

Topics

[Introduction](#)

[Automation](#)

[The language of Mathematics as a model for our approach](#)

[Extensibility](#)

[The role of Libraries](#)

[Virtual Engine](#)

[Derivation and Provability](#)

[The Science of Automation](#)

[The Choice of Paradigm](#)

[The notion of Variable](#)

[The notion of Object](#)

[The Assignment Statement](#)

[Slight Formalization](#)

Introduction

In this article I wish to discuss the need for an extensible language powered by a virtual machine. In doing so, we will also review the notions of specification language, and correctness proof among others. The language Z++ referred to in this article is a superset of C++. Z++ executables run on a virtual machine, which essentially requires a C++ compiler for porting.

For the most part, the article puts together already known ideas in a cohesive manner to provide rationale for the need mentioned above. The theme of article is scientific romance rather than a technical article. For instance, consider the following statement, and the argument for its validity as an illustration of the approach taken in this writing.

Proposition. Infinite memory requires infinite storage space.

Argument. Suppose we have established the notions of measure for space and memory. Now start with a finite amount of space holding infinite memory. Make a sequence of spaces each half of the previous term, but holding infinite memory. The sequence converges to space of measure zero holding infinite memory.

The point here is that, while the argument is quite vague, it is a lot more satisfying than a simple belief in the statement itself. At least it provides some general idea for a possible more precise argument to be designed by a mathematician. Obviously the argument is not valid when storage capacity depends on shape as well as on size of storage medium. However, we do not know whether the brain reshapes itself to absorb more information, or whether it is possible to do so. For memory chips the proposition seems quite reasonable.

Remark. Third person singular pronouns in this article are used without connotation of gender.

[Go back to Topics.](#)

Automation

The term **automation** here will mean reducing some form of procedure to a program. The procedure could be of computational nature, such as a solution for a particular class of differential equations, or otherwise a business procedure. However, the starting point is not the discovery of algorithm itself. Rather, by automation we mean an implementation of such an algorithm in the form of a program. This saves us repeatedly saying, "a programmatic solution to ...".

The sense in which automation is used here does not correspond to its usual use. In search of what can be automated one seeks algorithmic solutions to specific classes of problems. An algorithmic solution generally has mathematical characteristics, which can be used to prove its correctness, establish its time complexity, etc. However, an implementation of that algorithm in the form of a program depends on many factors most of which may not be related to the algorithm itself. An implementation is merely a concrete realization as an approximation to the algorithm. ***A program must be tested even if by some magic it can be proven correct.*** I would not fly an airplane that has been proven correct by chasing formulas on paper. I understand that no test will guarantee that the next flight will not result in a crash.

In modeling some business procedure so it can be implemented as a program we are not trying to discover new forms of doing business. Generally, the procedure has been used for some time. We try to remove vagueness so we can model it in the form of a program, i.e. automate the process. In doing so, algorithmic discoveries may spring up at a different level such as discovering and resolving deadlocks.

[Go back to Topics.](#)

The language of Mathematics as a model for our approach

We somehow came up with language for communication. Over hundreds of thousands of years we managed to divest some primitive concepts. Observing over and over that two entities next to two others makes four of them, we somehow were able to jump from the adjective two to the abstract notion of number two and the fact that "two plus two is four". From there, we eventually created formal subsets of our vague communication language. These subsets have now evolved into various mathematical formalizations.

The point we are interested in is that, **all mathematical subsets of our communication language constitute one large vocabulary, while at the same time each is being studied by one breed of mathematicians.** However, all of known mathematics is expressible using only the notion of set along with its axioms, as an interpretation of first

order predicate calculus. The only mathematical element needed to construct the rest is the empty set. We do not intend to travel so far, of course.

Mathematical abstractions in various subsets, is the power of mathematical formalism for modeling the physical world. For each subset, one has a well-understood sub-language to think in. The language of Linear Algebra is not adequate for dealing with the notion of limit. Instead, one uses Calculus. In the same vain, the expressive power of a programming language should be separated from its computational power (say, the power of Turing machine). One should be able to think in terms of queries without regard to other linguistic abstractions, for example.

A language with well-understood subsets allowing one to think about certain classes of problems is not a mess, unless of course we consider mathematics a big mess. There is no need for one single programmer to understand the entire language in detail. This is better than constantly having programmers to move to yet another language. For instance, when one moves from a big city to another, the driving skills go along. But the streets and locations of stores must be learned over. Now if one travels to another country then even the types of stores will be different. It will take a while to know where to shop for certain items.

[Go back to Topics.](#)

Extensibility

Since we are advocating language extensibility, a BASIC programmer may happily announce that BASIC is an easily extensible programming language, and so we need not search anymore. Well, extensibility here does not mean adding more keywords to linear algebra so it can also deal with the notion of continuity. As with any language, there will be times that newer more general constructs may replace some older ones. Eventually, the older forms will be dropped. However, there must be some scientific reason to do so for an automation language. At any rate, **by extensibility we do not mean extending BASIC vocabulary to the size of Webster Dictionary.** Nor does it mean creating an object-oriented PL/I.

The point here is that, we wish to converge the expressiveness of language to various limit points as domains of application of automation. We would like to do this very much as it is done in mathematics. When solving a problem in calculus we can easily switch to linear algebra without concern to incompatibility. We do not want to solve one part of an automation problem in BASIC, another in COBOL, APL and on and on, and then use some form of gluing technology to put it all together. The gluing and wrapping technologies are needed for what has already been done, though. Let us consider some simple examples of extending C++ linguistically, as is done in Z++.

As is done in C, one can use null-terminated character arrays to represent ASCII strings, and then provide a set of library functions for their manipulation. However, it is far better to allow the language support ASCII and Unicode strings with proper conversions. The

operator += is much easier to use for concatenation than a function call such as strcat. Note, however, that string type has stood the test of time for being useful.

APL brings up another interesting idea with regards to arrays. The term array is used for the notion of defining new types as well as instances of those types. Z++ makes this distinction more noticeable. For instance, consider the following declarations.

```
long X[5][8];  
short Y[5][8];
```

In Z++ the array objects X and Y are of the same array type and are therefore **compatible**. Using coercion, one can do "X += Y;" with obvious semantics. This may superficially seem like throwing APL into Z++. However, Z++ provides operator overloading. Suppose the type of cell of array X is more complex which may have overloaded operator +=. In that case, Z++ will in fact use the overloaded definition of operator += for each cell of the array. This illustrates the principle of orthogonality in extending a language.

For a long time the notion of thread was associated with operating systems. Thus, one would use such things as fork and join to deal with processes in UNIX. Z++ provides one possible abstraction via the notion of task, discussed elsewhere. **The degree of need for use of system calls to an operating system is a measure of lack of expressiveness.** A solution to an automation problem sprinkled with system calls is essentially useless. Such a solution depends heavily on future changes to the underlying operating system and is unlikely to be transportable to a different computing environment.

[Go back to Topics.](#)

The role of Libraries

Certain domains are best dealt with via libraries as opposed to extending the language itself. Unlike strings, arrays and database notions, it is not easy to come up with proper abstractions for notions relating to input and output. **A language must deal with abstractions, not specific actions depending on a particular technology.** In particular, the devices and IO interfaces will continue to change making library approach more attractive. The IO library will inevitably involve system dependencies, though it must contain a slowly growing set of abstractions. Essentially, an abstract subset of graphical entities for a GUI system has been defined. Nevertheless, such entities are better treated via libraries.

Knut is responsible for the introduction of notion of abstract data types, such as stack. In mid seventies, more extensive research was done on this topic. Parna's idea of hiding information paved the way for automating the concept of Abstract Data Type in the form of a class. Finally, C++ templates made it possible to create reusable templates of abstract data types. Smalltalk included such templates as part of the language, while for C++ we

find MFC, ATL, STL and various third-party vendors of template libraries. So, which approach is better?

In general, the solution to an automation problem should be based on abstract data types. However, we are not simply interested in some solution. For instance, we sometimes need a stack that can grow as big as the physical memory allows, and other times we need a bounded stack with faster response time. Furthermore, the emphasis on the use of abstract data types implies the discovery of more forms of such abstractions. Thus, it seems that providing a template mechanism for building libraries of abstract data types is more appropriate than packaging a set of such types as part of the language. However, the template library for abstract data types should grow slowly and remain stable as an integral part of the language for solving automation problems, very much like Smalltalk.

The main point to keep in mind in extending a language is its power of abstraction. A language is a mechanism to express a **solution** to an automation problem. The process of solving the problem is the **methodology** used to analyze the problem so it can be expressed in an automation language. The user interface and input/output mechanisms depend on the available technology and are treated better via libraries.

[Go back to Topics.](#)

Virtual Engine

To achieve Knut's utopia for a worldwide distributed computing, we need an extensible platform without dependence on hardware. The lack of dependency facilitates distributed computing performed by virtual machines communicating via traveling object without regard to underlying technology.

What I am going to say is not quite for our time. As an analogy, I personally would not create applications for a processor for which a C++ compiler does not exist. **I simply do not have enough number of extra years within my lifetime to spend it on learning a handful of languages just as well as I know C++.** I already have to deal with a great number of environment-related idiosyncrasies as I move from one PDA to another. With this analogy in mind, let us proceed with the proposal.

At first, by virtue that there is a C++ compiler for almost any platform, one may conclude that all we have to do is to drop threading from Z++ and still have a superset of C++ to work with in any platform. However, a look at Palm OS 4.0 reveals the kind of problems we can run into. Once the virtual machine is launched, it is not easy to launch another application to run on top of it.

Not all linguistic capabilities need to be supported by a minimal new platform. However, **everyone will benefit if platforms are designed to support some virtual machine rather than the other way round.** Anymore, no developer would go for a platform that only supports some proprietary language, regardless of the elegance of the language and its versatile support for the particular environment. As I mentioned earlier, I personally

expect no less than an ANSI C++ compiler. So, the proposal is to extend this notion to supporting some standard virtual machine. The challenge will be in agreeing on the subset of language libraries dealing with GUI. In some cases we may not need to reduce the library. For instance, **a window does not have to mean exactly the same thing on a PC as it does on a PDA**. Obviously, we are speaking of a platform that allows the development of reasonable applications by third parties.

[Go back to Topics.](#)

Derivation and Provability

Since the name Z++ comes from Z specification language and C++, one might think that Z++ connotes a specification language that can be compiled. This is like wishing that one merely specifies an algorithm and suddenly gets an implementation for it. A logic programmer may now get overly excited by recalling that he can do something like that for, say sorting. The problem is that, **if a specification language can be compiled, directly or indirectly by mapping it to a programming language, then itself is a programming language**. The discovery of Resolution Principle for automating a portion of logic is a great achievement. However, this is an application of automation, not an end for automation.

An automation language need not be concerned with correctness proof. The expressiveness of a language to detect and deal with unusual situations is a form of automated correctness guarantee. According to Halting Problem we cannot automate correctness proof, unless we restrict ourselves to Kleene's work before he extended it to generalized partial recursive functions to match up with Turing Machine. **Correctness guarantee** is more suitable for automation modeling. One relies on tested components and their invariants or preconditions to devise run-time remedy other than abort (or crash) to a flying shuttle.

As another problem, suppose some form of logic can be used to prove correctness of a program as objects transform overtime (we are stuck with the curse of Von Newman). The question is, how does this proof guarantee that at run time the transformations will keep objects within considered bounds? I am not saying that programming is an ad hoc activity. **In fact, the whole idea of extensibility is approximating the notion of correctness by converging the language to its problem domain**. Then, correctness guarantee can provide **some degree of certainty** that appropriate actions will take place should specified conditions fail. **We should differentiate between mathematical modeling of deadlock detection using Graph Theory, and an implementation of the algorithm resulting from such modeling**.

A person trained in reducing certain problems to a set of linear equations cannot prove that his equations correctly model the reality. *What can be proven is the correctness of solution for the set of equations*. **But automation modeling ends at writing down the equations**. Suppose you have automated a problem by mapping it to a program. So now what is it that you wish to prove correct? **What is the solution to your program so you**

can verify its correctness? So, how do you prove an airplane is correct, given that all the computations done to make it are in fact correct?

[Go back to Topics.](#)

The Science of Automation

The confusion about correctness proof starts as another curse of Von Newman, when he remarked in surprise as to how all of his code is in fact correct. The comment is along the lines of a similar remark by Einstein about mathematics itself.

Looking back at our history of science, we find that in the beginning there was mathematics. Then, gradually other sciences accumulated sufficient flesh to live on their own. However, the essence of science never really changed in that the mathematics of each era is the mirror reflecting the scientific advances of that era. It is the ultimate goal of any science to be formalized in such a way that it can be cast in a mathematical framework. Maxwell's work on electricity, Von Newman's treatment of quantum Mechanics, LaGrange's Dynamics, Newton's, Einstein's, and on and on are classic examples. So, why not do the same with automation. After all, Newton had to improvise Calculus, so perhaps we need to create some form of new mathematics.

First, let us observe that, while mathematics is the vehicle of proof for all other sciences, none of them can help mathematicians to derive any mathematical statement. Now note the parallelism that, while automation is the means of automating even mathematical computations, nothing can help Backus implement a FORTRAN compiler other than automation itself.

At this point one can think of such things as Formal Language Theory, Knut's work on Bottom up Parsing, and a whole lot more. Keep in mind that the scientific work to make an airplane is not the same as the airplane itself. In the same vain, a compiler is not the same as all the theory on which it is based.

What I am trying to say is that, Automation is a science apart from its origins. Theory of Computation is a branch of mathematics rather than something like Theoretical Physics. **It is the leap from Theory of Computation to Automation that has been so hard to accept.** We all want to be able to come up with some programming language that allows correctness proof simply because programming statements look very much like mathematical formulas. Automation is a science like Statistics in its infancy.

[Go back to Topics.](#)

The Choice of Paradigm

A programming environment is a measure of degree of abstraction that has been **automated** towards solving (i.e. automating) a particular class of problems. These environments are conceived as a result of our desire to widen the scope of automation

and are represented by a language for expressing solutions to problems. At the dawn, the entities considered for automation were all too well understood. After all mathematics has been with us for quite sometime. At any rate, an automated abstract medium (essentially a programming language) is a scientific creation, rather than an accidental consequence of random unrelated attempts. Let us briefly trace the history.

The process of abstraction and modeling is an exclusive function of brain with its unlimited capacity for imagination and creativity. Ironically, the brain is very awkward in manipulating the objects of its own creation. We can quite easily map a problem to the set of integers in our head. However, it is not so easy to add several numbers without pencil and paper. Thus, the initial phase of automation was essentially the creation of means for number crunching via abacus, calculators and FORTRAN. The next phase of automation was to automate a mathematical model such as Linear Algebra. The most significant step was taken with the realization of the fact that automation can solve its own internal problems. The trend begins with assemblers, followed by compilers and databases and continues to our day. The concept of a virtual machine is a sophisticated form of the notion of a BASIC interpreter. Of course compilation as opposed to interpretation is the translation mechanism. One can also recall Knut's virtual processor and similar virtual engines, JAVA being a recent attempt.

The notion of Type is the first major step towards making relatively large software. The ability to define rudimentary types as part of a program (COBOL records), in conjunction with the work of many researchers resulted in our ability to define new types within a program in a manner analogous to those built into the compiler as part of the language. The idea is simple and effective in making large software of our time automating many processes, including business operations. This notion is treated elegantly by the class construct of C++, which also took the next step of defining type templates, or parameterized types.

One maps a problem to an automation solution by expressing it in terms of interacting instances of types, or **automation objects**. Automation objects become virtual representations of actual entities, approximating the real entities by abstracting only their relevant characteristics. The degree of approximation depends on our decision as to what characteristics of the real entities (physical or conceptual) are relevant. Object-Oriented techniques allow successive approximations to almost any desired degree of accuracy.

Let us, for the sake of comparison, take a look at a familiar technique used by physicists in their modeling of nature. A few entities, such as Length, Mass and Time are taken as the basic dimensions from which others can be derived. For instance, speed is distance per second. The mathematics used by physicists in establishing relationships is at a much higher level of abstraction than the formula for the area of a triangle. Nevertheless, the principle is the same. A physicist maps nature to physical notions in such a way that relationships can be formulated using the basic dimensions.

The technique from physics was given to motivate the notion of built-in, or **fundamental** types from which one creates new types through some definition-mechanism such as

class. **The discovery of automating the construction of new types is a truly remarkable invention**, perhaps second only to the idea of stored program. The actual process of reducing a problem to a form that it can be automated through the abstraction provided by a language is still evolving. A top-down functional decomposition combined with some intuitive form of mapping the real entities into automation objects seems to be the main contemporary theme. Based on empirical success of scientific research of several decades, the language Z++ provides an abstract automation environment via the notion of types and object-oriented mechanisms of C++, and by adding many desirable extensions. It is important, however, to keep in mind that Z++ does not rival C++ by virtue of fact that its programs run on a virtual machine written in C++.

[Go back to Topics.](#)

The notion of Variable

In "Logic for Mathematicians" Rosser provides an exhaustive treatment of the notion of variable, while Tarski tells us that we do not even need variables anywhere in mathematics. Turning to automation we find unusual terminology arising from our desire to maintain our mathematical terminology apart from such things as memory locations.

Within an abstract automation language the equivalent of the notion of variable is Pointer. The difficulty to surmount is to drop the view that a loop counter such as X is a symbol that can range over a given set of values. This in turn requires a deeper shift of view. Thus, we first analyze the relationship between an (automation) object and its literal representation, and then attend to the notion of pointer.

An object is a virtual entity representing an abstraction of some (relatively) real entity. An object, just like the entity it represents, occupies space in its virtual domain of existence. It needs the space partly for its literal representation, which at language level is referred to as the **state** of the object. We observe that the literal representation must be retrievable in the form of state of the object. Thus, an object occupies space, which must be reachable. It so happens that the space occupied by objects is conceived as a linear memory, and obtaining the state of an object is generally achieved by accessing the **address** (coordinate) of start of the literal representation of that object.

The state of an object corresponds to the state of the entity it represents. For instance, suppose Height is an object representing the height of some tree. As the tree grows (in a simulation or for real), Height must maintain its correspondence via the abstraction of some literal representation, in this case perhaps mathematical integers relative to some unit such as centimeter. Otherwise put, objects are not variables ranging over some values. They are entities that can change in their virtual world just as real entities do. One does not say that the height of a tree is a variable ranging over some set of values. A tree can change in many ways among them it can grow. Quite analogously, a loop is like a tree, and the loop-counter-variable is an object representing its height (number of iterations). Keep in mind that being a real object (such as a tree, or a loop) is relative to the domain under consideration.

Let us now analyze the nature of pointers by looking at a simple case of notion of variable as used in mathematics. Consider the statement, "Let n be an integer". Here, n is understood as a variable ranging over the set of integers. Compare this to the statement, "Let X be a pointer of type A ", where A is some user-defined type, for instance. What this means is that, X can be (i.e. point to) any instance of type A in the virtual world depicted by linear memory or some other abstraction. It is not relevant at language level that a pointer must take on different address values in order to point to these objects. What matters is that, **the pointer X ranges over a domain consisting of instances of type A** , even though the entire set of objects in this domain is not specified as apparently was the case with the set of integers.

Going back to Tarski, it is possible to avoid pointers altogether. However, the expressiveness achieved in allowing us to deal with late binding and similar notions requires the ability to consider pointers to pointers, etc. We will have to say more about pointers. For now let us observe that the notion of object is not limited to instances of user-defined types. Documents and database records accessed across Internet are examples of objects, which use different addressing mechanism than linear memory.

[Go back to Topics.](#)

The notion of Object

Logic and functional programming languages are purely literal-oriented. Symbols are instantiated with specific literal values as opposed to being assigned to. This is because a symbol such as X is not associated with a virtual entity that occupies space. In other words, one does not think that X represents a certain memory location where one can store its value, or change it via assignment. Generally imperative languages such as Ada are somewhere in between. For instance in " $X := Y$;" the identifier X is referencing a memory location, while Y stands for the literal representation stored in location referenced by Y .

In contrast to literal-oriented languages, an identifier in an object-oriented language is in fact the **object itself**, not a name for some location, or some concept. Obviously, this contradictory statement needs quite some elaboration to make any sense. After all, statements can only include names of entities, not the entities themselves. When looking at " $X = Y$ ", a mathematician thinks that X and Y are two names for the same entity.

An object is a reflection of an entity through the mirror of abstraction. An automation of part of reality is a mapping of reality into a virtual form of existence. The purpose of this mapping is to deal with certain aspects of real entities and their interactions under a well-defined set of laws, which is a controlled subset of physical laws. Shifting our attention to a real entity such as a tree, one can easily accept that the actual tree is not a name for something. We use a symbolic representation when making statements about a real entity. In " $X++$;" the object X , and not its name, is being told to grow by one unit in a manner similar to watering a tree so it will grow.

Another way of looking at this matter is to consider the fact that when a tree grows it is not mapped to a different tree. Rather, some aspects of the same tree change. In the same vein, methods such as increment-operator cause objects to change, even though the successor function is a map from the set of Natural numbers into itself. Once we accept that an object is an entity that occupies space it becomes easier to accept that it can also change in its virtual world. We shall discuss the distinction between a method and a function at a later time.

Going back to our contradictory statement, we know a programming language is treated as a language so it can be translated to machine instructions via a compiler. Furthermore, one really wishes to think in terms of a language when devising an algorithm. Then, an implementation of that algorithm in an automation language should feel the same way. On the other hand, a mathematician solves a problem by visualizing certain mental images. This is even more so for automation problems. Instead of thinking in terms of statements in a language, one visualizes objects very much the same way that we watch TV. In fact, the actual entities are visualized so much as it is possible. The solution in an automation language is a description of this visualization in such a way that when looking at statements one still imagines the associated mental images. The best conclusion that can be drawn from these observations is that an automation language has dual nature. **It is an ordinary language except when it begins execution in our head, or on a machine.**

[Go back to Topics.](#)

The Assignment Statement

A novice programmer will probably use the assignment statement more wildly than goto. That is not our concern. But, we do need some mechanism in order to cause an object to change (its state). This again sounds like a contradiction. On the one hand we wish to think that an object is capable of changing, such as growing somewhat on its own, on the other hand we are speaking of making direct assignment to it.

We send our children to school with the expectation that they will learn something new. It never occurs to us that the teacher makes a knowledge-assignment to their brain, though in effect that is what a teacher strives for. After all, we do not send children to school to get them back with the mentality of an adult in just one day. Nevertheless, the goal of sending them to school is the change that we hope to happen to their brain. How the assignment of knowledge is executed depends on whether Skinner's theory is followed or Piage's.

Essentially we are saying that, when visualizing an object we imagine that it grows. But the mechanism to affect the growth is assignment of a different literal value, which signifies getting bigger or taller. Whether or not a programmer abuses the assignment statement, is an entirely different issue. Some programmers will never grow beyond solving automation problems by juggling code and trying it out. Some environments

encourage that attitude by providing thousands of system API, essentially as a requirement for implementing anything at all. In general, such implementations are randomly sprinkled with system-specific calls and at best are not abstract enough to qualify as a solution. **The difference between a programmer and an automation engineer is the same as the difference between a poorly educated person and a writer in their usage of a spoken language.**

[Go back to Topics.](#)

Slight Formalization

Without getting too far into it, let us try to formalize a few notions. Initially, we assume the existence of some coordinate system with a countable set of (coordinate) literals, to which we refer as addresses.

The notion of type is a primitive quite apart from its set of literals. For instance, the type integer is a notion of automation, which is closely related to the mathematical concept of set of integers. The difference is not merely in cardinality. An element as an instance of type integer is an object that occupies space and can change, whereas an element of the set of integers like 2 is just that symbol.

An object can be defined as a triplet (A, T, l) where A is the address, T is the type and l is the literal representation of the object. Of course l is an element from the set of literals L associated with type T . Note that two distinct objects can be of the same type and have the same literal value. Ideally, two different objects should not reside at the same address, which except for some uses of unions is generally the case.

Consider the sets D , R and S . An automation global function is a pair of functions (f, g) such that f maps D to R , and g is a map from S to S . The component g is the side-effect function, which ideally should be the identity map. A method has a third component h that maps the set L of literals of the type of object to itself.

For a proper formalization one needs the equivalent of the empty set for types. This is filled in by type `void`, which admits no literals and therefore no instances of this type can be created. The address literal `NULL` plays a similar role for set of addresses. No object can be at address `NULL`.

Let us use the term subtype of a type to mean that instances of the subtype can only take on a subset of literals of the type for their state. The **void pointer type** is the type that has the set of addresses as its literals. A pointer to a specific type T is a subtype of void pointer type.

Consider an object $O = (A, T, l)$. The literal value A for an object O will be indicated by $\&(O)$. For any object O , a predefined function (as opposed to method) called taking-address yields $\&(O)$.

A **pointer type** (as opposed to void pointer type) has a predefined function known as dereferencing, denoted $*$. Suppose for object $O = (A, T, l)$, and pointer $P = (B, T, m)$ the equality relation $m = A$ holds. This means that $\&(O) = m$. Then, $*P = O$. **It is important to keep in mind that $*P$ is the object O and not the literal representation l of O .**

Let V indicate the type void, and consider the pointer $P = (A, V, l)$. The operation $*P$ is undefined because the resulting object will have to be of type void. Similarly, for any pointer $Q = (A, T, \text{NULL})$ the operation $*Q$ is undefined because no object can be at NULL address.

The sense in which the set of integers is imbedded in the set of rational numbers is quite different from that of a group being imbedded in a ring. In the former, every integer qualifies as a rational, and in the latter, every ring is also a group, which goes the opposite direction. When dealing with types we usually focus on the set of literals of a type, which in turn shifts our attention to sort of imbedding of integers in the set of rational numbers. This could be a side effect of assignment in our way of thinking, or perhaps a consequence of the fact that the derived class usually has more members and as a result, a larger set of literals. However, the notion of **derivation** (also known as inheritance) is closer to the imbedding of a group in a ring.

The imbedding of a group in a ring allows us to treat a ring as if it were nothing more than a group. Suppose we could create an instance R of a ring, and an instance G of a group. It does not make sense to perform the assignment $G = R$, or the other way round. However, if P is a variable, i.e. pointer of type group, we could let it point to the ring R . This is because all we can do with the object $*P$ is group operations, after all.

In the realm of automation we allow for a slight generalization of the preceding situation. Suppose that the operations of ring R that coincide with those of group G have no mathematically sound relationship, such as function restriction. However, let the underlying sets be the same so that it makes sense to apply operations of group G , or the group operations of ring R to those same elements. Then the result of operations on $*P$ will not be known by looking at the statements of the language. One refers to this situation as polymorphism, and the technique used in its implementation is known as late binding. **For all that matters any pointer of a given type T can point to any instance of all types derived from T .** Therefore polymorphism does not specify a particular kind of pointer and is not a notion to be defined. For historical reasons polymorphism and late binding are needed to distinguish between pointers in contexts of C and $C++$, and similar situations. The same holds for the term virtual methods of $C++$. None of these terms are relevant in the context of $Z++$, though.

Formalizing the notion of derivation will serve us little purpose within the scope of this article. We reiterate the intuition needed to visualize solutions, instead. So, let R be a type derived from type G . We will think of G as imbedded in R along with the extension that methods of G could be redefined in R . When an instance Y of R is constructed, then an instance X of G is also constructed (actually before the construction of Y), and X is to be visualized as imbedded in Y . Finally, a pointer of type G can also point to Y without

causing any confusion for the compiler, which is a good corollary to establish in a more formal setting.

Before closing I wish to say a word about multiple (none-linear) inheritance. The automation of this type-construction mechanism is far beyond plain luxury. In languages limited to linear inheritance solutions are generally too far from the problem they solve thereby impeding the visualization of the solution. Let us consider an example from mathematics.

A vector space contains a group and a field along with an operation involving elements of each structure. It can be taken for a group, a field or a vector space. By adding a suitable inner product we can also think of it as Hilbert space or just plain metric space. It is not clear how to define a vector space within the limitations of linear inheritance.

I would like to remark that the construction of a vector space is not similar to including members in a structure. The latter was illustrated by citing the technique of dimensions used by physicists. Using the term method we can phrase the difference as follows. The definition of the method scalar-operation does not use the definitions of methods of the group or those of the field.

[Go back to Topics.](#)

Dr. Z.