Inaugural Lecture for the Schlumberger Centennial Chair

Jayadev Misra

1/25/2002

1 Preface

It is an honor to hold the Schlumberger Centennial Chair, a vote of confidence in me by my colleagues, which I value greatly. I am thankful to Schlumberger Corporation for endowing this chair. And I am especially proud to hold this particular chair because its previous occupant was Edsger W. Dijkstra. His work has influenced mine in substantive ways. In fact, what I have to say today is not profoundly different from what he has said earlier, and more elegantly.

Lectures of this kind tend to consist of polemics, holy pronouncements and a selective use of history. I will uphold that tradition. I won't talk about individual pieces of research being done in this department though much of it is very impressive, nor how we may all collaborate, though that is a noble goal. If I had the time, I would talk about theory and practice in computer science, its relationship to other disciplines and its impact on the society. I hold strong and unfounded views on all of the above. Fortunately, I don't have the time, so I will restrict myself to a few broad remarks on computer science teaching and research.

To summarize my position, and I borrow a lot from Tony Hoare here: I believe that theories and design principles of a general nature will be far too weak to be of much value to the practitioners. We should develop specialized theories that are applicable in specific domains, and we should work on binding these theories and principles much like the way we structure large systems today. What I am proposing amounts to a strong prescription for empiricism, that we have to do a large number of experiments to understand where theories can play a role, and which kinds of theories would be most appropriate and when.

A small clarification on terminology: I consider theory as a tool that reduces the amount of experimentation, often by an infinite amount.

2 Computer Science As An Introspective Discipline

Computer science has, traditionally, been an introspective discipline. Major developments have been inspired by the capabilities and limitations of computers, not by the dictates of the real-world applications. Programming languages have

not been inspired by natural languages, but by the way a computer works. Early languages had special features, and restrictions, to exploit the hardware of specific machines. Even in LISP, car and cdr stood for the contents of address register and decrement register; IBM 704 and 7090 could store an "address" and a "decrement" in each memory location. Constructs in operating systems, such as process and message communication are inspired not by their counterparts in the real world, but because of the structure of computers and communication networks. Mutual exclusion was introduced in the THE multiprogramming system by Dijkstra, to provide a form of coordination among the asynchronously operating devices. If a new kind of communication network can implement, say, atomic broadcasts efficiently, its inclusion in programming languages and exploitation in operating systems will be immediate. Growth of our discipline has been traditionally driven by internal considerations, not by the needs of the end users.

There are a few exceptions to this general observation; here is one instance where the real-world contributed to fundamental developments in computer science. Simula-67 was developed in 1967 for programming simulation applications. Its designers, prominently Ole Johann Dahl, were interested in representing the behaviors of real-world entities –bank tellers, customers in a hamburger joint, car washes– within a programming language. The most advanced language of the day, Algol 60, was inadequate because it did not allow a called procedure to live beyond its caller's lifetime. Simula introduced several new concepts which underly object-oriented design, modularity, the notion of process and multiprogramming. Clearly, Simula's impact has been felt far beyond the simulation community. It has abstracted the essence of a class of applications and shown that these principles have wider applicability.

In a similar vein, Robin Milner was doing theorem proving in the late 70s. He found that adding strong-typing to a programming language made it considerably easier to define "tactic", the strategy employed in a proof. Development of strong-typing and its implementation in the programming language ML is a watershed in programming language research; and we find that strong-typing is slowly trickling down into imperative languages and also being applied in areas such as security. And, finally, the world-wide-web is a direct consequence of some physicists trying to share their data. On a personal note, Mani Chandy and I developed an area in the early 80s that is now known as "Distributed Simulation". This work was directly inspired by the need to simulate the operation of a commercial telephone switch.

I contend that future advances in computer science will be driven more by external applications than by the intrinsic nature of the computing devices. Process

control systems, as we see in chemical plants, have already led us to ask questions about modeling continuous behavior. We are seeing impressive new theories about timed and hybrid automata which are abstractions of such systems. We are learning to pose control-theoretic questions using temporal logic. Vulnerability of smart cards is making us develop interesting theorems about security protocols, which clearly have wider applicability. And, I believe we will develop theories about streaming data –that is real-time football– for efficient transmission over the internet. May I suggest that embedded systems –devices with limited processing power and storage that have to provide real-time services– are a suitable topic of research?

Each application we consider will present its unique problems, and require the development of domain knowledge, a specialized theory for that domain. Such theories have to be integrated with our existing theories. How can computer scientists enrich their own discipline by drawing inspiration from others? The prospect is that we will work with other professionals to learn about new applications, observe those aspects which are not currently amenable to representation or solution, and, thus, abstract the problems that have deeper significance. The danger is that computer scientists will merely be used to write programs; they may not understand the significance nor master the finer points of an application in another domain. The solution, in my opinion, is to educate professionals in other areas in computer science, and encourage them to contribute to the core of computer science, much like the way physicists have enriched mathematics. This brings me to a point close to my heart; our horrendous record on teaching the basics of computer science.

3 Teachability As A Criterion Of Research Success

It is instructive to compare a mathematics text book with a computer science text book in a high school curriculum. I had the misfortune to do so at some length, and to put it simply, we suck. I assume that we are sending a strong message: don't even consider computer science as a career unless you are a masochist. As computing professionals, it is our duty to speak up against a culture that equates computer literacy with mastering the intricacies of a production programming language. And, as long as we don't clean up our act, we can't expect decent scientists from other disciplines to contribute to ours.

The horrid state of affairs is partially due to the novelty of our discipline. There is no consensus on what constitutes the basics of computer science, let alone how

a well-rounded computer scientist should be educated. In the next 20 years, I hope to see the first three volumes on the basics of computer science, textbooks that we all agree should be required reading for all scientists and engineers. Perhaps some of us could look into this long term project.

A related issue is teachability of our own research contributions. Teachability is perhaps the most important criterion of the success of academic research. If we can't teach what we do we have failed as teachers and researchers. And, to teach we are forced to abstract principles from a mass of details, which is often the ultimate goal of research.

I have personally benefited from having the "burden" to teach and teach well. The students are, understandably, skeptics. They are always on the look-out for someone pulling a fast one. Given an involved argument, inevitably, one student would say "but you have not considered the case where event B follows C which follows D" etc. Wrestling with questions of this nature forced me to find more convincing ways of imparting the information, which have led to some of the discoveries of which I am most proud.

Let me illustrate my remarks using designs of distributed systems as an example, an area with which I have some familiarity. In the early 80s, such systems were seen as extensions of sequential programming systems. You merely added a few primitives —send, receive, block— to a sequential programming language to turn it into a "powerful" concurrent programming language. Unfortunately, designing and reasoning about such programs proved immensely difficult, because of the entanglement of the sequential and concurrent programming issues. Mani Chandy and I would spend hours in a classroom explaining a single algorithm, and still convince very few that the algorithm "works".

Our classroom examples provided the impetus to embark on a project to formalize and simplify treatments of distributed algorithms. We made a conscious decision to separate the sequential from the concurrent aspects, and treat concurrency purely as non-determinism. It was then possible to treat each aspect separately, yet combine them appropriately. We were surprised by the simplicity gained through this disentaglement. And, we could now silence the skeptics in the class under a mountain of formalism!

4 Precise Vs. Lucid Systems

It has been axiomatic in our discipline that effective designs are possible only when they are simple. An effectively-designed system consists of a number of

nearly independent modules connected by narrow interfaces, the interfaces themselves are simple and the system provides an external interface that permits its integration in a variety of environments. Simplicity is so fundamental that it requires a more thorough treatment.

To that end, let me distinguish between a "precise" and what I call a "lucid" system. A precise system is one which obeys precise, i.e., mathematical rules. All the systems we build –computer architecture, operating systems, programming languages, databases– are precise. Most human activities are not. Hand-writing recognition is a very interesting example of a non-precise system. In order to implement a hand-writing recognition system on a computer we have to make it precise, by ignoring certain features, simplifying certain issues and making certain arbitrary decisions. Unlike hand-writing recognition, chess is precise: given a sequence of moves, it is possible for a computer program to decide if the moves follow the rules for a game of chess. Similarly, a compiler can check the syntactic validity of a C++ program and an unambiguous meaning can be assigned to any such program.

All that we do on a computer is precise; yet, that is not enough. It is not sufficient to know that a program has an unambiguous meaning, though that is essential. It is not sufficient to know how a particular program may be executed, though such knowledge is usually precise. It is not acceptable to claim that an inscrutable program satisfies another inscrutable specification, though both may be precise mathematical entities. We need to build systems that are not merely precise, but lucid, in the sense that they permit deductions of useful facts about them without extraordinary effort. The absence of interesting theorems about chess is what makes it so difficult to automate chess playing. We have to resort to searches, and a number of complicated heuristics to build a good player. And, I contend that lack of useful theorems about a typical C++ program is what makes it so difficult to make claims about it: its intent, whether a particular change will have a disastrous effect on its execution and whether it can be integrated with other programs. In other words, we typically build mathematical systems whose properties we cannot discern.

I would like to argue that our sole aim in programming is to build lucid systems, whose properties we can deduce, which allow easy integration in a variety of environments and which can allow evolution over time. One class of such systems is based on functional programming. Programs written in a functional programming style are impressively lucid. It has proved to be considerably easier to do modularization, programming in layers and program integration in this style compared to the traditional styles.

5 The Case Against A Grand Unification Theory

So, is functional programming the answer? It may be, but I don't see it. I am bothered by several important questions that are not covered by functional style of programming: non-determinism, persistent storage management, and matrix-based computations. I don't see how to do effective distributed application design in the absence of non-determinism. Efficient access and update of persistent storage is one of the most fundamental requirements in many applications. Most scientific packages insist upon constant time access to matrix elements; the best that functional systems can promise is O(log) access time; an unacceptably high price to pay to move from a precise system to an lucid one.

I don't see any easy extensions of functional-based system to include these domains. What I would like to propose is that we abandon the idea of a Grand Unification Theory, and design specialized theories that handle a few things well. I would like to see, for instance, a decoupling of (1) persistent storage management, (2) computational logic (which can be based on functional systems), and (3) methods for orchestrating a number of concurrent threads. I would like to see effective theories in each of these domains, and an integration of these theories. I am involved in such an effort at present.

The point I want to make is that general design principles for programming, or even distributed programming are inadequate for the developments of the next generation of applications. These applications may have to make use of a number of theories drawn from different areas: module integration, interface specification, type theory, security, fault-tolerance, perhaps aided by automatic tools such as program analyzers, static type-checkers, and performance profilers.

The next question is how to tie the various theories together. I have very little experience or intuition to go on here; I would like to hear from those of you who have thought longer and deeper about this problem. Let me suggest that the process is similar to module integration in programming. We have to develop the interfaces between the theories, and compose them much like the integration of program components. I believe that simpler composition rules should apply as we compose more complex theories. For compositions of very large systems and theories, something akin to type checking only should be required; detailed proofs should be rendered unnecessary by design.

In this context, let me suggest a problem worthy of your attention. The fundamental structuring principle for systems is hierarchical organization. It permeates the world around us; social and political systems are hierarchical. We have found from experience that tree-structured file systems, hierarchical organization

of large domains such as the internet, and system designs in which each component itself is regarded as a system, is convenient. Even within object-oriented programming, the notions of components and encapsulation impose hierarchical structures.

Is it time to consider a more democratic structuring principle, similar to what they have in relational databases? I am driven to these considerations by the need to view a system from different angles. I can look at this building floor by floor, or by its electrical, plumbing and computational infrastructures, or by the people who work here by their professions. I would like to slice the same system in a variety of ways depending on which aspect of it I want to study. I would like to see my file system not only as being hierarchical but also providing groupings according to other criteria: all files that constitute the chapters of a book ought to be arranged in a sequence so that I can read from Chapter 3 to 7, say. I would like to see the list of all printers in the campus, even when the campus network is organized by departments. How do we support multiple views of a system while retaining the advantages of hierarchical structuring? How do we exploit multiple views during program design and evolution?

6 Closing Remarks

Let me close with a couple of remarks on research and teaching. A few years back I was an editor for a computer science encyclopedia, an effort since abandoned. This gave me an opportunity to think carefully about the topics that are important, such as what constitutes the mainstream of our discipline and which ones are the tributaries. Most important, however, was the reflection on where my contributions fit into the mainstream, an exercise that Tony Hoare had recommended to me. Are you contributing to the mainstream, changing the direction of the mainstream, or rowing against the stream? I recommend this exercise to all of you.

Now, on teaching, since we all work in a university we believe that a university education imparts certain knowledge and skills to students that others without the education lack. The skeptical public, and their chosen leaders, usually do not share our view of our own exaltedness. As long as people without a university education are as effective in doing the tasks as those with an education this skepticism will persist. As long as high-school dropouts can code a large system as well or better than our graduates, we can't justify a large part of our existence.

It has been the guiding principle of my career in education and research to en-

sure that well-trained professionals are considerably more effective than inspired amateurs. There must be a gap in their performance, and the gap should be considerable. I suggest that it should be our collective goal to widen that gap.

Thank you.