The Ciao Approach to the Dynamic vs. Static Language Dilemma

(Position/System/Demo Paper[1])

M. V. Hermenegildo[1,2] F. Bueno[1] M. Carro[1] P. López-García[2,4]
E. Mera[3] J. F. Morales[2] G. Puebla[1]

[1]Universidad Politécnica de Madrid (UPM)
{bueno,mcarro,german,herme}@fi.upm.es
[2]Madrid Institute of Advanced Studies
in SW Development Technology (IMDEA Software Institute)
{manuel.hermenegildo,pedro.lopez,jose.morales}@imdea.org
[3]Universidad Complutense de Madrid (UCM)
edison@fdi.ucm.es
[4]Scientific Research Council (CSIC)

Dynamic vs. Static Languages

The environment in which much software needs to be developed nowadays (decoupled software development, use of components and services, increased interoperability constraints, need for dynamic update or self-reconfiguration, mash-up development, etc.) is posing requirements which align with the classical arguments for dynamic languages and which in fact go beyond them. Examples of often required dynamic features include making it possible to (partially) test and verify applications which are partially developed and which will never be "complete" or "final," or which evolve over time in an asynchronous, decentralized fashion (e.g., software service-based systems). These requirements, coupled with the intrinsic agility in development of dynamic programming languages such as Python, Ruby, Lua, JavaScript, Perl, PHP, etc. (with Scheme or Prolog also in this class) have made such languages a very attractive option for a number of purposes that go well beyond simple scripting. Parts written in these languages often become essential components (or even the whole implementation) of full, mainstream applications.

At the same time, detecting errors at compile-time and inferring many properties required in order to optimize programs are still important issues in real-world applications. Thus, strong arguments are still also made in favor of static languages. For example, many modern logic and functional languages (such as, e.g., Mercury [24] or Haskell [12]) impose strong type-related requirements such as that all types (and, when relevant, modes) have to be defined explicitly or

[1] In addition to the other references, this recent tutorial overview of Ciao [11] covering more fully the points made in this position paper can be downloaded from:
http://clip.dia.fi.upm.es/papers/hermenegildo10:ciao-design-tplp-tr.pdf

that all procedures have to be "well-typed" and "well-moded." One argument supporting this approach is that it clarifies interfaces and meanings and facilitates "programming in the large" by making large programs more maintainable and better documented. Also, the compiler can use the static nature of the language to generate more specific code, which can be better in several ways (e.g., performance-wise).

The Ciao Approach

In the design of Ciao [7,6,2,10,11] we certainly had the latter arguments in mind, but we also wanted Ciao to be useful (as the "scripting" languages) for highly dynamic scenarios such as those listed above, for "programming in the small," for prototyping, for developing simple scripts, or simply for experimenting with the solution to a problem. We felt that compulsory type and mode declarations, and other related restrictions, can sometimes get in the way in these contexts. Ciao aims at combining the flexibility of dynamic/scripting languages with the guarantees of static languages, to bridge programming in the small and programming in the large, while performing efficiently on platforms ranging from small embedded processors to powerful multicore architectures.

Important components of the solution we came up with are the rich Ciao assertion language and the Ciao methodology for dealing with such assertions [3,8,22], which implies *making a best effort to infer and check properties statically, even highly complex ones, by using powerful and rigorous static analysis tools based on safe approximations, while accepting that complete verification may not always be possible (at least in a fully automated way) and run-time checks may sometimes be needed.* This approach opens up the possibility of dealing in a uniform way with a wide variety of properties besides traditional types (e.g., rich modes, determinacy, non-failure, shapes, sharing/aliasing, term linearity, time, memory, general resources,...), while at the same time allowing all assertions to be *optional*.

The Ciao assertion language provides a homogeneous framework which allows, among other things, static and dynamic verification (including unit testing [17]) to work cooperatively in a unified way. It is also instrumental for auto-documentation. The Ciao Preprocessor (CiaoPP) [3,8,21,9]) is a compile-time tool, based on abstract interpretation and other related techniques, which is capable of statically finding non-trivial bugs, verifying that the program complies with specifications (written in the assertion language), introducing run-time checks for (parts of) assertions that cannot be verified statically, and performing many types of program optimizations (including automatic parallelization). Such optimizations produce code that is highly competitive not only with other dynamic (or "scripting") languages but even that of static languages, when the optimizing compiler is used, all while retaining the interactive development environment of a dynamic language. This static/dynamic compilation architecture supports modularity and separate compilation throughout.

In the Ciao approach many properties used in assertions, including for example types, are written directly (or with convenient syntactic sugar) in the source language, so that they can be *run* and experimented with. I.e., one can test interactively if a certain data structure belongs to a type, has a particular size, or does not contain aliased pointers by just passing the data structure to the definition of the corresponding property and executing it. Furthermore, properties can often be used to enumerate (produce examples) of data which meet the property, such as, e.g., generating concrete examples of a type. This is all instrumental in the implementation of run-time checks and testing. The underlying logic engine and meta-programming capabilities of Ciao are fundamental in these tasks.

As mentioned above, the assertion language and preprocessor design also allows a smooth integration with unit testing. Unit tests are expressed as assertions. Then, as with other assertions, the (parts of) unit tests that can be verified at compile-time are eliminated, and sometimes it not not necessary whole sets of tests.

We argue that the solutions that were adopted in the Ciao design allow programming both in the small and in the large, combining effectively the advantages of the strongly typed and untyped language approaches. In contrast, systems which focus exclusively on automatic compile-time checking are often rather strict about the properties which the user can write. This is understandable because otherwise the underlying static analyses are of little use for proving the assertions.

Some Related Work

The Ciao model is related to the *soft typing* approach [4]. However, compile-time inference and checking in the Ciao model is not restricted to types (nor requires properties to be decidable), and it draws many new synergies from its novel combination of assertion language, properties, certification, run-time checking, testing, etc. The practical relevance of the combination of static and dynamic features is in fact illustrated by the many other languages and frameworks which have been proposed lately aiming at bringing together ideas of both worlds. This includes the very interesting recent work in gradual typing for Scheme [25] (and the related PLT-Scheme/Racket language), the recent uses of "contracts" in verification [16,19], and the pragmatic viewpoint of [14], but applied to programming languages rather than specification languages. The fifth edition of ECMAScript [5], on which the JavaScript and ActionScript languages are based, includes optional (soft-)type declarations to allow the compiler to generate more efficient code and detect more errors. The Tamarin project [18] intends to use this additional information to generate faster code. For Python, the PyPy project [23] has designed a language, RPython [1], that imposes constraints on the programs to ensure that they can be statically typed. RPython is moving forward as a general purpose language. This line of thought has also brought the development of safe versions of traditional languages, such as, e.g., CCured [20] or Cyclone [13] for C, as well as of systems that offer functionality similar to those of the Ciao assertion preprocessor, such as Deputy (`http://deputy.cs.berkeley.edu/`) or Spec# [15]. In summary, we argue that Ciao pioneered what are becoming relatively widely accepted approaches to marrying the static and dynamic language worlds.

Language Extensibility in Ciao

While not as directly related to the dynamic vs. static dilemma, another important characteristic of Ciao is that it is built up from a kernel that includes significant extensibility capabilities, i.e., it includes an easily programmable and modular way of defining new syntax and giving semantics to it in terms of that kernel language. This idea is not exclusive to Ciao, but in Ciao the facilities that enable building up from a simple kernel are extensive and explicitly available from the system programmer level to the application programmer level.

Also, this mechanism to add new syntax to the language and give semantics to such syntax can be activated or deactivated on a per-compilation unit basis without interfering with others. In fact, all Ciao operators, "builtins," and most other syntactic and semantic language constructs are user-modifiable and live in *libraries*. Using these facilities, Ciao provides the programmer with a large number of useful features from different programming paradigms and styles, and the use of each of these features can be turned on and off at will for each program module. Thus, a given module may be using, e.g., higher order functions and constraints, while another module may be using imperative operations, objects, predicates, Prolog meta-programming builtins, and concurrency.

Conclusions

We believe that Ciao has pushed and is continuing to push the state of the art in solving the currently very relevant and challenging conundrum between statically and dynamically checked languages. It pioneered what we believe is the most promising approach in order to be able to obtain the best of both worlds: the combination of a flexible, multi-purpose assertion language with strong program analysis technology. This allows support for dynamic language features while at the same time having the capability of achieving the performance and efficiency of static systems. We believe that a good part of the power of the Ciao approach also comes from the synergy that arises from using the same framework and assertion language for different tasks (static verification, run-time checking, unit testing, documentation, . . .) and its interaction with the design of Ciao itself (its module system, its extensibility, or the support for predicates and constraints).

References

1. Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. RPython: a Step towards Reconciling Dynamically and Statically Typed OO Languages. In *DLS '07: Proceedings of the 2007 Symposium on Dynamic Languages*, pages 53–64, New York, NY, USA, 2007. ACM.

2. F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla-(Eds.). The Ciao System. Ref. Manual (v1.13). Technical report, School of Computer Science, T.U. of Madrid (UPM), 2009. Available at `http://www.ciaohome.org`.
3. 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.
4. Robert Cartwright and Mike Fagan. Soft Typing. In *Programming Language Design and Implementation (PLDI 1991)*, pages 278–292. SIGPLAN, ACM, 1991.
5. ECMA International. ECMAScript Language Specification, Standard ECMA-262, Edition 5. Technical report, September 2009. Available at `http://wiki.ecmascript.org`.
6. M. Hermenegildo, F. Bueno, D. Cabeza, M. Carro, M. García de la Banda, P. López-García, and G. Puebla. The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. In *Parallelism and Implementation of Logic and Constraint Logic Programming*, pages 65–85. Nova Science, Commack, NY, USA, April 1999.
7. M. Hermenegildo and The CLIP Group. Some Methodological Issues in the Design of CIAO - A Generic, Parallel, Concurrent Constraint System. In *Principles and Practice of Constraint Programming*, number 874 in LNCS, pages 123–133. Springer-Verlag, May 1994.
8. 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.
9. M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2), 2005.
10. M. Hermenegildo and The Ciao Development Team. Why Ciao? –An Overview of the Ciao System's Design Philosophy. Technical Report CLIP7/2006.0, Technical University of Madrid (UPM), School of Computer Science, UPM, December 2006. Available from: `http://cliplab.org/papers/ciao-philosophy-note-tr.pdf`.
11. 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. Technical Report CLIP2/2010.0, Technical University of Madrid (UPM), School of Computer Science, March 2010. Under consideration for publication in *Theory and Practice of Logic Programming (TPLP)*.
12. P. Hudak, S. Peyton-Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzman, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Peterson. Report on the Programming Language Haskell. *Haskell Special Issue, ACM Sigplan Notices*, 27(5), 1992.
13. Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. In Carla Schlatter Ellis, editor, *USENIX Annual Technical Conference, General Track*, pages 275–288. USENIX, 2002.
14. Leslie Lamport and Lawrence C. Paulson. Should your specification language be typed? *ACM Transactions on Programming Languages and Systems*, 21(3):14, May 1999.

15. 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.
16. Francesco Logozzo et al. Clousot. `http://msdn.microsoft.com/en-us/devlabs/dd491992.aspx`.
17. E. Mera, P. López-García, and M. Hermenegildo. Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework. In *25th International Conference on Logic Programming (ICLP'09)*, number 5649 in LNCS, pages 281–295. Springer-Verlag, July 2009.
18. Mozilla. Tamarin Project, 2008. Available at `http://www.mozilla.org/projects/tamarin/`.
19. MSR. Code contracts. `http://research.microsoft.com/en-us/projects/contracts/`.
20. George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, 2005.
21. G. Puebla, F. Bueno, and M. Hermenegildo. A Generic Preprocessor for Program Validation and Debugging. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 63–107. Springer-Verlag, September 2000.
22. G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, September 2000.
23. A. Rigo and S. Pedroni. PyPy's Approach to Virtual Machine Construction. In *Dynamic Languages Symposium 2006*. ACM Press, October 2006.
24. Z. Somogyi, F. Henderson, and T. Conway. The Execution Algorithm of Mercury: an Efficient Purely Declarative Logic Programming Language. *Journal of Logic Programming*, 29(1–3):17–64, October 1996.
25. 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.