Types, modes and so much more – the Prolog way

Manuel V. Hermenegildo[1,2], Jose F. Morales[1,2],
Pedro Lopez-Garcia[2,3], and Manuel Carro[1,2]

[1] Universidad Politécnica de Madrid (UPM)
[2] IMDEA Software Institute
[3] Spanish Council for Scientific Research (CSIC)
{manuel.hermenegildo,josef.morales,pedro.lopez}@imdea.org

Abstract. We present in a tutorial way some ideas developed in the context of the Ciao Prolog system that we believe could be useful for the future evolution of Prolog. We concentrate primarily on one area: the use of assertions with types, modes, and other properties, and how the unique characteristics of Prolog have made early advances possible in the area of combining static and dynamic language features. However, we also address briefly some other issues related to extending the expressiveness and functionality of the language.

Keywords: Prolog, Static Languages, Dynamic Languages, Types, Modes, Assertions, Verification, Testing, Test Generation, Language Extensions.

1 Combining in Prolog the best of the dynamic and static language approaches

Prolog is a dynamically-typed language and this aspect, combined with the intrinsic power of the language, has arguably contributed to its continued relevance and use in many applications. In fact, the environment in which much software is developed nowadays, aligns well with the classical arguments for dynamic languages, and many of the currently most popular languages, such as Python, JavaScript, Ruby, etc. (with Scheme and Prolog also in this class) are dynamic.

At the same time, detecting errors as early as possible at compile time, and inferring properties required to optimize and parallelize programs are clearly important issues in real-world applications, and thus, strong arguments can also be made for static languages. For example, statically-typed logic and functional languages (such as, e.g., Mercury [37] or Haskell [17]) impose strong type-related requirements such as that all types (and, when relevant, modes) have to be defined explicitly or that all procedures have to be well-typed and well-moded. An important argument supporting this approach is that types clarify interfaces and meanings and facilitate programming in the large by making large programs more maintainable and better documented. Also, the compiler can use the static

* Partially funded by MICINN projects PID2019-108528RB-C21 *ProCode*, TED2021-132464B-I00 *PRODIGY*, and FJC2021-047102-I, by the Comunidad de Madrid program P2018/TCS-4339 *BLOQUES-CM*, and by the Tezos foundation. The authors would also like to thank the anonymous reviewers for very useful feedback.

information to generate more specialized code, which can be better in several ways (e.g., performance-wise).

In the design of Ciao Prolog we certainly had the latter arguments in mind, but we also wanted to retain the usefulness of standard Prolog for highly dynamic scenarios, programming in the small, prototyping, developing simple scripts, or simply for experimenting with the solution to a problem. We felt that strong typing and other related restrictions of statically-typed logic languages can sometimes get in the way in these contexts.

The solution we came up with –the *Ciao assertions model*– involves the combination of a rich assertion language, allowing a very general class of (possibly undecidable) properties, and a novel methodology for dealing with such assertions [3,13,30,29,14], based on making a best effort to infer and check assertions statically, using rigorous static analysis tools based on *safe approximations*, in particular via abstract interpretation [7]. This implies accepting that complete verification or error detection may not always be possible and run-time checks may be needed. This approach allows dealing in a uniform way with a wide variety of properties which includes types [33,41], but also, e.g., rich modes [25,24], determinacy [19], non-failure [9,4], sharing/aliasing, term linearity, cost [26,35,20], etc., while at the same time allowing assertions to be *optional*. The Ciao model and language design also allows for a smooth integration with testing [21]. Moreover, as (parts of) tests that can be verified at compile time are eliminated, some tests can be passed without ever running them. Finally, the model supports naturally assertion-based test case generation. In the following we illustrate these aspects of the model through examples run on the system.[1]

1.1 The assertions model in action

While there are several ways to use the system, we will show screenshots of one of the most convenient, which is to have the system running in the background giving instant feedback as a program is opened or edited –we refer to this as the "verifly" ("verification on the fly") mode (see [34] for more details).

A first example. Consider the classic implementation of quick-sort in Fig. 1. If no other information is provided, the exported predicate `qsort/2` can be called with arbitrarily instantiated terms as arguments (e.g., with a list of variables). This implies that the library predicates `=</2` and `>/2` in `partition/4` can also be called with arbitrary terms and thus run-time errors are possible, since `=</2` and `>/2` require their arguments to be bound to arithmetic expressions when called. Even though there are no assertions in the program itself, the system is able to warn that it cannot verify that the calls to `=</2` and `>/2` will not generate a run-time error (note » symbol and code underlining in orange). This is the result of a modular global analysis and a comparison of the information inferred for the program points before the calls to `=</2` and `>/2` with the assertions that express the calling restrictions for `=</2` and `>/2`. Such assertions live in the libraries that provide these standard predicates. Further details can be obtained by hovering over the literal (Fig. 2).

[1] The examples are runnable in the Ciao playground ▶ ; they have been developed with version 1.22 of the system. Screenshots are from the Ciao Prolog Emacs interface.

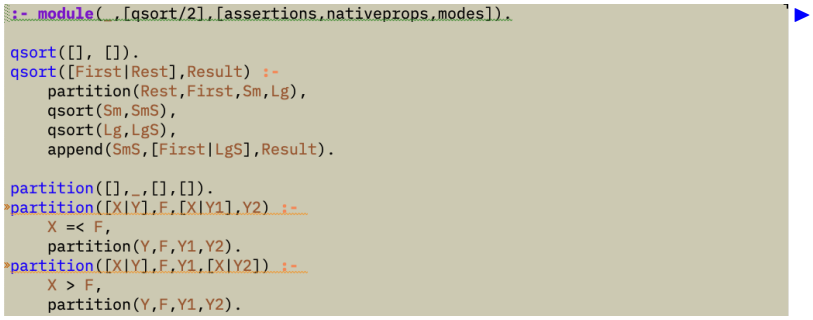

Fig. 1. With no entry information, the system warns that it cannot verify that the call to `=</2` will not generate a run-time error.

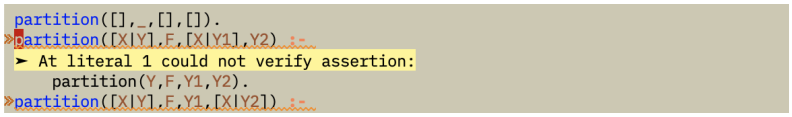

Fig. 2. Hovering over the clause the system shows a popup saying that it cannot verify the assertions for `=</2` (present in the library!).

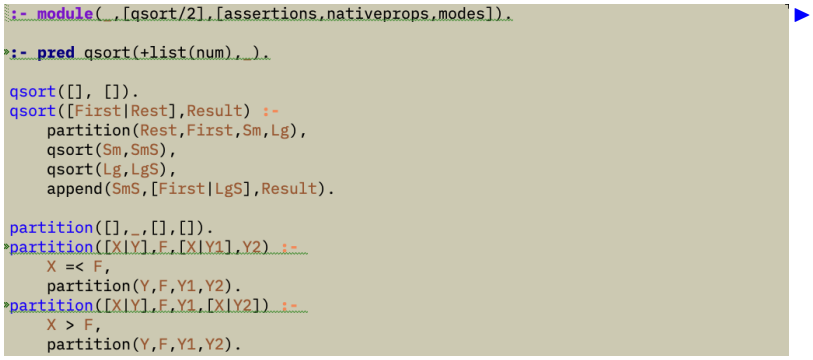

Fig. 3. Adding information on how the exported predicate should be called the system can infer that `=</2` will be called correctly, and no warnings are flagged.

In Fig. 3 we have added an assertion for the exported predicate `qsort/2` expressing that it should be called with its first argument bound to a list of numbers.[2] Assuming this "entry" information, the system can verify that all the calls to `=</2` and `>/2` are now correct (with their arguments bound to numbers in this case), and thus no warnings are flagged. Note that in practice this assertion may not be necessary since this information could be obtained from the analysis of the caller(s) to this module.

Let us now add more assertions to the program, stating properties that we want checked, as shown in Fig. 4. The assertion for predicate `partition/4` (eighth line of Fig. 4) expresses, using modes,[3] that the first argument should be bound to a list of numbers, and the second to a number, and that, for any termi-

[2] Due to space limitations we present the assertion language through –hopefully intuitive– examples. More complete descriptions of the assertion language can be found in[12,29,2].

[3] See, e.g., [43] in this same volume for an introduction to modes.

```
:- pred qsort(+list(num),-list(num)) + semidet.

qsort([], []).
qsort([First|Rest],Result) :-
    partition(Rest,First,Sm,Lg),
    qsort(Sm,SmS),
    qsort(Lg,LgS),
    append(SmS,[First|LgS],Result).

:- pred partition(+list(num),+num,-list(num),-list(num)) + det.

partition([],_,[],[]).
partition([X|Y],F,[X|Y1],Y2) :-
    X =< F,
    partition(Y,F,Y1,Y2).
partition([X|Y],F,Y1,[X|Y2]) :-
    X > F,
    partition(Y,F,Y1,Y2).
```

Fig. 4. We add more assertions expressing various properties.

```
:- pred qsort(+list(num),-list(num)) + semidet.

qsort([], []).
qsort([First|Rest],Result) :-
    partition(Rest,First,Sm,Lg),
    qsort(Sm,SmS),
    qsort(Lg,LgS),
    append(SmS,[First|LgS],Result).

:- pred partition(+list(num),+num,-list(num),-list(num)) + det.

partition([],_,[],[]).
partition([X|Y],F,[X|Y1],Y2) :-
    X =< F,
    partition(Y,F,Y1,Y2).
partition([X|Y],F,Y1,[X|Y2]) :-
    X > F,
    partition(Y,F,Y1,Y2).
```

Fig. 5. All the added assertions get verified by the system.

nating call meeting this call pattern: a) if the call succeeds, then the third and fourth arguments will be bound to lists of numbers; and b) the call is deterministic, i.e., it will produce one solution exactly, property `det` in the + field (as in Mercury [37]), which is inferred in CiaoPP as the conjunction of two properties: 1) the call does not (finitely) fail (property `not_fails` as in [9,4]) and 2) the call will produce one solution at most (property `is_det` as in [19]). Similarly, the assertion for `qsort/2` expresses the expected calling pattern, and that the call can have at most one answer, property `semidet`.

In the assertion model, modes are *macros* that serve as a shorthand for assertions, in particular *predicate-level assertions*. These are in general of the form:

`:- [` *Status* `]` `pred` *Head* `[:` *Pre* `]` `[=>` *Post* `]` `[+` *Comp* `].`

where *Head* denotes the predicate that the assertion applies to, and *Pre* and *Post* are conjunctions of *state property* literals. *Pre* expresses properties that hold when *Head* is called. *Post* states properties that hold if *Head* is called in a state compatible with *Pre* and the call succeeds. *Comp* describes properties of the whole computation such as determinism, non-failure, resource usage, termination, etc., aso for calls that meet *Pre*. In particular, the modes for `qsort/2` in Fig. 4 are expanded by the `modes` package (see module declaration in Fig. 3) to:

```
:- pred qsort(X,Y) : list(num,X) => list(num,Y) + semidet.
```

All the assertions in Fig. 4 indeed get verified by the system, which is shown by underlying the assertions in green (Fig. 5), and again further information can

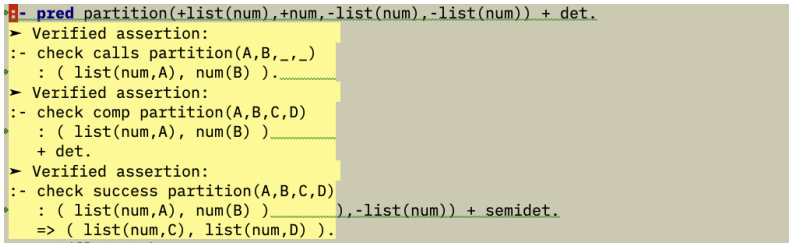

Fig. 6. The popup shows that calls, computational/global properties, and success pattern for `partition/` are verified.

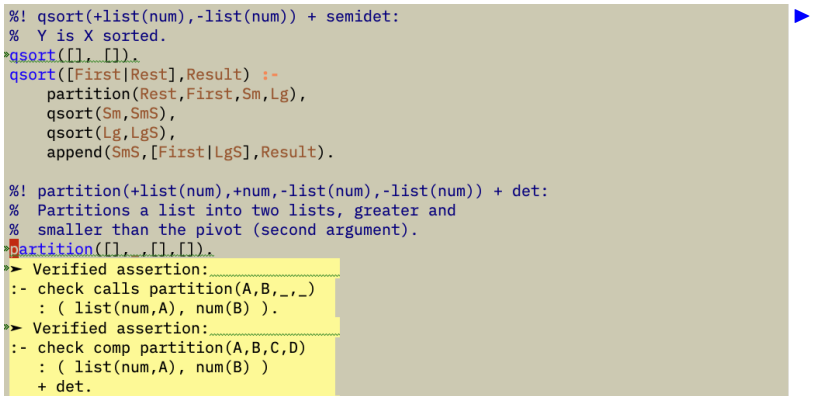

Fig. 7. Using modes/assertions in *doccomments* syntax (which are also verified).

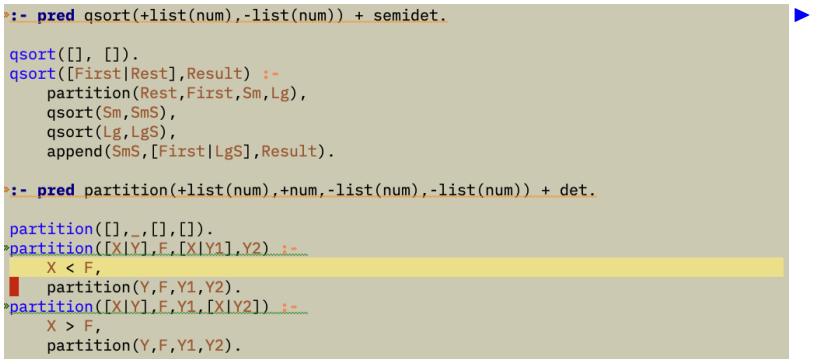

Fig. 8. If we replace `=</2` with `</2` the system warns that `partition/4` may fail.

be obtained in a popup (Fig. 6).[4] Fig. 7, shows again `qsort/2` but now the assertions are written as machine readable comments enabled by the `doccomments` package. Such comments can contain embedded assertions, which are also verified. Here we use again modes and determinacy. This format is familiar to Prolog programmers and compatible with any Prolog system without having to define any operators for the assertion syntax.

[4] Note that while, as mentioned before, the assertions in Fig. 4 use *modes* they are represented internally in normal form and the popup message uses syntax close to this form, where the computational properties and the state properties that must hold upon success are split into separate (`comp` and `success` assertions respectively).

```
:- pred qsort(+list(num),-list(num)) + semidet.

qsort([], []).
qsort([First|Rest],Result) :-
    partition(Rest,First,Sm,Lg),
    qsort(Sm,SmS),
    qsort(Lg,LgS),
    append(SmS,[First|LgS],Result).

:- pred partition(+list(num),+num,-list(num),-list(num)) + det.

partition([],_,[],[]).
partition([X|Y],F,[X|Y1],Y2) :-
    X =< F,
    partition(Y,F,Y1,Y2).
partition([X|Y],F,Y1,[X|Y2]) :-
    X >= F,
    partition(Y,F,Y1,Y2).
```

Fig. 9. If we replace `>/2` with `>=/2` the system warns that both `partition/4` and `qsosrt/2` may not be deterministic.

```
:- regtype color/1.
color(red).
color(green).
color(blue).

:- regtype colorlist/1.
colorlist([]).
colorlist([H|T]) :- color(H), colorlist(T).

:- prop sorted/1.
sorted([]).
sorted([_]).
sorted([X,Y|T]) :- X .>. Y, sorted([Y|T]).
```

Fig. 10. Defining some properties which can then be used in assertions.

```
:- regtype color/1.
color := red | green | blue.

:- regtype colorlist/1.
colorlist := [] | [~color|~colorlist].

:- prop sorted/1.
sorted := [] | [_].
sorted([X,Y|T]) :- X .>. Y, sorted([Y|T]).
```

Fig. 11. The properties of Fig. 10 written in functional notation.

In Fig. 8, we have replaced `=</2` with `</2` in the second clause of `partition/4`, and the system warns that this predicate may fail. This is because the case where `X=F` is not "covered" by the "tests" of `partition/4` [9,4]. Conversely, if we replace `>/2` with `>=/2` in the second clause of the original definition of `partition/4`, Fig. 9, the system warns that the predicate may not be deterministic. This is because the analyzer infers that not all the clauses of `partition/4` are pairwise mutually exclusive (in particular the second and third clauses are not), and thus, multiple solutions may be obtained [19].

Defining properties. The reader may be wondering at this point where the properties that are used in assertions (such as `list(num)`) come from. As mentioned before, such properties are typically written in Prolog and its extensions; and they can also be built-in and/or defined and imported from system libraries or in user code. Visibility is controlled by the module system as for any other predicate. Fig. 10 shows some examples of definitions of properties. Two of them are marked as *regular types* (`regtype` directive): `color/1`, defined as the set of values {`red`, `green`, `blue`}, and `colorlist/1`, representing the infinite set of lists whose elements are of `color` type. The third property is not a regular

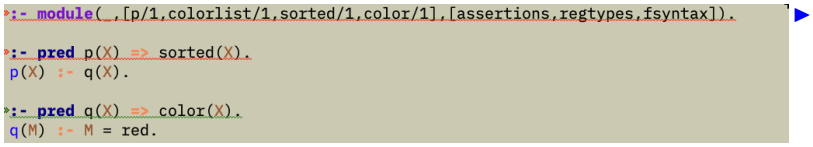

Fig. 12. An error is flagged in the success of `p/1`.

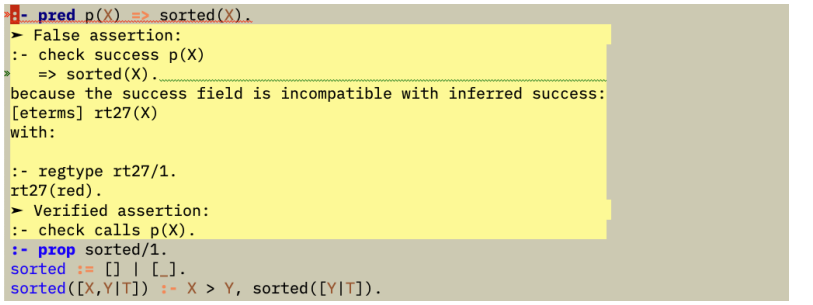

Fig. 13. Success and inferred properties (`sorted/1` and `red`) are incompatible.

Fig. 14. New definition of predicate `q/1` (and change in assertion).

type, but an arbitrary property (`prop` directive), representing the infinite set of lists of numeric elements in descending order. Marking predicates as properties allows them to be used in assertions, but they remain regular predicates, and can be called as any other, and also used as run-time tests, to generate examples (test cases), etc. For example:

```
?- colorlist(X).
X = [] ? ;
X = [red] ? ;
X = [red,red] ? ...
```

or, if we select breadth-first execution (useful here for fair generation):

```
?- colorlist(X).
X = [] ? ;
X = [red] ? ;
X = [green] ? ;
X = [blue] ? ;
X = [red,red] ? ...
```

Fig. 11 shows the same properties of Fig. 10 but written using functional notation. The definitions are equivalent, functional syntax being just syntactic sugar.

In Fig. 12 we add some simple definitions for `p/1` and `q/1`, and a `pred` assertion for `q/1`, meaning "in all calls `q(X)` that succeed, `X` is *instantiated* on success to a term of `color` type." This is verified by the system. We have also added an assertion for `p/1` meaning "in all calls `p(X)` that succeed, `X` gets instantiated to a term meeting the `sorted` property." The system detects that such assertion is false and shows the reason (Fig. 13): the analyzer (with the `eterms` abstract domain [41]) infers that on success `X` gets bound to `red`, expressed as the automatically inferred regular type `rt27/1`, and the system finds that `rt27(X)` and `sorted(X)` are incompatible (empty intersection of the set of terms they represent). In Fig. 14, we have changed the definition of `q/1` so that there is no

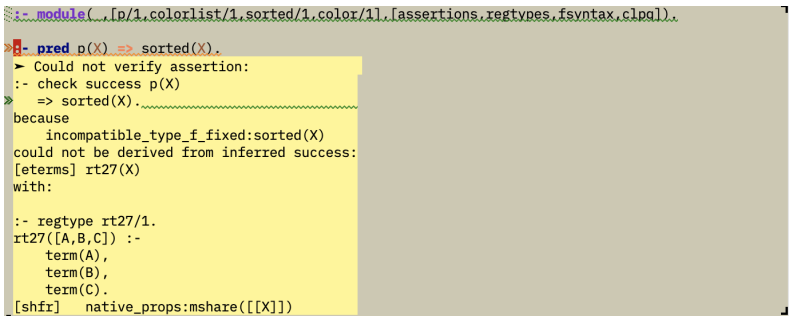

Fig. 15. Just a warning: `sorted` could not be verified (with selected domains).

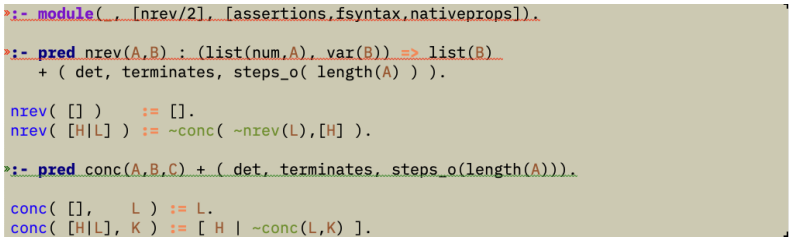

Fig. 16. An example with more complex properties, a cost error is flagged.

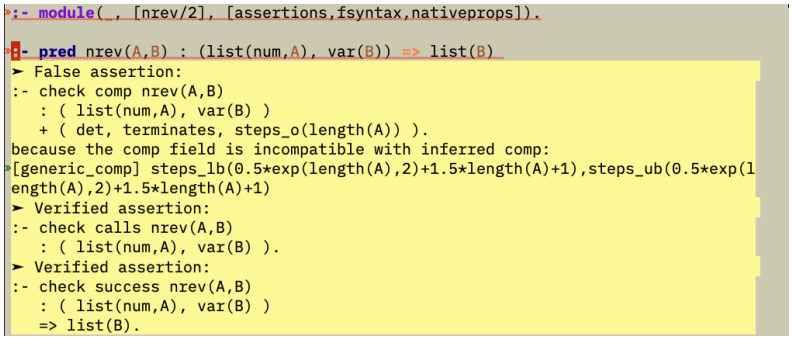

Fig. 17. The system reminds us that `nrev/2` is of course quadratic, not linear.

incompatibility, and now the system simply warns (Fig. 15) that it cannot verify the assertion for `p/1`. The success type `rt27(X)` inferred for `p/1` (lists of three arbitrary terms) and `sorted(X)` are now compatible, and thus no error is flagged. However, `rt27(X)` does not imply `sorted(X)` for all `X`'s, and thus `sorted(X)` is not verified (with the default set of abstract domains). In this case the system will (optionally) introduce a run-time check so that `sorted(X)` is tested when `p/1` is called. Furthermore, the system can run unit tests or generate test cases (in this case arbitrary terms) automatically to exercise such run-time tests.

An example with more complex properties (also using the functional syntax package) is shown in Fig. 16. It includes a user-provided assertion stating (among other properties) that the cost of `nrev/2` in resolution steps, for calls to `nrev(A, B)` with `A` a ground list and `B` a free variable, should be linear in the length of the (input) argument `A` ($O($`length(A)`$)$), property `steps_o(length(A))` in the `+` field. The system can infer that this is false and underlines it in red.

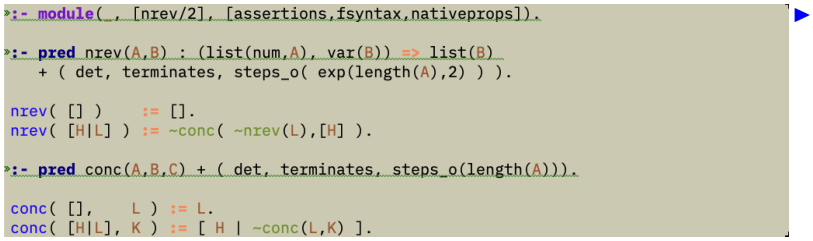

Fig. 18. With the cost expression fixed all properties are now verified.

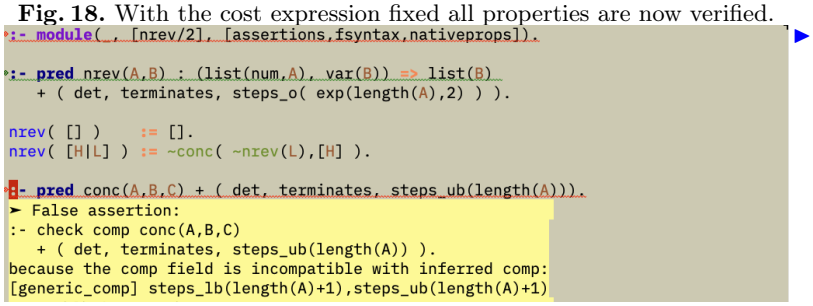

Fig. 19. If we change the assertion for `conc/3` from complexity order (`_o`) to upper bound (`_ub`) then the system flags that `length(A)` is not a correct upper bound.

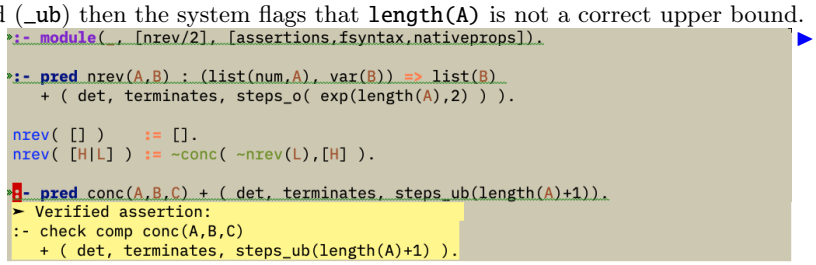

Fig. 20. With the cost expression fixed all properties are now verified.

The popup, Fig. 17, explains that the stated worst case asymptotic complexity is incompatible with the quadratic lower bound cost inferred by the analyzer (in fact: $\frac{1}{2}\ length(A)^2 + \frac{3}{2}\ length(A) + 1$, see the `steps_lb` property). If we change the assertion to specify a quadratic upper bound, it is now proven,[5] see Fig. 18 which also shows verification of the assertion for predicate `conc/3` and determinacy and termination properties. In Fig. 19, we have changed the assertion for `conc/3` from complexity order (`_o`) to a concrete upper bound (`_ub`), and the system detects the error: `length(A)` is not a correct upper bound because, as shown in the popup, it is incompatible with the lower bound `length(A) + 1` inferred by the analyzer [8,35]. Fig. 20 shows that if we change the upper bound to `length(A) + 1`, then the assertion is verified.

1.2 Discussion

We argue that this assertion model greatly enhances the power of Prolog for programming both in the small and in the large, combining effectively the advantages of the of dynamically- and statically-typed languages. It preserves the dynamic language features while at the same time providing safety guarantees and

[5] An upper bound [35,26] is also inferred, equal to the lower bound (Fig. 17).

the capability of achieving the performance and efficiency of static systems [6]. The novel combination of assertion language, properties, run-time checking, testing, etc. generates many new synergies.

We believe that a good part of the power of the approach (and perhaps why this approach was first proposed in the context of Prolog) arises from characteristics of the logic programming paradigm and the Prolog language in particular. For example, as we have seen, the fact that Prolog allows writing many properties (including types) in the source language is instrumental in allowing assertions which cannot be statically verified to be easily used as run-time checks, allowing users to obtain benefits even if a certain property cannot be verified at compile time. As another example, the reversibility of properties written in Prolog allows generating test cases automatically from assertions, without having to invent new concepts or to implement any new functionality, since "property-based testing" comes for free in this approach and thus did not need to be invented. Another contributing factor is that it was in the Prolog community that formal static analysis techniques, in particular abstract interpretation, flourished first, during the 80's and 90's [10], leading quite naturally to the development in the mid-90's of the Ciao model.

The practical relevance of the combination of static and dynamic features brought about by this approach is illustrated by the many other languages and frameworks which have been proposed more recently, aiming at bringing together both worlds, using similar ideas. This includes, e.g., the work on *gradual typing* [36,31,40] and *liquid types* [32,42]. Pfenning's et al.'s early work on *refinement types* [28] and *practical dependent types* [44] was seminal in this context and also uses abstract interpretation or constraint solving, but stays on the decidable side and is thus not combined with run-time checking or testing. Another example is the recent work on verifying *contracts* [39,18,23,27]. Prolog pioneered and is continuing to push the state of the art in this area. However, although some Prolog systems have introduced run-time checks or testing, there is still much work in this area that could become more widely adopted.

2 Making Prolog even more extensible, to support multiple features in a modular way

The future evolution of Prolog should arguably seek **increasing the power and expressiveness of the language** and its tools to make it even simpler to solve progressively more complex problems. This means continuing in the path exemplified by the addition of, e.g., constraints, concurrency/parallelism, tabling [38], assertions (as discussed previously), or (to name a more recent addition) s(CASP) [1,11]. As also advocated by Gupta et al. [11], it is also desirable to have systems that support all these and additional future extensions within the same implementation.

However, the syntactic and semantic elegance and simplicity of Prolog contrasts with (or may perhaps be thanks to) the implementation sophistication of

state-of-the-art Prolog systems, and this can potentially complicate the task of incorporating new functionality to the language.

Fortunately, many good ideas have progressively allowed making extensions in less painful ways. For example, attributed variables, pioneered by Holzbaur and Neumerkel in SICStus [16], made it much easier to add constraint systems to standard Prologs, and in a largely portable way.

Ciao Prolog introduced **new mechanisms for language extension**, such as more principled and modular versions of the *term expansion* facilities,[6] special features in the module system, and the notion of *packages*, which offer a clean distinction between compile-time and runtime extensions [5]. This is essential for global analysis (necessary for the assertion model and optimization), separate/incremental compilation, and language bootstrapping –in fact, most of Ciao, including its abstract machine, is defined in Prolog [22]. These ideas have allowed building the complete system starting from a small kernel in a layered way into a **multiparadigm language**, while having all built-ins and extensions (*constraints, different search rules, functions, higher-order, predicate abstractions, lazyness, concurrency, s(CASP),* etc.) as optional features that can be activated, deactivated, or combined on a per module basis. Even if for efficiency some such predicates (including for example the cut) may be implemented internally and supported natively in the virtual machine and compiler, none of them are considered *builtins* and their visibility can be controlled, including for example choosing to not load any impure ones, or to redefine them. This modular design allows moving from *pure LP*, where, e.g., no impure builtins are visible, to *full ISO Prolog* by specifying the set of imported modules, and going well beyond that into a multi-paradigm language, while maintaining full backwards compatibility with standard Prolog. Being able to travel these paths is also very useful in an educational context (see, for example, [15] also in this volume).

We believe that future systems should build further on these extensibility-oriented ideas and that the advocated *modularity* and *separation of concerns* are fundamental to Prolog's future evolution. Key features here are *advanced module systems* and the technology to *bridge the gap between the dynamic and static approaches*. They can facilitate **adding more declarative features and more advanced reasoning capabilities to Prolog**, while **providing guarantees** and **increasing performance**. This is specially relevant in a world where programs can be generated by learning systems and need to be modified and verified before use, and where they run on multi-core and heterogeneous computing devices, with complex specialized data representations to make optimal usage of the memory hierarchy. This can greatly benefit from more declarative program specifications (e.g., Prolog programs) and establishing a "dialogue" between programmers and the compiler (e.g., via the assertion language).

[6] See again [43] for in introduction to term expansion in Prolog.

References

1. Arias, J., Carro, M., Salazar, E., Marple, K., Gupta, G.: Constraint Answer Set Programming without Grounding. Theory and Practice of Logic Programming **18**(3-4), 337–354 (2018). https://doi.org/10.1017/S1471068418000285
2. Bueno, F., Carro, M., Hermenegildo, M.V., Lopez-Garcia, P., (Eds.), J.F.M.: The Ciao System. Ref. Manual (v1.22). Tech. rep. (April 2023), available at http://ciao-lang.org
3. Bueno, F., Deransart, P., Drabent, W., Ferrand, G., Hermenegildo, M.V., Maluszynski, J., Puebla, G.: On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In: Proc. of the 3rd Int'l. WS on Automated Debugging–AADEBUG. pp. 155–170. U. Linköping Press (May 1997)
4. Bueno, F., Lopez-Garcia, P., Hermenegildo, M.V.: Multivariant Non-Failure Analysis via Standard Abstract Interpretation. In: FLOPS'04. pp. 100–116. No. 2998 in LNCS, Springer-Verlag (2004)
5. Cabeza, D., Hermenegildo, M.V.: A New Module System for Prolog. In: International Conference on Computational Logic, CL2000. pp. 131–148. No. 1861 in LNAI, Springer-Verlag (July 2000)
6. Carro, M., Morales, J., Muller, H., Puebla, G., Hermenegildo, M.V.: High-Level Languages for Small Devices: A Case Study. In: Flautner, K., Kim, T. (eds.) Compilers, Architecture, and Synthesis for Embedded Systems. pp. 271–281. ACM Press / Sheridan (October 2006)
7. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: ACM Symposium on Principles of Programming Languages (POPL'77). pp. 238–252. ACM Press (1977). https://doi.org/10.1145/512950.512973
8. Debray, S.K., Lopez-Garcia, P., Hermenegildo, M.V., Lin, N.W.: Lower Bound Cost Estimation for Logic Programs. In: ILPS'97. pp. 291–305. MIT Press (1997)
9. Debray, S., Lopez-Garcia, P., Hermenegildo, M.V.: Non-Failure Analysis for Logic Programs. In: ICLP'97. pp. 48–62. MIT Press (1997)
10. Giacobazzi, R., Ranzato, F.: History of abstract interpretation. IEEE Ann. Hist. Comput. **44**(2), 33–43 (2022), https://doi.org/10.1109/MAHC.2021.3133136
11. Gupta, G., Salazar, E., Arias, J., Basu, K., Varanasi, S., Carro, M.: Prolog: Past, Present, and Future. In: Warren, D.S., Dahl, V., Eiter, T., Hermenegildo, M., Kowalski, R., Rossi, F. (eds.) Prolog - The Next 50 Years. No. 13900 in LNCS, Springer (July 2023)
12. Hermenegildo, M.V., Bueno, F., Carro, M., Lopez-Garcia, P., Mera, E., Morales, J., Puebla, G.: An Overview of Ciao and its Design Philosophy. Theory and Practice of Logic Programming **12**(1–2), 219–252 (January 2012). https://doi.org/10.1017/S1471068411000457, http://arxiv.org/abs/1102.5497
13. Hermenegildo, M.V., Puebla, G., Bueno, F.: Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In: The Logic Programming Paradigm: a 25–Year Perspective, pp. 161–192. Springer-Verlag (1999)
14. Hermenegildo, M.V., Puebla, G., Bueno, F., Lopez-Garcia, P.: Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). Science of Computer Programming **58**(1–2), 115–140 (October 2005). https://doi.org/10.1016/j.scico.2005.02.006
15. Hermenegildo, M., Morales, J., Lopez-Garcia, P.: Some Thoughts on How to Teach Prolog. In: Warren, D.S., Dahl, V., Eiter, T., Hermenegildo, M., Kowalski, R.,

Rossi, F. (eds.) Prolog - The Next 50 Years. No. 13900 in LNCS, Springer (July 2023), http://cliplab.org/papers/TeachingProlog-PrologBook.pdf
16. Holzbaur, C.: Metastructures vs. Attributed Variables in the Context of Extensible Unification. In: Int'l. Symposium on Programming Language Implementation and Logic Programming. pp. 260–268. No. 631 in LNCS, Springer Verlag (Aug 1992)
17. Hudak, P., Peyton-Jones, S., Wadler, P., Boutel, B., Fairbairn, J., Fasel, J., Guzman, M.M., Hammond, K., Hughes, J., Johnsson, T., Kieburtz, D., Nikhil, R., Partain, W., Peterson, J.: Report on the Programming Language Haskell. Haskell Special Issue, ACM Sigplan Notices **27**(5), 1–164 (1992)
18. Logozzo et al., F.: Clousot. http://msdn.microsoft.com/en-us/devlabs/dd491992.aspx (Accessed: 2018)
19. Lopez-Garcia, P., Bueno, F., Hermenegildo, M.V.: Automatic Inference of Determinacy and Mutual Exclusion for Logic Programs Using Mode and Type Analyses. New Generation Computing **28**(2), 117–206 (2010)
20. Lopez-Garcia, P., Klemen, M., Liqat, U., Hermenegildo, M.V.: A General Framework for Static Profiling of Parametric Resource Usage. TPLP (ICLP'16 Special Issue) **16**(5-6), 849–865 (2016). https://doi.org/10.1017/S1471068416000442
21. Mera, E., Lopez-Garcia, P., Hermenegildo, M.V.: Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework. In: 25th Int'l. Conference on Logic Programming (ICLP'09). LNCS, vol. 5649, pp. 281–295. Springer-Verlag (July 2009)
22. Morales, J., Carro, M., Hermenegildo, M.V.: Description and Optimization of Abstract Machines in a Dialect of Prolog. Theory and Practice of Logic Programming **16**(1), 1–58 (January 2016). https://doi.org/doi:10.1017/S1471068414000672
23. MSR: Code contracts. http://research.microsoft.com/en-us/projects/contracts/ (Accessed: 2018)
24. Muthukumar, K., Hermenegildo, M.: Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In: ICLP'91. pp. 49–63. MIT Press (June 1991)
25. Muthukumar, K., Hermenegildo, M.: Compile-time Derivation of Variable Dependency Using Abstract Interpretation. JLP **13**(2/3), 315–347 (July 1992)
26. Navas, J., Mera, E., Lopez-Garcia, P., Hermenegildo, M.: User-Definable Resource Bounds Analysis for Logic Programs. In: Proc. of ICLP'07. LNCS, vol. 4670, pp. 348–363. Springer (2007). https://doi.org/10.1007/978-3-540-74610-2_24
27. Nguyen, P.C., Tobin-Hochstadt, S., Van Horn, D.: Soft Contract Verification. In: Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming. pp. 139–152. ICFP '14, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2628136.2628156
28. Pfenning, F.: Dependent types in logic programming. In: Pfenning, F. (ed.) Types in Logic Programming, pp. 285–311. The MIT Press (1992)
29. Puebla, G., Bueno, F., Hermenegildo, M.V.: An Assertion Language for Constraint Logic Programs. In: Analysis and Visualization Tools for Constraint Programming, pp. 23–61. No. 1870 in LNCS, Springer-Verlag (2000). https://doi.org/10.1007/10722311_2
30. Puebla, G., Bueno, F., Hermenegildo, M.V.: Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs. In: Logic-based Program Synthesis and Transformation (LOPSTR'99). pp. 273–292. No. 1817 in LNCS, Springer-Verlag (March 2000). https://doi.org/10.1007/10720327_16
31. Rastogi, A., Swamy, N., Fournet, C., Bierman, G., Vekris, P.: Safe & Efficient Gradual Typing for TypeScript. In: 42nd POPL. pp. 167–180. ACM (January 2015)

32. Rondon, P.M., Kawaguchi, M., Jhala, R.: Liquid types. In: Gupta, R., Amarasinghe, S.P. (eds.) Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008. pp. 159–169. ACM (2008). https://doi.org/10.1145/1375581.1375602, https://doi.org/10.1145/1375581.1375602
33. Saglam, H., Gallagher, J.: Approximating Constraint Logic Programs Using Polymorphic Types and Regular Descriptions. Technical Report CSTR-95-17, Dep. of Computer Science, U. of Bristol, Bristol BS8 1TR (1995)
34. Sanchez-Ordaz, M., Garcia-Contreras, I., Perez-Carrasco, V., Morales, J.F., Lopez-Garcia, P., Hermenegildo, M.V.: Verifly: On-the-fly Assertion Checking via Incrementality. Theory and Practice of Logic Programming **21**(6), 768–784 (September 2021). https://doi.org/10.1017/S1471068421000430, http://arxiv.org/abs/2106.07045, special Issue on ICLP'21
35. Serrano, A., Lopez-Garcia, P., Hermenegildo, M.V.: Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types. TPLP, ICLP'14 Special Issue **14**(4-5), 739–754 (2014). https://doi.org/10.1017/S147106841400057X
36. Siek, J.G., Taha, W.: Gradual Typing for Functional Languages. In: Scheme and Functional Programming Workshop. pp. 81–92 (2006)
37. Somogyi, Z., Henderson, F., Conway, T.: The Execution Algorithm of Mercury: an Efficient Purely Declarative Logic Programming Language. JLP **29**(1–3), 17–64 (October 1996)
38. Swift, T., Warren, D.S.: XSB: Extending Prolog with Tabled Logic Programming. Theory and Practice of Logic Programming **12**(1-2), 157–187 (Jan 2012). https://doi.org/10.1017/S1471068411000500
39. Takikawa, A., Feltey, D., Dean, E., Flatt, M., Findler, R.B., Tobin-Hochstadt, S., Felleisen, M.: Towards Practical Gradual Typing. In: Boyland, J.T. (ed.) 29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic. LIPIcs, vol. 37, pp. 4–27. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2015). https://doi.org/10.4230/LIPIcs.ECOOP.2015.4, http://dx.doi.org/10.4230/LIPIcs.ECOOP.2015.4
40. Tobin-Hochstadt, S., Felleisen, M.: The Design and Implementation of Typed Scheme. In: POPL. pp. 395–406. ACM (2008)
41. Vaucheret, C., Bueno, F.: More Precise yet Efficient Type Inference for Logic Programs. In: SAS'02. pp. 102–116. No. 2477 in LNCS, Springer (2002)
42. Vazou, N., Tanter, É., Horn, D.V.: Gradual liquid type inference. Proc. ACM Program. Lang. **2**(OOPSLA), 132:1–132:25 (2018). https://doi.org/10.1145/3276502, https://doi.org/10.1145/3276502
43. Warren, D.S.: Introduction to Prolog. In: Warren, D.S., Dahl, V., Eiter, T., Hermenegildo, M., Kowalski, R., Rossi, F. (eds.) Prolog - The Next 50 Years. No. 13900 in LNCS, Springer (July 2023)
44. Xi, H., Pfenning, F.: Dependent types in practical programming. In: Appel, A.W., Aiken, A. (eds.) POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999. pp. 214–227. ACM (1999). https://doi.org/10.1145/292540.292560, https://doi.org/10.1145/292540.292560