DRAFT

A simple consideration with far-reaching consequences.

> "I believe that there will be no real progress until programmers learn to distinguish clearly between a language (definition) and its implementation in terms of compiler and computer. The former must be understood without knowledge of the latter. And we can only expect programmers to understand this vital distinction, if language designers take the lead [...]. Hence we conclude that the first criterion that any future programming language must satisfy, and that customers must ask for, is a complete definition without reference to compiler or computer."
>
> Niklaus Wirth [1]

In a truly decomposed system, as we have when a whole is regarded as composed of a number of parts, the composition defines how the relevant properties of the whole depend functionally on the explicitly postulated properties of the parts, a functional dependence which is given without any reference to the internal structure of the parts.

For instance, when a mathematical proof of a theorem appeals to a lemma, it appels to what the lemma asserts, and not to how the lemma has been proved: making the remainder of the proof of the theorem clearly independent of the specific way in which the lemma may have been proved, is one of the main justifications for the lemma's introduction. Similarly: when a program in machine code contains a multiply instruction, its occurrence has to be justified by what the multiplier, when invoked, should achieve, regardless of how the multiplier happens to work. (No mathematician using a proven lemma will complain, when the known proof for that lemma is replaced by a more elegant one; similarly, no machine code programmer will complain when the multiplier in his machine is replaced by a new multiplier with, say, a lower heat dissipation.)

The fact that the relevant properties of the whole depend in a defined way on the explicitly postulated properties of the parts has two important consequences. Firstly --as indicated above-- it creates the environment in which a correct part can safely be replaced by a different correct part, viz. provided

that it has the same postulated properties. Secondly, it localizes errors in the case of malfunctioning: either the desired properties of the whole do, in this composition, not follow from the postulated properties of the parts, in which case at least the upper level design needs a revision, or the flaw can be pinpointed to the malfunctioning part(s), viz. each part that does not display the properties of which it has been postulated that it should have them. Note that without true decomposition, we can at most observe that the whole does not have the desired properties, but that the question which of the parts is to be blamed, is void. (More specifically: a program that has not been truly (de)composed is either correct or wrong: in the latter case it makes no sense to try to count "the number of bugs. For lack of "bug identity" the only defensible answer would be "one".)

There is an overwhelming experimental evidence that such a true (de)composition is indispensable for the design of a logical system above the level of simplicity such that it can be designed with a single stroke of the pen. (If we take --and I do-- the analogy with mathematics seriously, we can even add: more than twenty centuries of experience.) The experimental evidence is so overwhelming that I find it very hard to defend any design effort of some ambition that is tried without such an explicit, true (de)composition.

* * *

High-level programming languages certainly serve as the carriers of designs above the level of simplicity where a single stroke of the pen suffices, and in the whole system envisaged we can identify as "a part" teh machinery that should be able to execute programs written in the high-level programming language used: that part has even a name, viz. the "implementation" of that language. (The significance of that part is considerably increased by the circumstance that it is common to many applications.) It is, therefore, indispensable to have an explicit statement of the relevant properties of that part, independent of its internal structure; we must have a statement of what that part should achieve, independent of how it might work. It is only then, that the language users knows what he can rely upon and is in a position to prove that his design has all the relevant properties, it is only then that the language implementer knows his obligations and his freedom.

* * *

All this is so obvious that it is nearly embarrassing to write it down.

And, indeed, I would feel like flogging a dead horse, were it not for the fact that the above conditions are more often violated than met, and that this status quo has even its defenders.

From a historical point of view this is understandable. Many programmers work with older programming languages, the semantics of which they can only appreciate via the language implementation they are daily working with. For lack of adequate definition they are seduced to mistake what the implementer has done for what the implementer should have accomplished, and to forget that deriving lacking specifications from a non-exhaustive set of explorations remains guesswork. They live in an environment where the strictdistinction between (the definition of) a programming language and its implementation has never been carried through, and the suggestion that a crucial language requirement is that its semantics be defined without any reference to implementation details, without any reference to computers and compilers, woulc very well amaze (or perhaps even alarm) them.

To make matters worse, a number of subsequent efforts at rigorous definition of programming language semantics have only added to the confusion, viz. all so-called "operational" language defintions that try to define the semantics via an interpreter for that language. The definition of the semantics by means of an interpreting automaton is logically equivalent to defining a language entirely via a (standard) implementation. Without a formal technique for deriving from such an operation definition the language properties that the programmer needs to know, such a definition is useless: as it stands it gives us just enough grip on the language to perform for a given program and a chosen input a desk simulation! With a formal technique for deriving (all?) the relevant language properties, such an operational definition is at least cumbersome (because indirect), usually overspecific, and certainly misleading. It is misleading because, of all the possible ways of implementing the language, a special one has been singled out as "the defining implementation"; it is also misleading because it fails to provide a definition independent of any computational model. Such a definition, however, is needed by the conscious programmer who tries to separate his concern for program correctness from his concern for the costs of program execution.

* * *

For nonoperational (also called "postulational" or "axiomatic") definition

methods see, for instance, Hoare [2]. To nonoperational methods two objections have been raised. The first objection is in theory a serious one, but in practice it seems less so: the nonoperational methods allow us to define "impossible" programming languages, "impossible" in the sense that they defy implementation. The second objection I have heard is that the axiomatic method is only particable in the case of simple and elegant programming languages. If --what I fervently hope-- that statement is true, I cannot regard it as a shortcoming of the axiomatic method, on the contrary!

* * *

The above has been written because, once a necessary condition for progress has been identified, it seems such a pity and such a waste when it remains for so many people unfulfilled.

[1] Wirth, Niklaus "Programming Languages: what to demand and how to assess them." Bericht 17 des Instituts für Informatik, ETH Zürich, March 1976. (Presented at the Symposium on Software Engineering, Belfast, 8 - 9 April 1976.)

[2] Hoare, C.A.R. "An Axiomatic Basis for Computer Programming." Comm.ACM 12, 10 (Oct. 1969), 576 - 583.

Plataanstraat 5
NL-4565 NUENEN
The Netherlands

prof.dr.Edsger W.Dijkstra
Burroughs Research Fellow