Programming as an Engineering Discipline

Juris Reinfelds[1]

Abstract *— For too long computer programming has been treated as an art or a craft rather than as a science or an engineering discipline. The Kernel Language approach provides a precise and concise basis for programming in all paradigms (imperative, logical, functional and object-oriented) as well as for parallel, concurrent and distributed multi-thread programming. The Kernel Language is implemented as a subset of Oz, a powerful, multi-paradigm programming language that is similar to Java. This allows us to apply the theory to enhance the art of practical problem solving. KL allows us to introduce multi-thread programming and the major programming paradigms in first courses of programming. With the rapidly expanding acceptance of multi-language programming capabilities of dotNET, a revision of traditional introductory programming courses becomes more and more important.*

Index Terms — multi-paradigm programming; net-centric programming; programmer's theory of programming; teaching of multi-paradigm, multi-thread programming;

INTRODUCTION

A craft becomes an engineering discipline when there is a theory, which practitioners of that discipline can use to predict the results of their actions with reasonable ease and reliability before they actually perform these actions. For example, the theories of statics, dynamics and strength of materials allow engineers to develop and test a bridge design, to optimize that design for economic or aesthetic considerations or we can use these theories to understand and analyze someone else's designs in reasonable time, with reasonable effort. To see why this is not so in programming, we have to look at history.

Large-scale engineering and scientific programming started with the introduction of the programming language FORTRAN [1]. The syntax (visible form) of FORTRAN statements was informally described in the FORTRAN Manual, but the semantics (what actually happens when statements are executed) was not defined anywhere.

To find out, we had to use the FORTRAN compiler as an oracle – compile and run the program and observe what happens. While there was only one FORTRAN compiler, it was not so bad, but soon there were many slightly different FORTRANs, because mainframe manufacturers encouraged their compiler writers to "trump" the so far undefined semantics as well as the loosely defined syntax of all other FORTRAN implementations. Without a theory that programmers could use, programming developed as an art [2], and as a craft [3], but not as an engineering discipline.

Perhaps because of such casual beginnings, engineers saw computers as super accurate, super fast slide rules, the use of which can be self-taught over the "proverbial weekend" rather than as an essential engineering discipline that needs to be developed as a key branch of software engineering.

Today all branches of engineering use software tools that permit the solving of specific problems of a problem area covered by that is covered by that software tool, by clicking on parameter values that define a specific problem, but there is little conce rn that the engineering discipline of programming, that would specialize in the design and implementation of software tools and software in general is still missing and software is still developed in an informal, ad hoc manner.

Computer Science had its beginnings in mathematics. It adopted the theory of computability that mathematics had long before computers were invented as THE theory of computer science. Programming was explained with mathematician's concepts such as Turing machines, lambda-calculus and pi-calculus, with concepts that do not arise in the day-to-day work of practicing programmers.

1. Juris Reinfelds, New Mexico State University, Klipsch School of EE & CE, Box 30001 dept 3-0, Las Cruces, NM 88003 juris@nmsu.edu

However, a theory is most useful in practice only if the practitioner can reason with concepts that are already familiar in the day-to-day work of the practitioner.

The first *"programmer's theory of programming"* was introduced by Edsger W. Dijkstra[4]. At that time imperative programming was the only important programming paradigm. Therefore Dijkstra's theory covered the essence of imperative programming. The core of Dijkstra's theory is elegant and simple. First he observes that a *program* is a *sequence of statements.* Second, he observes that wherever a statement appears in a program, we can replace that statement with a sequence of statements. Programmers use this idea all the time when they write or edit programs. Formally we capture this concept in Backus-Naur-Form (BNF) as

<sta> ::= <sta1> <sta2>

which reads: a sequence of statements, *statement1* followed by *statement2*, behaves in our language exactly like a single statement. According to this rule, a sequence of statements of any length *"is a statement"*. One consequence of this rule is that it allows the concept of a procedure. According to this rule, we can take any sequence of statements out of a program, place them in a procedure and place a one-statement procedure call in the program to achieve the same effect.

Here is the syntax of the core of the statements that Dijkstra's theory considers, expressed in a terminology that resembles C-programming language, but with less punctuation:

<statement> ::=

<statement1> <statement2>

<variable> **:=** <expression>

if <conditional expression> **then** <sta1> **else** <sta2> **fi**

while <conditional expression> **do** <statement> **od**

skip

To describe the semantics of these statements, Dijkstra uses preconditions and postconditions. Preconditions and postconditions are boolean expressions in the variables of the program. A statement (which might be a list of many statements) is guaranteed to perform its task correctly if and only if its precondition yields *true*. The *"task of a statement"* is captured in its postcondition, which becomes *true* when the statement completes its task. For example, if the square root function in the statement *Y := sqrt(X)* works only for positive arguments, its precondition is *X>0.* The postcondition is *X == Y*Y*

A *while* statement poses two problems. First, we have to capture the semantics of the loop. That is, we have to describe, with a boolean expression, what the state of the computation is at the start of each iteration of the loop. Since the statements contained in the loop body do not change during the execution of the loop and this boolean expression does not change during the execution of the loop, we call it the *"loop invariant"*. Second, we have to make sure that each iteration of the loop makes progress towards the completion of the loop. Otherwise the loop will run forever. Progress towards completion is made explicit by defining a positive-valued, monotone decreasing function of loop variables that measures progress towards completion of the task of the loop and denotes completion when it value reaches zero.

Dijkstra's theory had a strong impact on first-course teaching at a number of locations e.g. CalTech [9], Griffith University [8] and a profound impact on programming of small but difficult algorithms [10, 11].

THE KERNEL LANGUAGE APPROACH

A Kernel Language should define the essential concepts of programming without any unnecessary complexities. A KL should be Turing complete so that any program can be expressed in it. A professional level implementation of a KL should exist so that theoretical discussions can be supplemented with laboratory exercises. With such a KL we can teach key concepts of programming without getting bogged down in the complexities of full-size programming language syntax and semantics.
The Kernel Language described here was introduced by PeterVan Roy and Seif Haridi [5]. It captures the essence of programming in all major paradigms, using concepts that are familiar to programmers. The KL is designed in a modular fashion, start-

ing with a *Core Kernel Language* to which we successively add new concepts that extend the capabilities of the language. This KL is a simple, precisely defined subset of the multi-paradigm programming language Oz [6].

The KL is a *comprehensive subset* of Oz, in the sense that every statement and data value of the full programming language Oz, can be expressed in terms of Kernel Language statements and data types. This allows us to strengthen the teaching of the art of problem solving with a better understanding of the key concepts of programming, provided by the Kernel Language while using the convenience of the full language for programming of problem solutions. Once students learn how to link Kernel Language concepts to Oz, they can rapidly learn to see kernel language concepts in Java, C++ and other commercial languages.

Kernel Language programs can be compiled by the Oz compiler and executed by the Mozart-Oz runtime system. Theory becomes more accessible if one can check one's intuition at each step by running a program, to see if the desired effect was achieved. In this way we have a practical hands-on way to apply our theory. That is why we call it a programmer's theory of programming. We no longer have to rely entirely on the axiomatic basis of mathematical structures as in the theory of computer science.

By exploring the behavior of Kernel Language programs we can strengthen our trust in and understanding of the definitions and concepts of larger programming languages. In short, the Kernel Language is a promising candidate for a theoretical basis of the engineering discipline of programming.

Since the introduction of dotNET [7], which is currently under research and development by Microsoft, IBM, HP and others, we now know that program components written in languages as diverse as FORTRAN, COBOL, Eiffel, C++, C# and Java can be intermixed in one program system at various levels of interaction, as long as the compilers for these languages obey the rules of the dotNET platform. Because Oz captures the essence of the major programming paradigms as well as multi-thread computing, the Kernel Language is an excellent starting point for a programmer's theory of programming for the net-centric multi-platform, multi-language programs that will become commonplace through the evolution and exploitation of the dotNET platform.

The Syntax of the Kernel Language

The full syntax of the Kernel Language and of Oz is in [5]. Here we illustrate the layered structure of the Kernel Language and the simplicity of the concepts that constitute the essence of programming. Here we will discuss some of the key concepts of programming that are hard to teach when hidden behind or restricted or distorted by traditional complexities of conventional programming languages.

Any Statement is a Program

The Kernel Language and Oz do not have a main program and subprograms. Instead, any sequence of statements is a program. Execution starts with the first statement of the sequence and terminates with the completion of the execution of the last statement. As in Dijkstra's theory, a statement may consist of a sequence of statements. These concepts are captured by the syntax

<program> ::= <statement>

<statement> ::= Statement sequence

or in more formal Backus-Naur-Form:

<program> ::= <sta>

<sta> ::= <sta$_1$> <sta$_2$>

The Core Layer of the KL

Now let us add variable creation as just another statement, not as a *declaration* with special properties that declarations have in conventional languages

<statement> ::=

<sta_1> <sta_2>

local <var> **in** <sta> **end**

where the *local statement* creates a variable with a well defined scope, but with no value or type. The variables of KL are assign-once dataflow variables that, once bound to a value, retain that value and its type until the end of the computation. Variables may also be bound to each other, provided that such binding does not equate unequal values that may already be bound to the variables. Hence the syntax

<statement> ::=

<sta_1> <sta_2>

local <var> **in** <sta> **end**

<x> = <value> //var to value binding

<x_1> = <x_2> //var to var binding

As we will see later, this way of introducing variables gives us the simplest and most direct multi-thread synchronization mechanism that we can use to discuss, understand and study the process synchronization and locking mechanisms of programming languages like C++ and Java.

A *conditional statement*, a *pattern matching statement* and a *procedure call statement*, complete the core layer of KL. The core layer of KL unifies the two main declarative programming paradigms: strict functional programming and deterministic logical programming.

<statement> ::=

<sta_1> <sta_2>

local <var> **in** <sta> **end**

<x> = <value>//var to value binding

<x_1> = <x_2>//var to var binding

if <var> **then** <sta_1> **else** <sta_2> **end**

case <var> **of** <pattern> **then** <sta_1> **else** <sta_2> **end**

{ <var> <arg_1> ... <arg_N> **}**//procedure call

Why is a procedure declaration not specified? Because a procedure in KL is a "first class value", hence procedure declarations are defined where integer and other value declarations are defined. Here is the syntax of most of the data types of values:

<value> ::= <number> | <record> | <procedure>

<number>::= <int> | <float>

<record>::= <lit>

|<lit>**"("**<ft_1>**":"**<var_1>...<ft_N>**":"**<var_N> **")"**

<proc> ::= **proc "{" "$"** <arg1> ...<argN>**"}"** <sta> **end**

<lit> ::= <atom> | <bool>

<ft> ::= <lit> | <int>

<bool> ::= **true** | **false**

<pattern> ::= <record>

We often hear the statement, that in some programming languages a procedure is a *"first class value"*. Exactly what does this mean? An integer is a first class value in most programming languages. The statement

X = 1230

specifies an integer value and binds the integer value to the variable X. The compiler transforms the ASCII specification of the integer into a more computer-friendly binary integer form. In the same way, the statement

Cube = **proc** {$ X Result} Result = X*X*X **end**

Specifies a procedure-value and binds the procedure value to the variable *Cube*. The compiler transforms the ASCII specification of the procedure into a more computer friendly byte-code form. Wherever we can bind or use an integer value, we can also bind or use a procedure-value. The procedure call is a statement, but the procedure declaration creates a procedure-value that may be bound to a variable like any other value.

Concurrent Computations

Concurrent and parallel computations require one more concept. A sequential program executes in a single thread of execution. Multiple execution threads are created by the *thread-statement*

<statement> ::=

... as above ...

thread <sta> **end**

The thread statement starts a new execution thread that executes the statement (program) enclosed by the thread-end brackets. The current thread continues execution with the next statement that follows the thread statement. With such a simple thread creation capability, we can immediately discuss synchronization and atomic actions that are at the heart of concurrent programming, but that are obscured by syntactic and semantic complexity of thread creation and invocation in languages like C++ and Java.

Error Management

Management of runtime errors requires the ability to discontinue the execution of the offending statement and to execute a damage control (e.g. error message) statement instead. The programmer should be able to "raise an exception" so that when the programmer's *raise-statement* is executed, it will trigger its corresponding *try-statement* to, for example, complain about unacceptable user input. In KL this is achieved with two java-like statements

<statement> ::=

... as above ...

try <sta_1> **catch** <var> **then** <sta_2> **end**

raise <var> **end**

Assign-Many-Times Variables

Conventional, imperative-language variables are called *cells* in KL. We say that cells add explicit state to programs in the sense that the outputs of a program are determined by the inputs and the state of the cells (their current values) of the program.

<statement> ::=

... as above ...

{NewCell C X}//new cell C with value of X

{Access C X}//bind X to value of C

{Assign C X}//assign to C the value of X

Object Oriented Programming

KL introduces object oriented computing by defining *class* and *object* as two new first-class-values. No further statement concepts are needed.

The Semantics of the Kernel Language

There are several formalisms for precise definition of the semantics of programming languages. Operational semantics uses concepts that are closest to the programming concepts used by practicing programmers. The Kernel Language uses operational semantics. A complete definition and discussion of Kernel Language semantics is in [5]. Here we include just a flavor of it, just enough to explain the example that follows.

KL introduces unbound variables and permits their use in expressions and statements. KL semantics specifies that if the evaluation of an expression requires the value of an unbound variable, the evaluation suspends until another thread binds the variable to a value. This makes no sense in one-thread sequential programming, but in multi-thread programming this creates a simple basic synchronization mechanism, which we can use to construct semaphores, locks, monitors, rendezvous and other widely used synchronization mechanisms. To illustrate how KL makes the essence of a problem solution directly accessible, we will program a rendezvous of several threads.

Programming a Rendezvous

Suppose that we have four threads, each executing a different task, but at a certain point, called a *rendezvous-point*, in each task, the computation must not proceed until all other threads have reached their rendezvous-point. For example, a sending thread may not wish to proceed until all receiving processes are ready to receive the communication. The statement

thread <sta_{before}> <sta_{after}> **end**

separates the statement sequences that have to be executed before and after the rendezvous. To implement the rendezvous we use the semantics of unbound variables.

Suppose that we have four threads. Number the threads 1-4. Introduce four unbound variables R1, R2, R3, R4. At the rendezvous point of each thread, except the last one, bind "its variable" with the variable that has the number of the next thread. In the last thread bind "its variable" to a value

thread <sta_{before}> R1=R2 <sta_{after}> **end**

thread <sta_{before}> R2=R3 <sta_{after}> **end**

thread <sta_{before}> R3=R4 <sta_{after}> **end**

thread <sta_{before}> R4=val <sta_{after}> **end**

Note that the variable bindings form a chain and regardless in which order the bindings are executed, R1 remains unbound until all four bindings have taken place and all four thread-executions are at their rendezvous points. Therefore each thread may execute <sta_{after}> only if R1 is bound to a value

thread <sta_{before}> R1=R2 **if** R1==val **then** <sta_{after}> **end end**

thread <sta_{before}> R2=R3 **if** R1==val **then** <sta_{after}> **end end**

thread <sta_{before}> R3=R4 **if** R1==val **then** <sta_{after}> **end end**

thread <sta_{before}> R4=val **if** R1==val **then** <sta_{after}> **end end**

where we have used Oz shortcuts (== for comparison and no else-clause in if-statement) to keep each thread-statement in one line.

Once students have programmed the rendezvous concept with such a simple and direct program and once they have run the program and observed how the rendezvous concept works "at close quarters", they will be able to program it in the more baroque syntax and semantics of ADA or C++ or Java without loosing track of how powerful, yet simple this programming mechanism really is.

The KL in Computing Courses

The KL approach can be introduced into the curriculum in several ways [12]. One way is to reformulate the typical introductory two-course sequence on programming and problem solving from the current one-imperative-language approach to a

multi-paradigm, multi-thread, net-centric approach. Work is on the way to create appropriate introductory lecture material using the KL approach.

In the early nineties we did apply a three-paradigm approach with some success[13, 14], but the mastering of three entirely different program development environments was too big a burden, especially, so it turned out, for the instructors.

Another way is to teach a KL based *Theory of Programming* course instead of the traditional senior-level "survey of programming languages" course. This approach has been tried with success at the Catholic University of Louvain, Belgium, at the Royal Institute of Techchnology, Stockholm, Sweden [15] and at New Mexico State Univesity[12]. This approach is ably supported with the Van Roy, Haridi book [5] and course materials that are available at the PVR web site at [5] less last component of the link.

SUMMARY

The Kernel Language, backed by a full-fledged multi-paradigm, multi-thread programming language and its implementation allows us to use one program development environment, one compiler and one programming language to introduce students to the essential concepts, features and relative strengths of major programming paradigms as well as to multi-thread programming that may be concurrent in one computer or distributed over internet-connected platforms.

With only one program development environment to learn, all this can be achieved within the two-course, 6-8 credit hour limit that is allocated to computer programming in the typical engineering curriculum.

With the net-centric, multi-language programming platform of dotNET [7] gaining acceptance rapidly, the one language based problem solving and programming courses are rapidly becoming obsolete. A way has to be found to introduce our students to a wider spectrum of programming without loss of depth, but still within the time constraints of the engineering curriculum.

With its small but precisely defined syntax and semantics, the Kernel Language can be used to introduce concepts and clarify issues that are more confusing and time-consuming in the larger programming languages.

Since it is a subset of an implemented programming language Oz, Kernel Language reasoning can at all times be clarified, supplemented or tested by writing and running Kernel Language programs. Pedagogically this makes the theory much more concrete and therefore easier to teach and learn than abstract reasoning by itself.

At the moment the Kernel Language is directly associated with the Oz programming language only, but work is underway to link it to other languages such as Java as well as to the dotNET platform for multi-language programming. A programming platform as large and ambitious as dotNET, needs a programmer's theory of programming to help to keep the complexity of net-centric, multi-language programs within reasonable bounds.

REFERENCES

[1] Sammet, Jean E., *Programming Languages History and Fundamentals,* pp. 143-172, Prentice Hall, (1969).

[2] Knuth, Donald E., *The Art of Computer Programming,* Addison Wesley, (1968).

[3] Reynolds, John C., *The Craft of Programming*, Prentice Hall, (1981).

[4] Dijkstra, Edsger W., *A Discipline of Programming*, Prentice Hall, (1976).

[5] Van Roy, Peter, Haridi, Seif, *Concepts, Techniques and Models of Computer Programming,* book manuscript, www.info.ucl.ac.be/people/PVR/book.html (2002).

[6] Mozart-Oz homepage, www.mozart-oz.org (2002).

[7] Meyer, Bertrand, *.NET Is Coming,* pp. 92-97, IEEE Computer, August 2001.

[8] Dromey, Geoff, Program Derivation, Addison Wesley, (1989).

[9] van de Snepscheut, Jan L.A., What Computing is All About, Springer, (1993).

[10] Gries, David, The Science of Programming, Springer (1981).

[11] Dijkstra, Edsger W., Feijen, W.H.J., A Method of Programming, Addison Wesley, (1988).

[12] Reinfelds, Juris, Teaching of Programming with a Programmer's Theory of Programming, Proceedings ICTEM'2002, Kluwer Academic Press (2002).

[13] Reinfelds, Juris, A Three Paradigm Course for CS Majors, Proc. 26th ACM SIGCSE Technical Symposium on Computer Science Education, p.223-227, (1995).

[14] Reinfelds, Juris, 1996 Lecture Notes for CS 272 (new), 251 pages, Department of Computer Science, New Mexico State Univ. (1997).

[15] Van Roy, Peter, Haridi, Seif, Teaching Programming Broadly and Deeply: the Kernel Language Approach, Proceedings ICTEM'2002, Kluwer Academic Press (2002).