Jan Bergstra

Professor Jan Bergstra was born in Rotterdam in 1952. He read mathematics in Utrecht, where he obtained a Ph.D. in mathematical logic. After six years with the applied mathematics group at Leiden University, and two years with the computing science group at CWI in Amsterdam, he became professor of software engineering at the University of Amsterdam and simultaneously professor of applied logic in the Department of Philosophy at Utrecht University. He still holds these chairs. He worked for Philips Research in 1988-1989. He is a consultant with CWI in Amsterdam and with Philips Research Laboratories in Eindhoven, and a visiting professor at Swansea University. His research has focused on methods for specification, design and verification of computer programs and systems, in particular process algebra, module algebra, frame algebra and term-rewriting systems. He is an associate member of the 'Onderzoeksschool IPA'. He has been active in the design of new curricula, in particular in the field of artificial intelligence in Utrecht and in Amsterdam.

Jan Bergstra

Provably Correct Programming and Beyond

A Comment

Introduction It is a pleasure and a special honour for me to comment on professor Hoare's lecture on this occasion. In my e-mail conversations preceding this event I have written him that in my view The Netherlands is too small a country for his talk. By this I mean that the number of occasions in which software is produced of such precious importance that a proof of correctness is considered vital is very small indeed. Aeroplanes, missiles, nuclear reactors, that line of products generates the search for bug-free software, which has been the subject of Tony Hoare's work for many years. It may be said that these worries have met a fair amount of scepticism here. This is understandable. At the time that I was a student, the famous theoretical physicist Veltman, himself quite an influential programmer, was known in Utrecht for the statement that physicists make computers faster and computer scientists make them slower. How right he was!

With a big leap in time I invite you to contemplate the latest hype in computer software. This is Java, a programming language that was designed by researchers from SUN micro-systems, a spin-off from Stanford University. The latest Java conference numbered about fifteen thousand participants the language being only four years old. This is a so-called revolution. So one may ask: what important feature does Java have to offer. I will return to this question shortly; first however let me say that no one in the field of software research can ignore Java; it is an elegant collection of many ideas that have been developed in the last twenty years. It combines object-oriented programming with parallel programming in the form of sophisticated multithreading. Object orientation is a notion that has come to stay in programming since Dahl designed the language Simula in the sixties (6).
In the seventies Hoare co-authored a book with Dahl and Dijkstra, which has turned out to be the best known title in programming methodology ever produced (7). This marks Hoare's influence on the emergence of the concept of structured programming, which has been of heuristic value ever since.

At the University of Amsterdam I was supposed to lecture on Java last year. So I visited the local book shop, I bought some fifteen books each introducing the language, in order to see what it is about. It turned out that even the tobacco-shop near to where we live sold books on Java! My first discovery, to which I will return later, is that I could obtain no real clue as to what Java is and how it works and that this will probably be the case forever. My second observation was that only one reference to the theoretical literature on programming and software engineering has made it into the majority of these books. This is a reference to Hoare's concept of a monitor. A monitor is a generalisation of Dijkstra's concept of a semaphore. Semaphores are values that can be inspected and changed by several processes that operate in parallel and do not even need to know about oneanother's existence. Every computer user nowadays takes it for granted that computers perform tasks in parallel in very much the same way that cars move through the streets simultaneously and almost independently. It would be rather silly if computers or processes taking place within computers were waiting all the time for one another to terminate, even if they were not dependent on oneanother's output at all. But every now and then some form of communication between computers and between processes running inside computers must necessarily take place. And now the comparison to car movements provides little indication as to how such communication may work at all. Human parallelism takes place within a complex intertwinement of different communication mechanisms, many of which are not easy to understand at all. How parallel mechanisms can or should work on a computer has been the topic of a great deal of research. One solution is to install mechanisms like the well-known surface mail, with which all of us are familiar. In the case of surface mail independent processes, in particular the person who posts a letter and the employee who empties the mailbox, handle the same object, in this case the mailbox. I hope that you see why the semaphore and monitor play a role here. These notions formalise the simplest possible mechanisms that allow one to realise something like a surface mail system within a single computer or within a computer network. That communication structure in connection with a strict regime of first come first served takes place in Java. Although the idea of a semaphore is trivial in its very simplicity, it contains the key to the systematic construction of operating systems which allow a multitude of tasks or processes to run in parallel. Monitors generalise this notion in a remarkable way. Where a semaphore is just a Boolean value, that is a zero or a one, the monitor lives in a space of arbitrarily complex combinatorial objects. These monitors play a key role in the implementation of the multithreaded parallelism that Java's designers are so enthusiastically promoting.

It should be noticed that fifteen years ago, when the language Ada was designed for general use in all American Defence activities (it is still widely used there at the time of my writing this), it was also a reference to a design proposal by Hoare which constituted by far the most visible reference to then existing theoretical work. That reference concerns the notion of synchronous communication or handshaking, which Hoare was the first to demonstrate to be a perfectly meaningful construction in programming. Two processes in parallel may decide to shake hands, and to transfer some item from the one to the other at the same time. The key issue is that the way in which the two processes come to the conclusion that the time has come to shake hands, is not described in the program but left to the underlying automatic implementation. It means that by ignoring this sort of details altogether, programming could be done at a significantly higher level of abstraction than before. That makes it simpler to write programs, and at the same time but far more importantly, makes it simpler to understand their correctness. In my view Java's perspective on parallel mechanisms is a step backward in comparison with the perspective offered by Ada so many years ago. No matter whether I am right in this assertion or not, I dare say that programming language design and use is as much a matter of cyclic development of fashions and trends as it is of scientific and technological progress.

Returning to Veltman's observation that computing science makes computers slower, we cannot fail to notice that Java has quite a remarkable performance. The language from which Java has been derived is called C++. This is a language of a staggering complexity, which was designed by Stroustrup at AT&T Bell Labs (12). Now Java happens to be slower than C++. One should not be amazed to observe a Java program running 20 to 40 times slower than its straightforward translation into C++. This is breathtaking even for computer scientists, I can assure you. Whereas even computer scientists hope to make computers faster - usually they hope that the field in general will accomplish this - the more pretentious researchers even intend to do this themselves. Stroustrup is clearly one of those.

How can it be the case that this kind of breakdown in computing power provokes such an unbelievable hype at the same time. The answer is: methodological advantages. This sounds vague and so it is. One advantage claimed for Java is that Java programs are supposed to be far more portable than those written in previous languages. Portability means that programs can run on other people's computers once they work on ones own machine. In the course of my teaching job I have come to understand that "more portable" does not imply "portable". It just means that you face fewer miserable difficulties than you had to face before when porting a program. A second advantage claimed for Java is in the field of security. I will briefly try to explain Java's perspective on this issue. A major distinction in software is between enemy and friend. A friend, or rather a "friendly program", often referred to as friendly code is allowed access to your files because you know that it will not do any harm. Hostile code, however, should be feared because it can, on purpose and with bad intentions, damage your data. Hostile code can only be admitted under quite restricted conditions. The Java designers have introduced a very nice terminology in this connection.

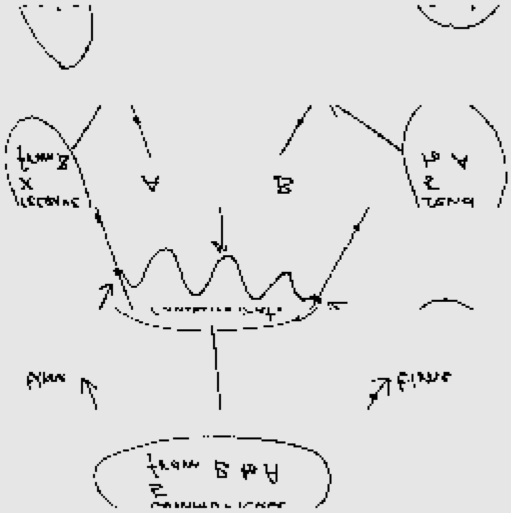

FIGURE 1 A HANDSHAKE

FIGURE 2 TWO COMPUTERS IN PARALLEL

Java implementations produce a so-called sandbox in which Java programs may play. While playing, these programs cannot leave the sandbox and that implies that they will not do any harm. The official term for such a sandbox is a virtual machine and that sounds nice as well. Unfortunately, playing in the sandbox is far slower than the real work and this explains why Java is so slow. The big advantage, however, is that you can invite Java programs from all over the world to demonstrate their sand castles on your own machine. That implies that if you download an unkown Java program from a remote site on the Internet, worrying whether this is a friendly program or a hostile one is irrelevant. An even more important consequence of the sandbox concept is that the programmer has in mind a far more abstract or high-level model of the machine when programming Java than in case he or she is programming in C++. This in turn implies that certain very notorious kinds of errors, so-called memory leaks, will never occur in a Java program. As a consequence, for many tasks, writing correct Java programs is significantly easier than writing flawless C++ programs.

Next I invite you to observe how young computing is as a field by pointing out an elementary kind of result which professor Hoare was the first to obtain. Take an automated version of the telephone book. First, however, consider the physical telephone book as it is still in archaic use these days and imagine that the number connected to a given name has to be retrieved from it. When looking up a number most people will automatically do the right thing. Rather than start at page one and read the entire book until the given name is found, upon which the required number can be immediately obtained by looking in the second column of data, they will open the phone book in the middle and then decide whether subsequently to look forward or backward in the book. This process is then repeated until a small set of names is inspected in a linear fashion till the target name has been spotted (or found to be absent from the listing). The difference in speed between both procedures, reading the entire text from the beginning, versus opening the book in the middle and so on, is enormous. Now if the phone book has to be constructed automatically, a similar observation applies. The naive approach is to take entry after entry and to put these in the list one by one. But this process, which is called sorting, can be done in just the same way as the search with the same impressive gain in time. Given the unsorted listing of entries, one can split it in the middle and now decide which part to sort first. As early (or late) as 1962, professor Hoare came up with the Quicksort algorithm that does just this (9). Given the fundamental importance of sorting, it is absolutely amazing that the discovery of an efficient algorithm to perform sorting is so recent. Of course, the algorithm would always have been found, if not by Hoare then by someone else and perhaps not much later, but someone has to do it.

Since 1984 my own work and that of most of my direct colleagues and students has focused on the consequences of Milner's version of Hoare's design of the handshaking mechanism, which I have mentioned before. Let me remind you that a handshake between two processes is an event in which these processes share the same action, so to speak. The novelty of the mechanism as an instruction for computer languages is that both processes co-operate in finding the right time to perform a handshake. In fact, the handshake is not a single instruction, but a pair of instructions, one of which is executed by each of the partners in the handshake. As these partners in our topic are computer programs running on a machine, these instructions feature as so-called atomic actions in the respective programs. Two years after Hoare's proposal (10) for the handshaking mechanism, Milner proposed a more symmetric notation with a very nice mathematical background (14).

In this fashion Milner took Hoare's proposal and turned it into a calculus, a kind of logic of processes. This calculus was then transformed into a topological space by De Bakker and Zucker (2). Together with Jan-Willem Klop I have modified Milner's design and De Bakker's topological space into an algebra, in particular a process algebra (4), and many years have been happily spent on this opportunity to formalise the basic aspects of computer programming in terms of algebraic equations.

Before that, I spent several years with John Tucker on the algebraic formulation (or rather specification) of so-called abstract data types, a subject which has one of its two roots in the monitors that I mentioned before as Hoare's contribution (still affecting as recent a design as Java) and the other root in the work of three East-German computer scientists working in Dresden during the early seventies.

I very much cherish the view that many, if not most, concepts of programming can be best expressed in terms of simple equations. It is quite likely that my optimism slightly exceeds realistic proportions. Justified or not, in co-operation with Jan Heering and Paul Klint in Amsterdam I have been involved in an attempt to use sets of equations directly as a programming language.

That is no news as such. As early as Backus (1), the designer of FORTAN, in his Turing award lecture, presented the view that machines could do a far better job on computing the solutions of certain kinds of systems of equations than on evaluating the well-known kind of programs as written in COBOL, PASCAL or FORTRAN (and the vast majority of the 430 programming languages that Capers Jones describes in his remarkable book (13) on the even more remarkable year 2000 problem). Since then many experiments have followed, most notably in recent years in the languages CLEAN and HASKELL. Our project in Amsterdam makes an attempt to be innovative by being very liberal about the kind of systems of equations that can be solved. (Though definitely less liberal than several preceding efforts in this subject have been.) That is necessary because the problems that emerge in practice turn out not to fit the best known and theoretically well understood patterns of systems of equations. At the same time we have been forced to become increasingly vague about what a solution of a collection of equations stands for. Each insight that we could obtain on what a solution designates has been misused by us to allow more types of systems of equations to be given as an input to the machine. We are now at the point that we have a language, called ASF+SDF (3), and a program which solves systems of equations (abstract data type specifications in different terminology) written in ASF+SDF. Members of our team(s) have improved this program through several stages and an attractive performance can be reported at this stage. Moreover, we have produced a mysterious if not contrived theoretical explanation of our calling the output of the this program a solution of a set of equations. To the mathematical minds in the theory of computation, what we propose exhibits an appalling lack of elegance. But in practice it works. It helps, for instance, to modify programs that contain the notorious year 2000 bug into programs that have a higher probability to do useful work in the next century. At present there is no known theoretical insight into our language ASF+SDF at all. Each useful insight that did exist at some time has been incorporated in the design in such a way that this very insight fails to apply in full generality afterwards. I call this diagonalisation over theoretical understanding. Every effective and general theoretical insight is incorporated so as to allow more programs, with the effect that fewer general facts hold true and in particular the insight ceases to be clearly valid. In the use of ASF+SDF that I currently notice, and which I expect to grow significantly in the future, explanation, reflection and analysis play no role at all. What really matters is the expressive power of the formalism, which pays off in terms of programming effort only. The process algebra mentioned above made its entry in these efforts after many years as well (5); the details of that application would exceed the space available here however.

It is well-known that adding to a language a notion which is clear from the viewpoint of an external observer invalidates the insight on which that notion was based. The old liar paradox indicates that if one assumes that a complete insight in the validity of natural language sentences exists, and that therefore a truth predicate based on that insight may be used, that very insight has evaporated into a paradox. Set theory was designed by Cantor to be a theory about collections of mathematical items. Some years later, Russell concluded that the set of all sets must be one of the more interesting examples. He then observed that this notion cannot be added to set theory that easily, and even worse, that set theory had to be redesigned. A naive theory of sets leads to a paradox quite similar to the liar paradox. Simultaneously, set theory provides an excellent example of a case where a viewpoint, in this case the attractiveness of collecting all sets into a single container, is best kept outside the formalised theory.

Similarly Gödel started with his famous insight that all consequences of number theory can be proved by simply searching through a listing of all possible proofs (completeness). Then he added a notion of truth to the language only to find out that immediately completeness was lost (incompleteness).

In the case of programming languages, matters are definitely less clear cut and the analogies with the famous paradoxes may seem somewhat optimistic. Nevertheless the observation that formalised versions of insights available to an external observer can be added only at the cost of these very insights is being illustrated on a daily basis. That is what happened to us in the case of ASF+SDF as well.

The development of programming languages seems to follow this seemingly irrational pattern all the time. Java is now so incredibly complex that a hard-headed attempt to understand its working in full and theoretical detail seems to be absurd and unrealistic, if not simply irresponsible. I may be entirely wrong in this observation, but where the computer scientists have done their best to make programs easier to understand, at the price of making the computer slower, the concepts that they have developed have effected a converse result altogether. Computing as a social phenomenon pays very little attention to analysis and reflection and much more to action. As a consequence the mathematical tools for analysis of programs and reflection about them have all been "applied" in order to design novel programming mechanisms. In isolation these mechanisms do indeed lead to very understandable code. In almost unconstrained combination, however, the theoretical mind is totally defeated. That development is vividly visible in Java. This language combines features of programming in a way which makes it particularly difficult to make any substantial and manageable mathematical model of what is going on at all. At least temporarily, one is led to believe that the property of a programming language that it is understandable from a mathematical point of view is simply an ergonomic disadvantage. Not only is it a disadvantage, a language simply cannot survive the emergence of a clear mathematical model for it. Successful theoretical work is the most damaging attack on a language that one can imagine; programmers cannot live in a world which they can fully understand. This disadvantage of comprehensibility is so damaging that even the very existence of less comprehensible languages seems to be a problem for the life expectancy of a programming language. The explanation of Java's attraction over C++ is not by no means its relative simplicity. As a group, programmers sense that due to its multilevel design, the sandbox and so on, Java is structurally and definitely more complex than C++ and that that creates its attraction. After throwing away the most notorious sources of complexity of C++, namely the memory pointers and multiple class inheritance, the addition of the sandbox concept and flexible parallelism to Java easily makes up for the difference.

These views, however, should by no means be considered to be a source of worries at all. Not even for that part of the programming research community that is so totally committed to the concept of provably correct software. What can be concluded is that although professor Hoare has been remarkably influential in the recent history of programming design, the question whether we are even close to a point of conceptual stabilisation in the design of programming languages is entirely open. The ambition to offer our students the eternal principles of programming is at present both futile and pointless. This certainly makes the field quite intriguing to watch. Of course both the possibility and the importance of a logically complete analysis of programs written in a wide range of present and future formats and notations must be fully appreciated at all stages. This very methodology of logic analysis itself does constitute a piece of knowledge, highly relevant to programming, and of stable and lasting value indeed.

In addition, besides investigating the foundations of correct program construction, research should take better notice of the psychological aspects of programmer productivity. The cycle of trial and error with every now and then a bit of systematic testing places a programmer in a feedback loop all the time. Fundamentalists refer to this approach as "quick and dirty" programming. Even if clear thinking and systematic look-ahead could prevent a range of errors in advance, it seems to be more satisfactory for a programmer in the long run to spot and correct these errors experimentally in a very systematic and reassuring feedback loop. The methodology of quick and dirty programming, or tactical programming as I prefer to call it, may turn out to be just as rewarding a subject as the methodology of correct programming has proved to be. No doubt tactical programming objectives have become a very strong influence in programming language design. It is conceivable that vastly different programming languages will be needed for both styles of programming in the future.

I cannot end this talk without returning to my initial remark that the quest for correct programs fares better in a bigger country than ours. Dutch thinking is entirely commercial and in times where millions of millennium bugs must be swallowed at once, the very worry that well-designed software might still be flawed occurs to most industrial managers as a sign of naive and academic luxury, largely out of touch with practice and certainly out of fashion. This, however, is an illusion. Program verification in Holland is applied in important cases only. For a case to be important, the potential loss of money etc. is not a decisive criterium. What counts is national pride. One source of such pride is that since the 1953 disaster no casualties have occurred from lack of control of the sea or from high water levels in the delta

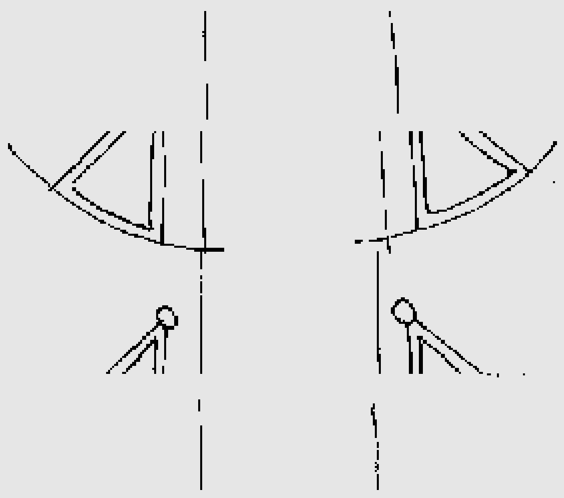

FIGURE 3 ‘WATERKERING’, PARALLEL DOORS

caused by excessive rain. This is in part due to enormous projects constructing protective artefacts along the various coasts. The recent construction of the “Waterkering” in the Nieuwe Waterweg did require some very sophisticated program design.
That project is clearly sufficiently critical to merit program verification. Non-trivial verifications have indeed been carried out on the software for the Waterkering.
It is interesting to notice that the tools for that verification work have been designed by a Dutchman as well, Gerard Holzmann (12). At AT&T Bell Labs he designed and implemented a remarkably effective tool kit for protocol validation.

Together with his predecessors Floyd and Dijkstra, Hoare has designed the conceptual principles that can be applied if bug-free software has to be manufactured.

The so-called Hoare Logic (10) can be used to explain to anyone, from first principles and remarkably easily, how to perform valid reasoning on program behaviour in the case of a limited programming language, which is in principle of universal expressive power at the same time. Even the most capricious development of programming languages cannot prevent this ability from becoming essential every now and then, be it in a small minority of cases only. Besides these conceptual principles, technological principles too are essential for the production of bug-free software. Technology has to do with the production of large and complex correctness proof. The subject of large proofs is technological because it unavoidably requires computer support. The technology in its turn has a mathematical foundation, which has to do with the representation of extended logical arguments on a machine. As a closing remark it is worth mentioning that Brouwer’s intuitionism provides the conceptual foundations for these representations.

The version of type theory that de Bruijn has developed in Eindhoven, exploits the intuitionistic ideology for the purpose of designing computer tools supporting the representation and checking of the large mathematical proofs that emerge from program verification tasks. Several decades later this approach is still surprisingly alive throughout the world.

I thank you for your attention.

References

(1) Backus, J., 1978. Can Programming be Liberated from the Von Neumann Style? A Functional and its Algebra of Programs. *Communications of the ACM*, 21(8): 613-641.

(2) Bakker, J.W. de, and Zucker, J.I., 1982. Processes and the Denotational Semantics of Concurrency, *Information and Control,* 1/2: 70-120.

(3) Bergstra, J.A., Heering, J., and Klint. P. (eds.), 1989. Algebraic Specification, *ACM Press Frontier Series*, The ACM Press in co-operation with Addison-Wesley, New York.

(4) Bergstra, J.A., and Klop, J.W., 1984. Process Algebra for Synchronous Communication, *Information and Control*, 60: 82-95.

(5) Bergstra, J.A., and Klint, P., 1998. The Discrete Time TOOLBUS - a Software Co-ordination Architecture, *Science of Computer Programming*, vol. 31: 205-229.

(6) Dahl, O.-J., and Nygaard, K., 1967. SIMULA: a Language for Programming and Description of Discrete Event Systems, *Norsk Regencentral*, Oslo.

(7) Dahl, O.-J., Dijkstra, E.W., and Hoare, C.A.R., 1972. *Structured Programming*, London Academic Press.

(8) Gosling, J., Joy, B., and Steele, G., 1996. *The Java Language Specification*, Addison Wesley, The Java Series.

(9) Hoare, C.A.R., 1962. Quicksort, *The Computer Journal,* vol. 5: 10-15.

(10) Hoare, C.A.R., 1969. An Axiomatic Basis for Computer Programming, *Communications of the ACM,* vol. 2: 567-583.

(11) Hoare, C.A.R., 1978. Communicating Sequential Processes, *Communications of the ACM,* 21: 666-677.

(12) Holzmann, G.J., 1991. *Design and Validation of Computer Protocols*, Prentice Hall.

(13) Jones, C., 1998. *The Year 2000 Software Problem, Quantifying the Costs and Assessing the Consequences*, Addisson-Wesley, ACN Press.

(14) Milner, R., 1980. *A Calculus of Communicating Systems*, Springer Verlag.

(15) Stroustrup, B., 1985. *The C++ Programming Language*, Addison Wesley, Reading MA etc.

COLOFON

uitgave
Nederlandse Organisatie voor Wetenschappelijke Onderzoek
Postbus 93138
2509 AC Den Haag
070 344 06 40
e-mail: news@nwo.nl

NRC Handelsblad
Rotterdam

samenstelling en vormgeving
Afdeling Voorlichting en Communicatie NWO

basislayout
Kees Endenburg
Den Haag

druk
Van Marken Delft Drukkers
Delft

oktober 1998