The Next Generation of Programming

Programming as an Engineering Discipline

Chair: Juris Reinfelds (US)

Panel: Peter Van Royt (BE), Tiajaru Diverio (BR)

Keywords: programming, software engineering, computing

1. PROGRAMMING IS NOT TAUGHT IN THIS WAY

We define *programming* broadly as the step from specification to running program, which consists in designing the architecture and its abstractions and coding them into a programming language. Up to now, programming has been taught more as a craft than as an engineering discipline. Usually, programming is taught in the context of one (or a few) programming languages (e.g., Java, complemented with Haskell or Scheme). The historical accidents of the particular languages chosen are so closely interwoven with the fundamental concepts that the two cannot easily be separated. There is confusion between tools and concepts. This has led to different programming communities, based on different programming 'paradigms': object-oriented, logic, functional. The unity of programming as a single discipline has been lost. So programs often suffer from poor design. For instance, concurrency in Java is complex and expensive, so its use is discouraged. So Java taught programmers conclude that concurrency is *always* complex and expensive.

Program specifications are designed around concurrency restrictions, often in a contorted way. But such restrictions are not fundamental. There are forms of concurrency that are quite useful and yet as easy to program with as sequential programs. It is possible to implement threads almost as cheaply as procedure calls. If programmers were taught concurrency correctly, they could write specifications for and program in systems without concurrency restrictions, including improved implementations of Java.

2. THE KERNEL LANGUAGE APPROACH

Current languages, after five decades of language design, are capable of scaling to programs of millions of lines of code. This is not fortuitous, because they model essential aspects of how to construct complex programs. They are not just arbitrary constructions of the human mind; they merit scientific study. How can we separate the fundamental concepts in these languages from their historical accidents? The kernel language approach is one way. The full language is translated into a small *kernel language* that has a minimal number of *programmer-significant* elements, giving a student insight into what the language does. The kernel language has simple formal semantics, a foundation for the programmer's intuition and the programming techniques built on top of it. Reducing a complex phenomenon to its primitive elements is characteristic of scientific method. Earlier attempts to make programming an engineering discipline failed. The two approaches used to define languages precisely are foundational calculus and the virtual machine. A foundational calculus, like the λ -calculus or π -calculus, reduces programming to a minimal number of elements, regardless of whether these are relevant for practical programming – important for computation theory but largely irrelevant to programmers. A virtual machine, which defines a language in terms of implementation on an idealised machine, is useful for implementers, but concepts close to hardware are irrelevant to programmers.

3. A TEXTBOOK

Van Roy and Haridi's textbook based, on the kernel language approach, organises programming at three levels: *Concepts* – compositionality, concurrency, encapsulation, lexical scoping, higherorder, dataflow, state, inheritance; *Techniques* – how to write programs with these concepts; and *Computation models ('paradigms')* – data entities, operations, and a kernel language. Often-used programming techniques are explained in terms of simple kernel languages, and introduced progressively, with complete and simple formal semantics. Programming paradigms emerge as a kind of epiphenomenon, depending concepts one used and which properties held. The textbook is being tested 2001-2002, in three universities (KTH in Stockholm, Sweden, UCL in Louvain-la-Neuve, Belgium, and NMSU in Las Cruces, NM, USA), The latest draft of the textbook is always available at [<http://www.info.ucl.ac.be/people/PVR/book.html>] complemented by a full-featured open-source software package, the Mozart Programming System [<http://www.mozart-oz.org>].