On Improving Run-time Checking in Dynamic Languages[*]

Nataliia Stulova[†]

IMDEA Software Institute, Madrid, Spain and
Universidad Politécnica de Madrid (UPM), Madrid, Spain
nataliia.stulova@imdea.org

Abstract

In order to detect incorrect program behaviors, a number of approaches have been proposed, which include a combination of language-level constructs (procedure-level annotations such as assertions/contracts, gradual types, etc.) and associated tools (such as static code analyzers and run-time verification frameworks). However, it is often the case that these constructs and tools are not used to their full extent in practice due to a number of limitations such as excessive run-time overhead and/or limited expressiveness. This issue is especially prominent in the context of dynamic languages without an underlying strong type system, such as Prolog. In our work we propose several practical solutions for minimizing the run-time overhead associated with assertion-based verification while keeping the correctness guarantees provided by run-time checks. We present the solutions in the context of the Ciao system, where a combination of an abstract interpretation-based static analyzer and run-time verification framework is available, although our proposals can be straightforwardly adapted to any other similar system.

1998 ACM Subject Classification D.3.3 Language Constructs and Features, D.2.4 Software/Program Verification, D.2.5 Testing and Debugging, F.3.1 Specifying and Verifying and Reasoning about Programs, F.3.3 Studies of Program Constructs, F.3.2 Semantics of Programming Languages

Keywords and phrases Runtime Verification, Assertions, Prolog, Logic Programming

Digital Object Identifier 10.4230/OASIcs.ICLP.2017.15

1 Introduction

Detecting incorrect program behaviors is an important part of the software development life cycle. It is also a complex and tedious one, in which dynamic languages bring special challenges.

A number of techniques have been proposed to aid in the process, among which we center our attention on the use of language-level constructs to describe expected program behavior, and of associated tools to compare actual program behavior against expectations, such as static code analyzers/verifiers and run-time verification frameworks.

Approaches that fall into this category are the assertion-based frameworks used in (Constraint) Logic Programming [10, 24, 16, 19], soft/gradual typing approaches in functional

[*] Research supported in part by projects EU FP7 318337 *ENTRA*, Spanish MINECO TIN2012-39391 *StrongSoft*, TIN2015-67522-C3-1-R *TRACES* and TIN2008-05624 *DOVES*, and Comunidad de Madrid TIC/1465 *PROMETIDOS-CM* and S2013/ICE-2731 *N-Greens Software*, and Madrid Region program M141047003 *N-GREENS*.

[†] Supervised by José F. Morales (IMDEA Software Institute, Madrid, Spain) and Manuel V. Hermenegildo (IMDEA Software Institute, Madrid, Spain and Universidad Politécnica de Madrid (UPM), Madrid, Spain)

programming [5, 34, 32] and contract-based extensions in object-oriented programming [17, 18, 11]. These tools are aimed at detecting violations of the expected behavior or certifying the absence of any such violations, and often involve a certain degree of run-time testing, specially for non-trivial properties.

In practice, however, run-time overhead often remains impractically high, specially for complex properties, such as, for example, deep data structure tests. This reduces the attractiveness of run-time checking to programmers, which may allow sporadic checking of very simple conditions, but tend to turn off run-time checking for more complex properties. Some approaches even opt for limiting the expressiveness of the assertion language in order to reduce the overhead.

Our research objective is twofold:

- First, we aim for enhancing the expressiveness of the assertion language to reflect all the features of the related programming language, including, e.g., higher-order constructs, and to do so in a way that allows the programmer to write precise program specifications while not imposing a learning or programming burden on them.
- At the same time, our goal is to efficiently check such specifications, mitigating the associated run-time overhead as much as possible without compromising the safety guarantees that the checks provide.

While our work is general and system-independent, we present it for concreteness in the context of the Ciao run-time checking framework. The Ciao model [13, 24] is well understood, and different aspects of it have been incorporated in popular (C)LP systems, such as Ciao, SWI, and XSB [14, 31, 21].

2 Current Research Results

2.1 Supporting Higher-Order Properties

Higher-order programming is a widely adopted programming style that adds flexibility to the software development process. Within the (Constraint) Logic Programming ((C)LP) paradigm, Prolog has included higher-order constructs since the early days, and there have been many other proposals for combining the first-order kernel of (C)LP with different higher-order constructs, e.g., [35, 23, 4]). Many of these proposals are currently in use in different (C)LP systems and have been found very useful in programming practice, inheriting the well-known benefits of code reuse (templates), elegance, clarity, and modularization.

When higher-order constructs are introduced in the language it becomes necessary to describe properties of arguments of predicates/procedures that are themselves also predicates/procedures. While the combination of contracts and higher-order has received some attention in functional programming [12, 9], within (C)LP the combination of higher-order with the previously mentioned assertion-based approaches has received comparatively little attention to date. Current Prolog systems simply use basic atomic types (i.e., stating simply that the argument is a `pred`, `callable`, etc.) to describe predicate-bearing variables (see Listing 1). Other approaches [1] are more oriented instead towards meta programming, describing meta-types but there is no notion of directionality (modes), and only a single pattern is allowed per predicate.

Our proposal [26] contributes towards filling this gap between higher-order (C)LP programs and assertion-based extensions for error detection and program validation. Our starting point is the Ciao assertion model, which we have enhanced with a new class of properties, "predicate properties" (or *predprops*), for which we have proposed a syntax and semantics.

Listing 1 A simple program with a higher-order predicate `min/4` that accepts a custom comparator predicate

```
:- pred min(X,Y,Cmp,Min) : callable(Cmp).

min(X,Y,P,Min) :- P(R,X,Y), R <= 0, Min = X.
min(X,Y,_,Y  ).

less( 0,A,A).             lt('=',A,A).
less(-1,A,B) :- A < B.    lt('<',A,B) :- A < B.
less( 1,_,_).             lt('>',_,_).

test_min :- min(4,2,lt,2). % lt/3 is passed, but less/3 is expected
```

Listing 2 A *predprop* `comparator` example and an *anonymous* assertion in its definition.

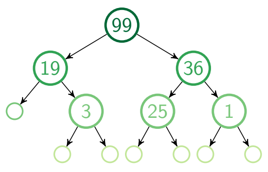
```
:- comparator(Cmp) {
  :- pred Cmp(Res,M,N) : (num(M), num(N)) => between(-1,1,Res).
}.

:- pred min(X,Y,Cmp,Min) : comparator(Cmp).
```

These new properties can be used in assertions for higher-order predicates to describe the properties of the higher-order arguments. An example of a predprop is provided in Listing 2, where an *anonymous* assertion (note the variable symbol `Cmp` in place of a predicate symbol) is used to describe a comparison predicate, that is not known at compilation time. By reusing the original assertion language syntax to describe call and success conditions of predicate-bearing arguments we allow both for better integration of these new constructs into the verification framework and at the same time lessening the burden on a programmer, who needs to provide such annotations.

Our predprop properties specify conditions for predicates that are independent of the usage context. This corresponds in functional programming to the notion of *tight* contract satisfaction [9], and it contrasts with alternative approaches such as *loose* contract satisfaction [12]. In the latter, contracts are attached to higher-order arguments by implicit function wrappers. The scope of checking is local to the function evaluation. Although this is a reasonable and pragmatic solution, we believe that our approach is more general and more amenable to combination with static verification techniques. For example, avoiding wrappers allows us to remove checks (e.g., by static analysis) without altering the program semantics. Moreover, our approach can easily support *loose* contract satisfaction, since it is straightforward in our framework to optionally include wrappers as special predprops.

2.2 Trading Memory for Speed

While having become an integral part of software development process, run-time testing can generally incur a high penalty in execution time and/or space over the standard, test-less program execution. A number of techniques have been proposed to date to reduce this overhead, including simplifying the checks at compile time via static analysis [2, 13] or reducing the frequency of checking, including for example testing only at a reduced number of points [19, 20]. Our proposal of [27] describes an approach to run-time testing that is efficient while being minimally obtrusive and remaining exhaustive. It is based on the use of memoization to cache intermediate results of check evaluation in order to avoid repeated checking of previously verified properties over the same data structure.



```
tree(e).
tree(t(L,_,R)) :- tree(L), tree(R).

t(                ,99,                          )
  t( ,19,          )    t(            ,36,         )
    e      t( ,3, )        t( ,25, )     t( ,1, )
               e   e           e    e        e   e

```

Figure 1 A minimalistic tree data structure implementation as a regular type `tree/1`.

Memoization has of course a long tradition in (C)LP in uses such as tabling resolution [33, 8]. Memoization has also been used in program analysis [36, 22], where tabling resolution is performed using abstract values. However, in tabling and program analysis it is call-success patterns that are usually tabled, whereas in our case the aim is to cache the results of test execution.

We concentrate our attention on checks for conformance of run-time heap structures to *regular types* [7], a useful subset of properties that are often used in assertions. An example in Fig. 1 shows a binary tree (left) and its possible implementation as a regular type together with an instance of that type describing the tree (right).

Our approach is based on the observation that run-time checks of regular types are monotonic instantiation checks, i.e., terms only become more and more instantiated with every subsequent state that the program enters.[1] We extend the Ciao run-time checking framework with a common cache, accessible to run-time checks, that stores tuples (x, t), where x is a term address and t is an identifier of a regular type. After the initial check on the term is performed, the corresponding tuple is added to the cache and for all the subsequent checks on the term, the check is replaced by a (computationally cheaper) cache lookup operation.

While studying the resulting performance of the enhanced verification framework we have observed that a straightforward use of a cache does not necessarily lead to a significant reduction in checking cost. An important issue that must be taken into account is the character of cache rewrites: as the terms grow in size cache collisions happen more often, which leads to element eviction. This in turn cancels out the advantage that using a cache gives: the ability to just lookup the regular type of some term without actually performing the check of it. However, this effect can be remedied by limiting the depth of terms that are stored in the cache (e.g., not caching terms with depth more than some n).

The idea of using memoization techniques to speed up checks has attracted some attention recently [15]. Their work (developed independently from ours) is based on adding fields to data structures to store the properties that have been checked already for such structures. In contrast, our approach has the advantage of not requiring any modifications to data structure representation, or to the checking code, program, or core run-time system.

2.3 Intertwining Compile- and Run-time Checks

A complementary approach to run-time overhead reduction consists in using static analysis to minimize the number and cost of the run-time checks that need to be placed in the program to detect incorrect program behaviors. This idea was pioneered by the Ciao system where a

[1] This is not the case of course if the language allows mutable variables and they are used in the program, but in those cases variable inmutability is tracked by the compiler / preprocessor.

number of (abstract interpretation-based) static analyses are combined in order to verify assertions to the largest extent possible at compile time, and for simplifying and reducing the number of remaining properties that need to be introduced in the program as run-time checks. However, while there has been evidence from use, there has been little systematic experimental work presented to date measuring the actual impact of analysis on reducing run-time checking overhead.

In our work [28] we generalize the existing practices as four *assertion checking modes*, each of which represents a trade-off between code annotation depth, execution time slowdown, and program behavior safety guarantees:

- *Unsafe*: no run-time checks are generated from program assertions, program execution is fast but incorrect program behaviors may stay undetected; the run-time overhead is nonexistent.
- *Client-Safe*: run-time checks are generated from the assertions of the program's interface (providing behavior guarantees for its clients), yet internal program assertions remain unchecked; the run-time overhead is taken as a minimal unavoidable one.
- *Safe-RT*: run-time checks are generated from all program assertions; this mode of checking is characterized with the strongest behavior safety guarantees yet is at the same time associated with highest check costs;
- *Safe-CT-RT*: a variation of *Safe-RT* checking mode with an additional static analysis phase, during which some of program assertions may be proven to always hold and generating run-time checks from them can be omitted; depending on the kind of analysis and the program the overhead generated by the checks from remaining assertions may be closer to that of *Client-Safe* *Safe-RT* modes.

We also define a transformation-based approach in order to implement each one of these modes. The differences between the transformations used in different modes are illustrated in Fig. 2. Starting with the source of the program and its assertions in the bottom-left corner, a sequence of preprocessing and compilation steps (denoted by arrows) is applied. Respective components of the verification framework that perform the source transformations are represented as hexagons.

We then concentrate on the reduction of the number of run-time tests via (abstract interpretation-based) program analysis. To this end we propose a technique that enhances analysis precision by taking into account that any assertions that cannot be proved statically will be the subject of run-time testing. In practice, it means that it is safe to make the two following assumptions for any predicate that has an assertion specifying properties that should hold on calls and success:

- The calls conditions hold after the analysis has entered the predicate definition, since either the checks for these calls conditions have already succeeded or the program has exited with error.
- The relevant success conditions hold after the predicate has exited, since, again, at this point either these success conditions have already succeeded or the program has exited with error.

2.4 Benefiting from Information Hiding

While dynamic languages offer programmers great flexibility in term creation and manipulation, for the very same reason the need for exhaustive run-time checks arises in order to guarantee the data manipulation safety and correctness. Reusable libraries, i.e., library modules that are pre-compiled independently of the client, pose special challenges in this

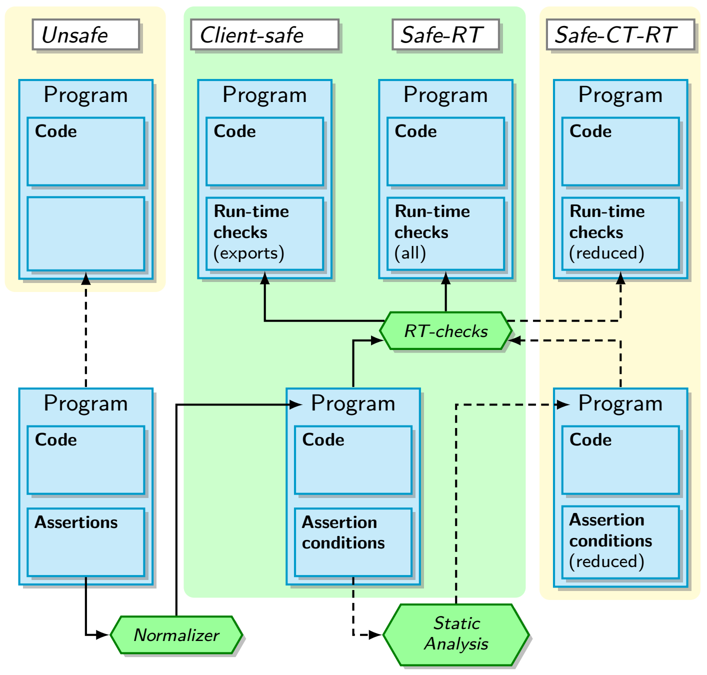


Figure 2 Source transformation differences per checking mode.

context. The key issue here is that there is virtually no control on how and where valid terms can be created, and thus it is quite common in the client-library interaction that any of these modules can create any data shape and pass it. As a consequence, (often expensive) run-time checks on the module boundaries become a necessary evil. In our work [29] we propose a possible solution to overhead reduction for run-time checks on module boundaries based on the *information hiding* principle (which is adopted in many other systems in form of encapsulation or opaque data types).

Currently, most mature Prolog implementations adopt some flavor of a module system, *predicate-based* in SWI [37], SICStus [30], YAP [25], ECLiPSe [6], and *atom-based* in XSB [31]. The difference between the two systems is the strictness of the term visibility rules: in an atom-based system local to the module terms are not visible outside it if they are not a part of the module interface. The Ciao approach [3] has until now been closer to a predicate-based module system.

We propose an extension of the predicate-based module system, that allows to specify only a subset of module terms as local (*hidden*) ones. Our argument is that in this setup we have the guarantee of the module terms' structural homogeneity, as only one module is allowed to construct/deconstruct some particular data shapes. With this there is no need to perform thorough checks of properties that verify the correctness of the data term structure at the module boundaries. Instead, it would suffice to perform checks of *shallow* versions of data shape properties: the weakened forms of the original properties that are semantically equivalent to them in the context of the possible program executions. These versions typically require asymptotically fewer execution steps and in some cases of the calls across module boundaries allow us to achieve constant run-time overhead.

To illustrate the idea behind the approach, let us consider that the `tree/1` regular type from Fig. 1 is defined in a following module `bintrees`:

```
:- module(bintrees,[tree/1,add/3,del/3,find/2]). % exported predicates

:- hide e/0.                                     % \__ hidden
:- hide t/3.                                     % /   functors

:- regtype tree/1.
tree(e).
tree(t(L,_,R)) :- tree(L), tree(R).

:- pred add(El,T0,T1) : tree(T0) => tree(T1).
add(El,T0,T1) :- ...

...  % rest of the implementation of module predicates
```

If we can assure by static analysis that none of the exported predicates of `bintrees` passes `t(_,_,_)` terms outside the module, which could allow module clients to construct arbitrary terms with this functor (e.g., `t([...],_,[...])`), then we can safely substitute the full `tree/1` property in the precondition check for `add/1` (and also in precondition checks for other exported predicates!) by its *shallow* version:

```
:- regtype shallow_tree/1.
shallow_tree(e).
shallow_tree(t(_,_,_)).

:- pred add(El,T0,T1) : shallow_tree(T0) => tree(T1).
add(El,T0,T1) :- ...
```

This way in all calls across module `bintrees` boundaries the cost of precondition checks goes down from linear in the size of the term to constant (functor correctness check).

Preliminary experimental results and details about the algorithms that perform the inference of shallow properties and conditions under which it is safe to use them in run-time checks are provided in a technical report, to which [29] refers to.

3 Conclusions and Future Work

The specification-based runtime verification approach has attracted significant interest in recent decades, both from academia and industry. As it is the case with any other open problem, finding an approach that would suit each and every system is hard, and thus opting for tailored partial solutions is more practical.

In our work we have concentrated on the peculiarities of the run-time verification task in the context of dynamic languages and their use in (C)LP systems. We have proposed an enhancement for the Ciao assertion language that allows us to capture the concrete execution contexts of higher-order terms and thus adapts the verification framework to this new use case. We have also proposed several solutions for reducing the overhead associated with run-time checks, that treat the issue from several different angles.

While for concreteness of presentation our work was carried out within the Ciao language and combined static/dynamic verification framework, our results are general and system-independent. We believe they can be straightforwardly transferred to the contexts of other declarative languages. In addition, given the advances in verification of a wide class of programming languages, including imperative languages, by translation into Horn clauses and proving properties at this level, and the fact that this approach is fully supported in the Ciao system, we argue that our results can easily be adapted to a much broader spectrum of languages.

For future work we plan to concentrate first on fully incorporating the optimization techniques described here into Ciao's run-time verification framework, as now they are available as separate system bundles. Among the issues we would also like to address more profoundly are further developments on blame assignment in the case of higher-order assertion checking, scalability evaluation of the combination of all optimization techniques, and providing on-line demos and documentation.

Acknowledgements. We would like to thank the anonymous reviewers for providing valuable comments and suggestions that had helped to improve this paper.

References

1 C. Beierle, R. Kloos, and G. Meyer. A Pragmatic Type Concept for Prolog Supporting Polymorphism, Subtyping, and Meta-Programming. In *Proc. of the ICLP'99 Workshop on Verification of Logic Programs, Las Cruces*, Electronic Notes in Theoretical Computer Science, volume 30, issue 1. Elsevier, 2000. URL: `http://www.elsevier.nl/locate/entcs/volume30.html`.

2 F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszynski, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 155–170, Linköping, Sweden, May 1997. U. of Linköping Press. URL: `ftp://cliplab.org/pub/papers/aadebug_discipldeliv.ps.gz`.

3 D. Cabeza and M. Hermenegildo. A New Module System for Prolog. In *International Conference on Computational Logic, CL2000*, number 1861 in LNAI, pages 131–148. Springer-Verlag, July 2000.

4 D. Cabeza, M. Hermenegildo, and J. Lipton. Hiord: A Type-Free Higher-Order Logic Programming Language with Predicate Abstraction. In *Ninth Asian Computing Science Conference (ASIAN'04)*, number 3321 in LNCS, pages 93–108. Springer-Verlag, December 2004.

5 Robert Cartwright and Mike Fagan. Soft Typing. In *Programming Language Design and Implementation (PLDI 1991)*, pages 278–292. SIGPLAN, ACM, 1991.

6 Cisco Systems. *ECLIPSE User Manual*, 2006.

7 P.W. Dart and J. Zobel. Efficient Run-Time Type Checking of Typed Logic Programs. *Journal of Logic Programming*, 14:31–69, October 1992.

8 S. Dietrich. *Extension Tables for Recursive Query Evaluation.* PhD thesis, Departament of Computer Science, State University of New York, 1987.

9 Christos Dimoulas and Matthias Felleisen. On contract satisfaction in a higher-order world. *ACM Trans. Program. Lang. Syst.*, 33(5):16, 2011. `doi:10.1145/2039346.2039348`.

10 W. Drabent, S. Nadjm-Tehrani, and J. Małuszyński. The Use of Assertions in Algorithmic Debugging. In *Proceedings of the Intl. Conf. on Fifth Generation Computer Systems*, pages 573–581, 1988.

11 Manuel Fähndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In *Proceedings of the 2010 International Conference on Formal Verification of Object-oriented Software*, volume 6528 of *Lecture Notes in Computer Science*, pages 10–30, Berlin, Heidelberg, 2011. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=1949303.1949305`.

12 Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In Mitchell Wand and Simon L. Peyton Jones, editors, *ICFP*, pages 48–59. ACM, 2002. `doi:10.1145/581478.581484`.

13 M. Hermenegildo, G. Puebla, and F. Bueno. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: a 25–Year Perspective*, pages 161–192. Springer-Verlag, July 1999.

14 M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, J.F. Morales, and G. Puebla. An Overview of Ciao and its Design Philosophy. *Theory and Practice of Logic Programming*, 12(1–2):219–252, January 2012. http://arxiv.org/abs/1102.5497. `doi:10.1017/S1471068411000457`.

15 Emmanouil Koukoutos and Viktor Kuncak. Checking Data Structure Properties Orders of Magnitude Faster. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification*, volume 8734 of *Lecture Notes in Computer Science*, pages 263–268. Springer International Publishing, 2014. `doi:10.1007/978-3-319-11164-3_22`.

16 Claude Laï. Assertions with Constraints for CLP Debugging. In Pierre Deransart, Manuel V. Hermenegildo, and Jan Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, volume 1870 of *Lecture Notes in Computer Science*, pages 109–120. Springer, 2000. `doi:10.1007/10722311_4`.

17 Leslie Lamport and Lawrence C. Paulson. Should your specification language be typed? *ACM Transactions on Programming Languages and Systems*, 21(3):502–526, May 1999.

18 Gary T. Leavens, K. Rustan M. Leino, and Peter Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Asp. Comput.*, 19(2):159–189, 2007. `doi:10.1007/s00165-007-0026-7`.

19 E. Mera, P. López-García, and M. Hermenegildo. Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework. In *25th Int'l. Conference on Logic Programming (ICLP'09)*, volume 5649 of *LNCS*, pages 281–295. Springer-Verlag, July 2009.

20 E. Mera, T. Trigo, P. López-García, and M. Hermenegildo. Profiling for Run-Time Checking of Computational Properties and Performance Debugging. In *Practical Aspects of Declarative Languages (PADL'11)*, volume 6539 of *Lecture Notes in Computer Science*, pages 38–53. Springer-Verlag, January 2011.

21 Edison Mera and Jan Wielemaker. Porting and refactoring Prolog programs: the PROSYN case study. *TPLP*, 13(4-5-Online-Supplement), 2013.

22 K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, July 1992.

23 Gopalan Nadathur and Dale Miller. Higher–Order Logic Programming. In D. Gabbay, C. Hogger, and A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5. Oxford University Press, 1998.

24 G. Puebla, F. Bueno, and M. Hermenegildo. Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs. In *Logic-based Program Synthesis and Transformation (LOPSTR'99)*, number 1817 in LNCS, pages 273–292. Springer-Verlag, March 2000.

25 Vítor Santos Costa, Luís Damas, and Ricardo Rocha. The YAP Prolog System. *Theory and Practice of Logic Programming*, 2011. http://arxiv.org/abs/1102.3896v1.

26 N. Stulova, J. F. Morales, and M. V. Hermenegildo. Assertion-based Debugging of Higher-Order (C)LP Programs. In *16th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'14)*. ACM Press, September 2014.

27 N. Stulova, J. F. Morales, and M. V. Hermenegildo. Practical Run-time Checking via Unobtrusive Property Caching. *Theory and Practice of Logic Programming, 31st Int'l. Conference on Logic Programming (ICLP'15) Special Issue*, 15(04-05):726–741, September 2015. URL: `http://arxiv.org/abs/1507.05986`.

28 N. Stulova, J. F. Morales, and M. V. Hermenegildo. Reducing the Overhead of Assertion Run-time Checks via static analysis. In *18th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'16)*, pages 90–103. ACM Press, September 2016.

29 N. Stulova, J. F. Morales, and M. V. Hermenegildo. Term Hiding and its Impact on Run-time Check Simplification (Extended Abstract). In *Proceedings of the Technical Com-*

munications of the 33rd International Conference on Logic Programming (ICLP 2017), Melbourne, Australia, August 28 - September 1, 2017. OASIcs, August 2017.

30 Swedish Institute for Computer Science, PO Box 1263, S-164 28 Kista, Sweden. *SICStus Prolog User's Manual*, 4.1.1 edition, December 2009. Available from `http://www.sics.se/sicstus/`.

31 Terrance Swift and David Scott Warren. XSB: Extending Prolog with Tabled Logic Programming. *TPLP*, 12(1-2):157–187, 2012.

32 Asumu Takikawa, Daniel Feltey, Earl Dean, Matthew Flatt, Robert Bruce Findler, Sam Tobin-Hochstadt, and Matthias Felleisen. Towards practical gradual typing. In John Tang Boyland, editor, *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, volume 37 of *LIPIcs*, pages 4–27. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015. URL: `http://www.dagstuhl.de/dagpub/978-3-939897-86-6`, `doi:10.4230/LIPIcs.ECOOP.2015.4`.

33 H. Tamaki and M. Sato. OLD Resolution with Tabulation. In *Third International Conference on Logic Programming*, pages 84–98, London, 1986. Lecture Notes in Computer Science, Springer-Verlag.

34 Sam Tobin-Hochstadt and Matthias Felleisen. The Design and Implementation of Typed Scheme. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 395–406. ACM, 2008. `doi:10.1145/1328438.1328486`.

35 D.H.D. Warren. Higher-order extensions to Prolog: are they needed? In J.E. Hayes, Donald Michie, and Y-H. Pao, editors, *Machine Intelligence 10*, pages 441–454. Ellis Horwood Ltd., Chicester, England, 1982.

36 R. Warren, M. Hermenegildo, and S. K. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 684–699. MIT Press, August 1988.

37 J. Wielemaker. *The SWI-Prolog User's Manual 5.9.9*, 2010. Available from `http://www.swi-prolog.org`.