Types in Programming Languages, between Modelling, Abstraction, and Correctness

Extended Abstract

Simone Martini

Università di Bologna, Dipartimento di Informatica–Scienza e Ingegneria, Italy;
and INRIA Sophia-Antipolis, France

Keywords:
Types; Programming languages; History of computing; Abstraction mechanisms.

1 Introduction

The notion of *type* to designate a class of values, and the operations on those values, is a central feature of any modern programming language. In fact, we keep calling them *programming* languages, but the part of a modern language devoted to the actual specification of the control flow (that is, programming *stricto sensu*) is only a fraction of the language itself, and two different languages are not much apart under that perspective. What “makes a language” are much more its modelling capabilities to describe complex relations between portions of code and between data. In a word, the central part of a language is made by the abstraction mechanisms it provides to model its application domain(s), all issues the language theorist may well group together in the type chapter of a language definition.

The conquest of the summit by the notion of type is the result of a rather slow process in the history of programming languages. In a previous paper [21] we have sketched some of the earliest history, observing that the concept of type we understand nowadays is not the same it was perceived in the sixties, and that it was largely absent (as such) in the programming languages of the fifties. While the technical term “type” arrives on the scene at the end of the fifties (for sure in the report on Algol 58 [26])[1], the use of types as a modelling tool for the “objects of the real world” is the contribution of the sixties (in particular under the influence of McCarthy [22] and Hoare [15]), which will materialize

[1]The very first use of the term “type” in programming is probably H.B. Curry’s [7], to distinguish between memory words containing instructions (“*orders*”) and those containing data (“*quantities*”). These reports by Curry, as reconstructed by [10], contain a surprising, non-trivial mathematical theory of programs, up to a theorem analogous to the “well-typed expressions do not go wrong” of [23]! Despite G.W. Patterson’s review on JSL 22(01), 1957, 102-103, we do not know of any influence of this theory on subsequent developments of programming languages.

in languages like Algol W [30] or Pascal. Moreover, we observed in [21] that the notion of "type" of programming languages, which we now conflate with the concept of the same name of mathematical logic, is instead relatively independent from the logical tradition, until the Curry-Howard isomorphism [18] will make an explicit bridge between them. The connection between these two concepts remains anonymous for a long time—some of the people knew very well the other field, and it is certain that, from mid sixties, the mathematical logic work started influencing programming languages (we think, among other, to Landin, Scott, Strachey, Hoare, McCarthy, Morris etc.). But there is no explicit, mutual recognition—concepts and formal systems are systematically re-discovered in the two fields. The first explicit connection we know of, in a non technical, but explicit, way is [16].

The present paper will elaborate on this story, focusing on that fundamental period covering the seventies and the early eighties. It is there that the types become the cornerstone of the programming language design, passing first from the abstract data type (ADT) movement and blossoming then into the object-oriented paradigm. This will also be the occasion to reflect on how it could have been possible that a concept like ADTs, with its clear mathematical semantics, neat syntax, and straightforward implementation, could have given way to objects, a lot dirtier from any perspective the language theorist may take.

2 Modeling, and correctness

A central issue of the story we have told in [21] is the provision of a language mechanism to introduce new data types, in an extensible way (that is, different from a palette of types fixed at language design time). This materializes in the two related proposals of *records and typed references* [15], and Simula's *classes* [9]. A further, important realization is that a type is given not only by its class of values, but it is defined *together* the operations acting on those values (see, e.g.,[8][2]). Simula was an extension of Algol 60 designed for discrete event simulation. One of the main concepts in Simula is the *class*[3]—the specification of both data and operations, which may have several dynamic instances (*objects* in the modern terminology, or *processes* in Simula I), whose life is not required to be dynamically nested. Under this view, Simula classes are a good candidate as a language mechanism for the definition of such types, since they permit the simultaneous definition of data and operations. However, this has to be seen together with the need to enforce some level of correctness, at a syntactic (and if possible, static) way—a central feature of what Priestly [27] calls the "Algol research programme[4]", where the design of programming languages should

[2] "A type is a class of values. Associated with each type there are a number of operations which apply to such values".

[3] "Class" is, however, the terminology of Simula 67; in Simula I they are called *activities*.

[4] Under this term Priestly refers to the "coherent and comprehensive research programme within which the Algol 60 report had the status of a paradigmatic achieve-

assist (or even guide) the programmer in avoiding bugs or, worse, unintended behaviors in a program. While Simula classes may be used to define types as "data plus operations", they do not provide the necessary *abstraction* to enforce correctness, because the language does not distinguish between a type and its implementation (between, say, a stack, and the list used to implement that stack: a stack may be incautiously manipulated by any operation on lists). The need for this abstraction is the core of much literature on data and programming languages in the early seventies.

3 Abstract data types

The search for economic and terse linguistic constructs comes together with the need for the definition of a precise semantics for those constructs. Parnas' seminal [25] introduces the term *information hiding*, meaning that a stable interface towards the rest of the program should protect the design choices which are bound to change (which may thus evolve without affecting the other parts of the program). In Parnas' view this is a general design methodology, which applies to types, modules, packages, etc. It is a crucial forward step of the programming language community that this hiding should be enforced by linguistic abstraction mechanisms, and not merely guaranteed by a design methodology. Looking at the published literature (e.g., Morris [24], Hoare [17], Goguen [12], Liskov and Zilles [20], Reynolds [29]—who also explicitly introduces the expression "representation independence"[5]—, Guttag [14], etc.), we see that around 1972-1973 the time is ripe for a substantial achievement. If Hoare [17] uses an axiomatic settings, starting from Goguen [12] the (informal) abstraction requirement is described semantically by *freeness*—a data type is, in its mathematical semantics, a free algebra over a set of constructors (that is, non-interpreted function names). Freeness means that one cannot uses implementation dependent information on a value, because a values is simply an inductive construction over the constructors—hence abstraction. Moreover, this provides a powerful proof-technique on programs—structural induction [3,4]. Finally, equations on terms—and hence all the good properties of equational logic—provide the axiomatic semantics needed to distinguish between types (algebras) over isomorphic sets of constructors, but with different behavior (stacks and queues, say). The notion of abstract data type will make into programming languages with CLU [19], which will have a significant impact on subsequent languages.

ment, in the sense defined by the historian of science Thomas Kuhn. This research programme established the first theoretical framework for studying not only the design of programming languages, but also the process of software development." Therefore, are grouped under this broad term the developments of structured programming, of software engineering à la Dijkstra, of the formal description of programming language semantics, etc.

[5]A language provides representation independence if two correct implementations of a single specification of an ADT are observationally indistinguishable by the clients of these types.

At the end of the seventies, thus, it seems that types in programming languages are in a successful, and positive position—type abstraction mechanisms made into good (albeit essentially academic) programming languages, and their linguistic constructs come with a clear and mathematically sound formal semantics. However, in a sort of *coup de théâtre*, at the peak of their success ADTs will have to give way to another concept—objects, a much dirtier mechanism, able to enforce a lot less correctness than ADTs. Objects, and not ADTs will be the real players of the programming languages of the following decade(s).

4 Objects

To understand the fall of ADTs we must contrast correctness with flexibility. The utopia is to have them both at the same time, but realism tells us that they represent, most of the time, conflicting aims. ADTs provide abstraction at the expenses of (reuse and, then, of) compatibility. We cannot give here all the details of an example of the problem[6]. Suppose only to have an ADT T with an operation f, which is then extended into a new type $T1$, sharing the same values of T but with an extended set of operations and for which the operation f is also redefined. It is now natural (and convenient) to assume that the language enforces that $T1$ is compatible with T (a value of $T1$ may appear in any context requiring T). The problem now is that in an expression like $f(t)$ the choice of which code for f will be executed (the one for T or the one for $T1$) depends on the static context, that is, on the static type of t. Thus, if t (of static type T) references a value $t1$ of type $T1$ (which may correctly happen, in view of the compatibility of $T1$ with T), it is the erroneous f to be applied to a value of $t1$—abstraction breaks, which means that compatibility of $T1$ and T must be abandoned, which results in a drastic programming burden.

From this perspective the solution is easy—allow for a *dynamic* choice for the selection of the code for f in $f(t)$, depending not from the static typing of t, but from the actual type of the value referenced by t. In the programming language jargon, do not use functions and overloading, and use instead methods and dynamic lookup. Object oriented programming may be seen as the result of this observation. It is a paradigm where: (i) there is mechanism which, under certain conditions, supports the inheritance of the implementation of certain operations from other, analogous constructs; (ii) there is a notion of compatibility defined in terms of the operations admissible for a certain construct; (iii) operations on values are dynamically selected on the basis of the "actual type" of the arguments to which they are applied. These features *together* allow for the flexibility of the paradigm when used in actual programming of large scale systems. But, at the same time, these features *together* cannot be given semantics in the clean framework of algebraic types, at least in their simple formulation. Providing a sound semantics and a formal treatment of objects will be a challenge for almost fifteen years (see, e.g., [28,1,2] for the references therein). Finally, we will have to explain why types are so central for objects, in view of the *lack* of types in

[6]For a pedagogical discussion, see [11], Chapter 10.

languages like Smalltalk [13]. The distinction made in [6] (and the discussion in [5]) will guide our reflection.

5 Conclusions

The history of computer science is innervated by the continuous tension between formal beauty and technological effectiveness. Types in programming languages are an evident example of this dialectics. They are introduced for a better verification of the correctness of programs, and yet—contrary to mathematical logic—they must be experienced by the working programmer as an enabling feature[7], allowing for simpler writing of programs.

In its formal approach, computer science never used ideological glasses (types per se; constructive mathematics per se; linear logic per se; etc.), but exploited what it found useful for the design of more elegant, economical, usable artifacts. This eclecticism (or even anarchism, in the sense of epistemological theory) is one of the distinctive traits of the discipline, and one of the reasons of its success.

References

1. Martin Abadi and Luca Cardelli. A semantics of object types. In *Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 332–341, 1994.
2. Martin Abadi and Luca Cardelli. *A theory of objects*. Springer, 1996.
3. Rod Burstall. Proving properties of programs by structural induction. *The Computer Journal*, 12(1):41–48, 1969.
4. Rod Burstall and P. J. Landin. Programs and their proofs: an algebraic approach. *Machine Intelligence*, 4:17–43, 1969.
5. William R Cook. Object-oriented programming versus abstract data types. In *Foundations of Object-Oriented Languages*, pages 151–178. Springer, 1990.
6. William R. Cook, Walter Hill, and Peter S. Canning. Inheritance is not subtyping. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 125–135, New York, NY, USA, 1990. ACM.
7. Haskell B. Curry. On the composition of programs for automatic computing. Technical Report Memorandum 10337, Naval Ordnance Laboratory, 1949.
8. Ole-Johan Dahl and C. A. R. Hoare. Hierarchical program structures. In *Structured programming*, chapter 3, pages 175–220. Academic Press, 1972.
9. Ole-Johan Dahl and Kristen Nygaard. Simula: An ALGOL-based simulation language. *Commun. ACM*, 9(9):671–678, September 1966.
10. Liesbeth De Mol, Martin Carlé, and Maarten Bullyinck. Haskell before Haskell: an alternative lesson in practical logics of the ENIAC. *Journal of Logic and Computation*, 25(4):1011–1046, 2015.
11. Maurizio Gabbrielli and Simone Martini. *Programming Languages: Principles and Paradigms*. Undergraduate Topics in Computer Science. Springer, 2010.
12. Joseph Goguen. Some comments on data abstraction. Notes for a course at ETH Zurich, 1973.

[7] An expression of Vladimir Voevodsky.

13. Adele Goldberg and Alan Kay. *Smalltalk-72 instruction manual.* Technical Report SSL 76-6. Learning Research Group, Xerox Palo Alto Research Center, 1976.
14. John Guttag. *The specification and application to programming of Abstract Data Types.* PhD thesis, University of Toronto, 1975.
15. C. A. R. Hoare. Record handling. *ALGOL Bull.*, 21:39–69, November 1965.
16. C. A. R. Hoare. Notes on data structuring. In *Structured programming*, chapter 2, pages 83–174. Academic Press, 1972.
17. C. A. R. Hoare. Proof of correctness of data representation. *Acta Informatica*, pages 271–281, 1972.
18. William A. Howard. The formulae-as-types notion of construction. In Jonathan P. Seldin and J. Roger Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.
19. B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in CLU. *Commun. ACM*, 20(8):564–576, 1977.
20. Barbara Liskov and Stephen Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*, pages 50–59. ACM, 1972.
21. Simone Martini. Several types of types in programming languages. Paper presented at HAPOC 2015, Pisa, 2015.
22. John McCarthy. A basis for a mathematical theory of computation, preliminary report. In *Papers Presented at the May 9-11, 1961, Western Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '61 (Western), pages 225–238, New York, NY, USA, 1961. ACM.
23. Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.
24. James H. Morris. Types are not sets. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 120–124, New York, NY, USA, 1973. ACM.
25. D.L. Parnas. On the criteria to be used in decomposing systems into modules. *CACM*, 15(2):1053–58, 1972.
26. A. J. Perlis and K. Samelson. Preliminary report: International algebraic language. *Commun. ACM*, 1(12):8–22, December 1958.
27. Mark Priestley. *A Science of Operations. Machines, Logic and the Invention of Programming.* Springer, 2011.
28. Uday S. Reddy. Objects of closures: Abstract semantics of object oriented languages. In *ACM Conference on Lisp and functional programming.* ACM, 1988.
29. John C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings. Colloque sur la programmation*, pages 408–423, London, 1974. Springer.
30. Richard L. Sites. Algol W reference manual. Technical Report STAN-CS-71-230, Computer Science Department, Stanford University, 1972.