and NfDet [5, 11], a combined domain to abstract determinacy and non-failure computational hyperproperties. We want to understand how each domain contributes to the verification of the assertion's properties. We start by running the debugger in interactive mode, showing only Call, Succ and fixpoint events, to start with a high-level picture of the analysis. The Succ message for the second iteration of ETerms shows:

```
33 1 Succ: nrev(L,L2): [rt7(L),rt6(L2),...] ?
```

where, *e.g.*, `rt7` is a list type (see figure 4 in the appendix).

We leap to the NfDet analysis directly, stepping over the sharing analysis. Given that NfDet abstracts computational properties, clause results and their combination are key to understanding the final result. We therefore hide all events

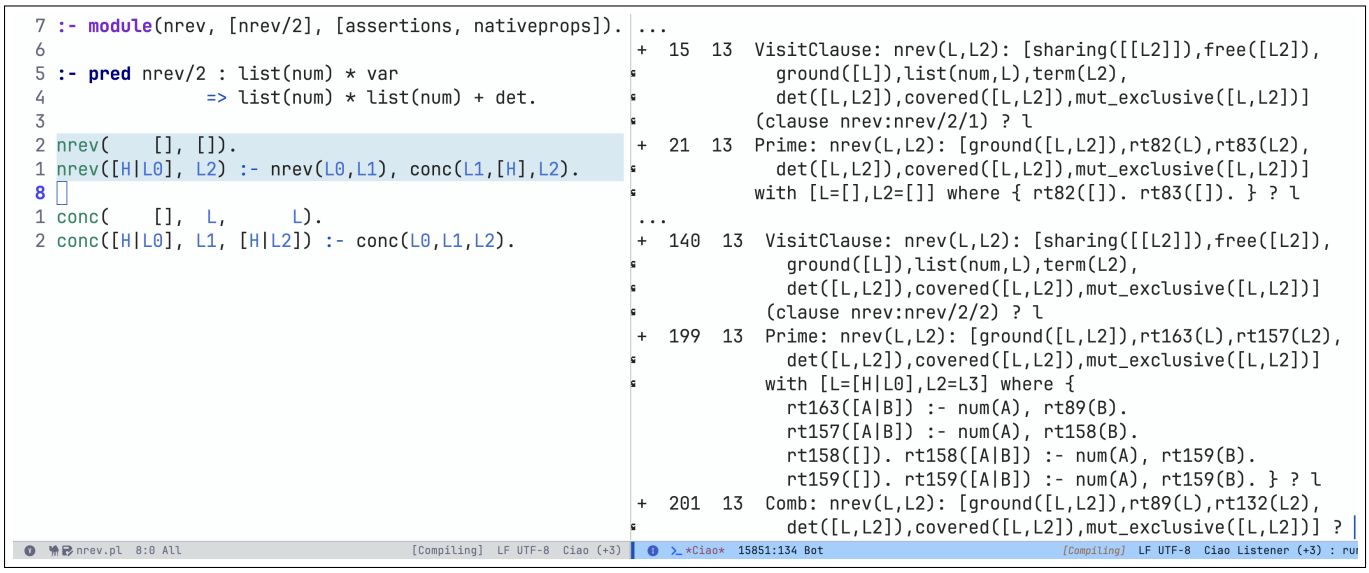

Figure 3. Screenshot of the interactive debugger running in the GNU Emacs editor for the **nrev**/2 NfDet analysis.

and show only VisitClause, Prime, and Comb. Figure 3 shows the Prime and Comb messages for **nrev**/2 under the NfDet domain. Notably, each NfDet abstraction carries sharing, typing, and determinism information about each of the variables. Finally, we see that Comb carries the combination of the two prime abstractions, confirming that **nrev**/2 is deterministic and does not fail when called with a list of numbers, and the assertion is verified.

References

[1] F. Bueno, P. Lopez-Garcia, J.F. Morales, G. Puebla, and M.V. Hermenegildo. [n. d.]. *The Ciao Program Preprocessor*. Technical Report. CLIP Lab, Technical University of Madrid (UPM) and IMDEA Software Institute. https://ciao-lang.org/ciao/build/doc/ciaopp.html/

[2] M. Ciccalè, J. F. Morales, P. López-García, and M.V. Hermenegildo. 2026. *The CiaoPP Abstract Debugger: a Box Model for the Top-Down Solver*. Technical Report CLIP-1/2026.0. CLIP Lab, IMDEA Software and Tech. U. of Madrid (UPM).

[3] P. Cousot and R. Cousot. 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.

[4] E. De Angelis, F. Fioravanti, J.P. Gallagher, M.V. Hermenegildo, A. Pettorossi, and M. Proietti. 2021. Analysis and Transformation of Constrained Horn Clauses for Program Verification. *TPLP* 22, 6 (2021), 1–69.

[5] S.K. Debray, P. Lopez-Garcia, and M.V. Hermenegildo. 1997. Non-Failure Analysis for Logic Programs. In *ICLP'97*. MIT Press, 48–62.

[6] M.V. Hermenegildo, F. Bueno, M. Carro, P. Lopez-Garcia, E. Mera, J.F. Morales, and G. Puebla. 2012. An Overview of Ciao and its Design Philosophy. *TPLP* 12, 1–2 (2012), 219–252.

[7] M.V. Hermenegildo, G. Puebla, F. Bueno, and P. Lopez Garcia. 2005. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming* 58, 1–2 (2005), 115–140.

[8] K. Holter, J.O. Hennoste, P. Lam, S. Saan, and V. Vojdani. 2024. Abstract Debuggers: Exploring Program Behaviors using Static Analysis Results. In *ACM Onward!'25*. ACM, NYC, USA, 130–146.

[9] D. Jacobs and A. Langen. 1989. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In *1989 North American Conference on Logic Programming*. MIT Press.

[10] D. Jurjo-Rivas, J.F. Morales, P. López-García, and M.V. Hermenegildo. 2024. Abstract Environment Trimming. *TPLP* 24, 4 (2024), 863–884.

[11] P. Lopez-Garcia, F. Bueno, and M.V. Hermenegildo. 2005. Determinacy Analysis for Logic Programs Using Mode and Type Information. In *LOPSTR'04 (LNCS, 3573)*. Springer-Verlag, 19–35.

[12] J. Massart. 2026. The Eva Abstract Debugger: A new way to explore and interact with the Frama-C/Eva static analyzer. https://www.frama-c.com/2026/02/02/Eva-abstract-debugger.html. (Visited on July 2026).

[13] M. Méndez-Lojo, J. Navas, and M.V. Hermenegildo. 2007. A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In *LOPSTR (LNCS, Vol. 4915)*.

[14] R. Monat, A. Ouadjaout, and A. Miné. 2024. Easing maintenance of academic static analyzers. *Int'l. Journal on Software Tools for Technology Transfer* 26, 6 (2024), 673–686. doi:10.1007/s10009-024-00770-1

[15] K. Muthukumar and M.V. Hermenegildo. 1989. Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In *NACLP'89*. MIT Press, 166–189.

[16] K. Muthukumar and M.V. Hermenegildo. 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). https://cliplab.org/papers/mcctr-fixpt.pdf

[17] J. Navas, F. Bueno, and M.V. Hermenegildo. 2006. Efficient Top-Down Set-Sharing Analysis Using Cliques. In *8th PADL (LNCS, 2819)*. Springer, 183–198.

[18] V. Perez-Carrasco, M. Klemen, P. Lopez-Garcia, J.F. Morales, and M.V. Hermenegildo. 2020. Cost Analysis of Smart Contracts via Parametric Resource Analysis. In *Static Analysis Simposium (SAS'20) (LNCS, Vol. 12389)*. Springer, 7–31. doi:10.1007/978-3-030-65474-0_2

[19] G. Puebla, F. Bueno, and M.V. Hermenegildo. 2000. Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs. In *Proc. of LOPSTR'99 (LNCS 1817)*. Springer-Verlag, 273–292.

[20] C. Vaucheret and F. Bueno. 2002. More Precise yet Efficient Type Inference for Logic Programs. In *SAS'02 (LNCS, 2477)*. Springer, 102–116.

[21] V. Vojdani, K. Apinis, V. R otov, H. Seidl, V. Vene, and R. Vogler. 2016. Static race detection for device drivers: the Goblint approach. In *IEEE/ACM Int'l. Conf. on Automated Software Engineering*. ACM, NYC, USA, 391–402. doi:10.1145/2970276.2970337

A Supplementary material

```
:- module(simp_types, [
    less_or_equal_el/2, glb_el/3, lub_el/3,
    abstract_term/3, abstract_literal/4, known_builtin/1
], []).

% Abstract domain interface
:- include(domain(basicdom_nonrel_base)).

less_or_equal_el(X, X) :- !.
less_or_equal_el(_, top) :- !.
less_or_equal_el(bot, _) :- !.
less_or_equal_el(int, gnd) :- !.
less_or_equal_el(atm, gnd).

glb_el(X, Y, X) :- less_or_equal_el(X, Y), !.
glb_el(X, Y, Y) :- less_or_equal_el(Y, X), !.
glb_el(_, _, bot).

lub_el(int, atm, gnd) :- !. % missing case in section 4!
lub_el(atm, int, gnd) :- !. % missing case in section 4!
lub_el(X, Y, Y) :- less_or_equal_el(X, Y), !.
lub_el(X, Y, X) :- less_or_equal_el(Y, X), !.
lub_el(_, _, top).

abstract_term(X, _, int) :- integer(X), !.
abstract_term(X, _, atm) :- atom(X), !.
abstract_term(X, _, gnd) :- ground(X), !.
abstract_term(X, A, Val) :- var(X), !,
    current_value(A, X, Val).
abstract_term(_, _, top).

abstract_literal('integer/1', integer(V), Call, Succ) :- !,
    replace_value(Call, V, int, Succ).
abstract_literal('atom/1', atom(V), Call, Succ) :- !,
    replace_value(Call, V, atm, Succ).
abstract_literal('ground/1', ground(V), Call, Succ) :- !,
    replace_value(Call, V, gnd, Succ).

known_builtin('integer/1').
known_builtin('atom/1').
known_builtin('ground/1').

% Auxiliar procedures
current_value([X/Val|A], V, Val) :- X == V, !.
current_value([_|A], V, Val) :-
    current_value(A, V, Val).

replace_value([], _, _, []).
replace_value([X/_|A0], V, Val, [X/Val|A0]) :- X == V, !.
replace_value([X/Val0|A0], V, Val1, [X/Val0|A1]) :-
    replace_value(A0, V, Val1, A1).
```

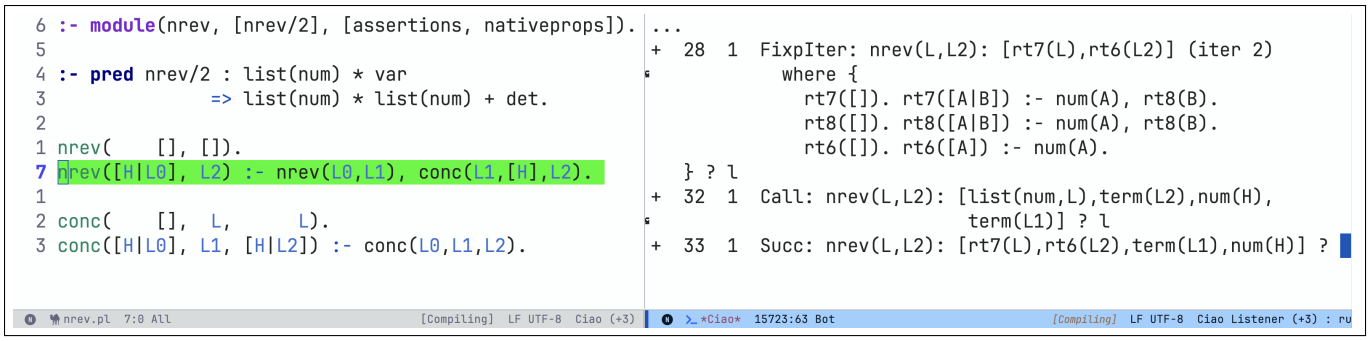

Figure 4. Screenshot of the interactive debugger running in the GNU Emacs editor for the **nrev**/2 ETerms analysis.

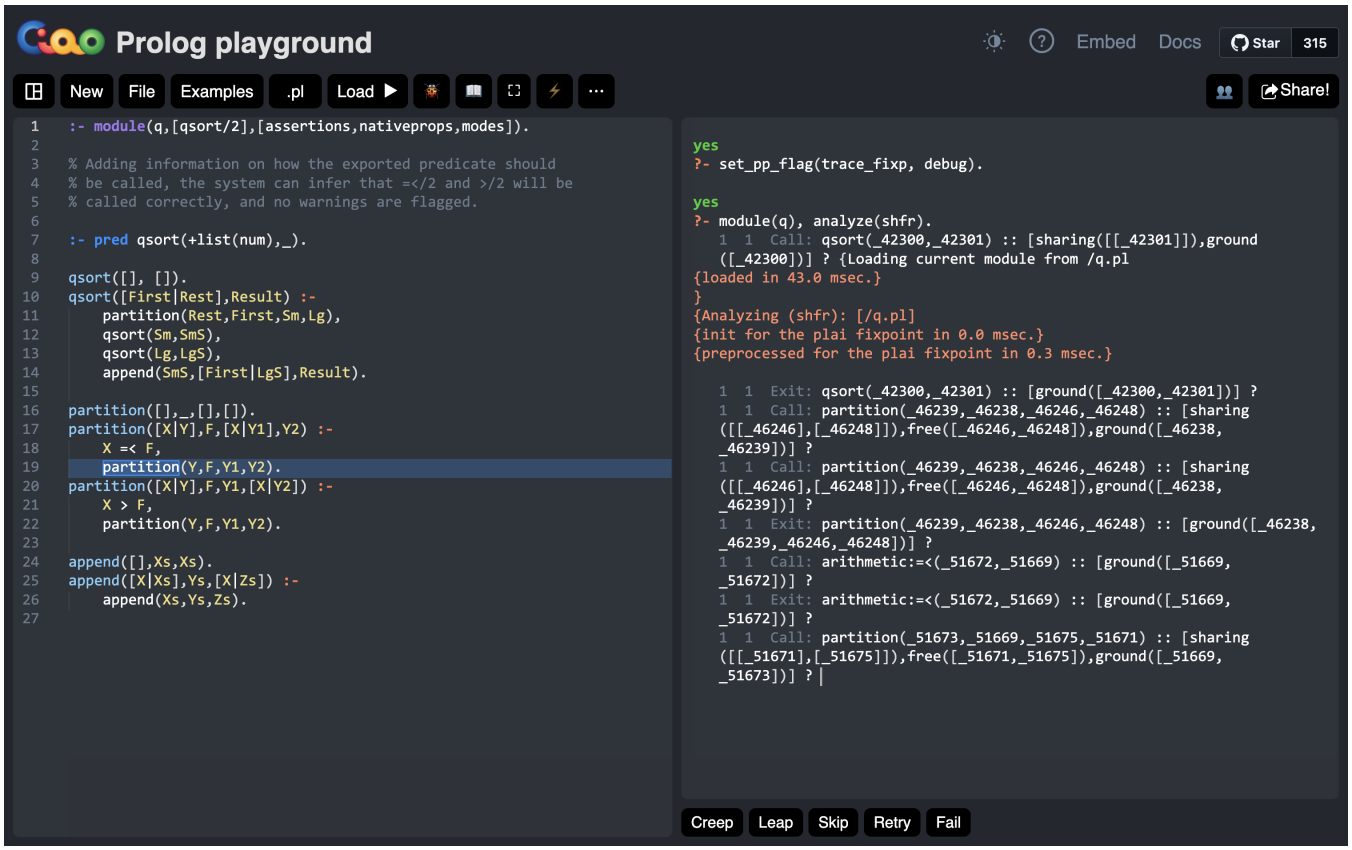

Figure 5. Screenshot of the interactive debugger running in the Ciao web Playground: https://ciao-lang.org/playground/.