

Introduction

Once bit by a lizard, we keep away from anything that looks like a lizard. This is the primitive inductive mechanism of brain, needed for survival. The pain is recorded as an experience associated with a set of similar entities. One can see the abstraction (recording of pain) and the generalization mechanisms (lizard-like entities) of our brain. These primitive mechanisms go through a series of evolutions as we learn to speak, go to school, etc. Roughly speaking, they finally take the form that we are going to refer to here as the mechanism of abstraction and generalization. Primitive as it may be, abstraction is a main component of the engine that led to the beginnings of mathematics.

Empirical discoveries in Egypt enabled the Greek to divest geometrical notions from physical entities representing them. It was only then that the Elements could be conceived. In the same vain but more algorithmically, rather than theorem-proving, Algebra was born by Alkharazmi, and Khaje Nassir created trigonometry. Hilbert revisited the process of envisaging geometrical notions in isolation and completely on their own in ways that opened the doors to modern mathematics. Thus, a straight line could be the great circle on Riemann's sphere, or anything else for that matter.

This mechanism of abstraction has a two-fold advantage. On the one hand, mathematics can grow on its own and without regard to such things as the color or material of objects representing the notions. On the other hand, the notions, and discoveries about their relationships can be applied to other apparently unrelated physical problems (the generalization phase).

Turning to programming languages, researchers have been following the same path as with mathematics all along. Starting with FORTRAN, data structures, abstract data types, structured programming and classes all are evolutionary attempts towards an abstract language that can live on its own and without dependence to material on which it executes, or more appropriately the operating system. The concept of a virtual machine for this liberation is at least as old as BASIC. Thus, researchers in computer science shrunk several millennia to a few decades by attempting solutions with awareness of history of mathematics.

At this time, computer scientists and researchers have provided us with sufficient abstraction, and domain experts have demonstrated how to generalize such abstractions in automating problems in many domains. It is therefore time for an abstract language with a high degree of semantic automation, dependant only on a universal virtual machine. Such language must be designed with the view to play the same role for a software engineer as applied mathematics does for an engineer.

The purpose of high **degree of automation of semantics** associated with statements in the language is to **alleviate implementation details irrelevant to abstract algorithmic solutions**. We will illustrate these ideas in the sections to follow.

The target of interest to us is the domain of problems that can be automated solely within the abstraction of a language based on a universal virtual machine. We shall refer to this domain as the Application Software. Intuitively speaking application software seems to cover a large area of the entire domain of software engineering, making it reasonable to be studied for itself. Business

applications such as Insurance, Banking, Financial, Real Estate and so on are some of the obvious areas included in this domain. **By Application Software Engineering (ASE) then we mean the branch of software engineering that deals with the automation of domain of problems requiring only the layer of abstraction entirely contained within a universal language.**

In what follows, we will investigate the degree of coupling between an abstract language and its virtual machine. Then, we illustrate the notion of semantic automation. By the way, ASE will refer to Application Software Engineer, as well.

Relationship between Language and Machine

Suppose, for a moment, that we have reached the conclusion that some form of abstract language is essential for application-level programming. The natural corollary to this conclusion is the existence of a virtual machine to automate the abstraction offered by this language. So, let us investigate the level of coupling between the language and its virtual machine.

The design of BASIC is such that, every new feature for the language promotes a corresponding change to the virtual machine. This is a consequence of interpretation as opposed to compilation. For instance, it is rather formidable to **define** a template class in an interpreted language, then instantiate it at a later point. A compiler/linker combination, on the other hand, is more suitable for abstract definitions like classes, inheritance, templates etc.

Smalltalk partially solves the problem of defining something, and later using the definition. It does so by adding programmer's definitions to its Dictionary. However, as far as templates, you are really stuck with whatever comes with the Smalltalk Dictionary. Furthermore, Smalltalk execution is inherently single-threaded, and the design was not intended for distributed computing.

JAVA, and its virtual machine JVM, is an example of a compiled language running on a virtual machine. That is in the right direction, without taking the step. Like Smalltalk, you get a set of container classes but you cannot define your own templates. The language allows threading and is intended for distributed computing. Yet still, it shares the same problems with BASIC and Smalltalk. The JAVA compiler seems to be doing little more than the BASIC interpreter in mapping the statements in the language to the modules of JVM. Thus, minor changes to the language require changes to JVM, which is rather enormous for what it does.

Increasing the size of virtual machine by a large factor for every new linguistic feature is definitely not on the right track. Even from a practical standpoint, new language features will outdate the machine too frequently. Instead, one should design a virtual machine that supports the language via a small number of instructions, and the techniques of translation. **There should not be a one to one correspondence between a feature in the language, and a module in the virtual machine for that language.** The machine can become smaller and faster without any effect to the language, and the language can absorb new discoveries with scarce need to modify the machine, if any.

A short digression concerning pointers is in order. Pointers were unheard-of when BASIC was designed. At any rate, both BASIC and Smalltalk were meant for educational purposes. JAVA,

regardless of how it skirts its lack of support for pointers, should not have left them out. Actually, pointers are equivalent of mathematical variables, not a notion to simply dismiss.

According to Tarski one can write the entire mathematics without ever using a single variable. To illustrate this, consider the set of programs that can be written in JAVA. Now, Tarski is saying that, adding pointers to JAVA will not enable you to write more programs. This is only of theoretical interest in that I doubt any mathematician can live without variables, as some do without certain forms of Axiom of Choice. **Perhaps pointers do not increase expressive power** (ability to state more programs), **but they definitely increase expressiveness** (how one states programs).

Degree of Automation within the Language

In order to motivate the notion of semantic automation, let us begin with an analogy. Consider a system of linear equations resulting from mapping some problem to a mathematical model. At this point, we believe we have solved the problem, although it is really the solution to the set of equations that is of interest to us. Our belief stems from the fact that the process of solving a system of linear equations has already been automated, somewhat as part of semantics of the system.

Now consider the case of copy constructor in C++ with regard to classes with pointer members. In this case, the solution is that the assignment operator be treated same way as the copy constructor by the language, and that the language does whatever is decidable to the case of pointers. ***This is what has been done in Z++ where an ASE does not have to remember doing anything special regardless of nature of members of a class.*** An ASE does not need to make up for gaps in order to ensure correctness, just as an engineer does not need to prove the correctness of mathematical formulas.

When a certain research activity yields the addition of features to an abstract language, it must be complemented with the research that identifies and provides the required degree of semantic automation for the new features. Leaving it to the ASE to remember all kinds of details in order to use the new features is only acceptable during the evolutionary phase. If we leave it at that point, then alternatives would be to come up with other look-alike languages. **This however, defeats the whole purpose of having an abstract, monotonically growing language, resembling applied mathematics for an ASE.**

A consequence of high degree of semantic automation is that an ASE can map a domain problem to a solution in Z++ in a manner analogous to the way an engineer maps a problem to the formulas of applied mathematics. There is no concern about special implementation details as required for system programming with C++. An ASE does not deal with a specific hardware where constructs are expected to do no more than what the eye sees for efficiency reasons. On the contrary, **the constructs should possess automated background semantics ensuring correctness.**

When a specific case is not decidable, then the compiler can generate all kinds of helpful messages. There should not be any reason to write a large bogus program, and then put it through all kinds of defect-detecting software. **That procedure can result in having to redo a large portion of the code, perhaps creating more defects, and even worse, requiring new thinking at design level.**

The iterative process of software development is a pragmatic necessity, but not a substitute for making up for shortcomings resulting from the inadequacy of tools.

Dr. Z. August 2002.