A Theory of Programming for Engineers

Juris Reinfelds

Klipsch School of EE &CE, New Mexico State University

Introduction

A theory is a concise set of precisely defined concepts and notation-symbols that can be used to reason about a much larger, much more complicated and less precisely defined system. A good theory captures the essential structure of the system that we want to study and reason about.

An elegant theory is simple in concept, yet wide in scope. For example, Kepler's theory of elliptical planetary orbits is more elegant than the epicycle theory of the same orbits. A theory is more useful to engineers and scientists if it is expressed in terms of concepts that are already at least partly familiar from the day-to-day work of these engineers or scientists. For example, thermodynamic theory of molecular motions is much less effective in the derivation of the period of the pendulum than Newton's Laws of Motion.

Mathematicians have developed theories of computability that we now refer to as "*theory of computer science"*, but these theories are not directly useful to practicing programmers who want to reason about their programs, because they are based upon mathematician's concepts such as Turing machines and lambda calculus.

The First Programmer's Theory of Programming

Some 30 years ago EdsgerW. Dijkstra [1, 2] developed a simple, yet very effective theory, which programmers could use to reason about simple imperative single-thread programs. Earlier, Robert Floyd [3] had introduced the notions of preconditions, postconditions and loop invariants in mathematical terms and Sir Anthony Hoare FRS [4] had incorporated these concepts into a mathematically oriented, precisely defined programming language. However such a mathematical theory was not directly useful to programmers and very little use of these concepts was made until Dijkstra and Peter Naur [5] expressed the theory in programmer's terminology.

The core of Dijkstra's theory is very 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"*. A consequence of this rule is that it allows the formation of compund statements, in particular, 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 replace them by 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 the more familiar C-programming notation of today:

<statement> ::=
<statement1> <statement2>
<variable> = <expression>
if <conditional expression> **then** <statement1> **else** <statement2> **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 sequence 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 non-negative arguments, its precondition is *X>0* and its postcondition is *X == Y*Y*

A program that, at a certain point, expects the value of a *key* to be greater or equal than the first value *a[0]* of an array and less than the last value *a[N]* of that array, requires a precondition

a[0] =< key < a[N]

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 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 cal 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. The usefulness of Dijkstra's theory to the construction of loops is best illustrated with an example. Let us consider the well know algorithm of binary search. Binary search is a deceptively simple algorithm that easily leads to an incorrect program if we "guess" the program in the usual way.

Binary Search Problem

Given a *key* and an ordered array *a[0] ... a[N]* such that *a[i] =< a[i+1]* for *0 =< i < N*, write a program that sets the variable *present* to *true* if at least one array element is equal to the *key*. Otherwise the variable *present* is set to *false*. Design a loop that takes the smallest number of iterations to determine the value of *present*.

Ad-hoc Reasoning

Start with two variables *bot:=0* and *top:=N-1*. In the loop, check the midpoint of the range *mid:=(bot+top) div 2* and discard the half where the *key* cannot be. When *bot==top*, then the one remaining array element in the range is either equal to the *key* or the *key* is not equal to any array element. This leads to one of two programs that differ only slightly

```
bot := 0   top := N-1
while  bot < top   do
       mid := (bot + top) div 2
       if  key > a[mid]
             then  bot := mid + 1
             else  top := mid
       fi
od
present  :=  (key == a[bot])
```

```
bot := 0   top := N-1
while  bot<top   do
       mid := (bot + top) div 2
       if  key < a[mid]
             then  top := mid - 1
             else  bot := mid
       fi
od
present  :=  (key ==a[bot])
```

Only one of these programs “works”. The other one loops forever. The reader is invited to translate the programs into an available programming language and find out which program does not work and why.

Applying Dijkstra’s Theory to Binary Search

Dijkstra’s theory requires us to make sure that each iteration of the loop makes progress towards the postcondition of the loop. In binary search, progress is made by reducing the number of array values between *bot* and *top*, but when *bot* and *top* are adjacent indices, no progress is made because *mid* equals *bot* or *top*, depending on whether integer division rounds 0.5 up or down. Therefore let us stop the loop with the postcondition

a[bot] =< key < a[top]   AND   top == bot+1

so that *a[bot] == key* decides whether key is equal to at least one array element, but the treacherous last iteration of the ad hoc loop is avoided. This postcondition suggests the loop invariant

a[bot] =< key < a[top]

and the loop condition

while top > bot+1 **do ...**

as well as the precondition for the loop statement of the program

a[0] =< key < a[N]

that is established by extending the array with a sentinel value *a[N]* that satisfies *a[N] > key* and an if statement (not shown) that sets *present* to *false* and exits if *key* is less than *a[0].* A correct program for binary search follows immediately

```
bot := 0   top := N
while  top > bot+1   do
        mid := (bot + top) div 2
        if  key < a[mid]
                then  top := mid
                else  bot := mid
        fi
od
present  :=  (key == a[bot])
```

The Kernel Language Approach

Dijkstra's theory is concise, elegant and useful, but since it is built around the assignment statement and iterative loops, it applies only to imperative programming. A theory that includes other programming paradigms as well as parallel, concurrent and distributed computing requires a different starting point.

The Mozart-Oz [6] programming system implements a new and innovative programming language Oz that incorporates the best of functional, logical, imperative, object-oriented, parallel, concurrent and distributed programming into one programming language.

To show that Oz is a concise and elegantly structured programming language built on a set of carefully selected concepts from the paradigms mentioned above, Peter Van Roy and Seif Haridi[7], introduced a concise, elegant and precisely defined Kernel Language in which we can express every construct and feature of the full Oz programming language. The Kernel Language is a subset of the Oz programming language, so that kernel language programs are executable in the Mozart-Oz programming system.

Much as Dijkstra's theory applied to imperative programming, the Kernel Language provides a programmer's theory for all the computational paradigms mentioned above. So far it has been applied to a better understanding of Oz programming, but in key programming concepts Java is a subset of Oz, so kernel language skills should lead to a better and deeper understanding of the programming concepts of Java. In Spring 2002 at NMSU, a course EE 590 entitled *"A Programmer's Theory of Programming"* is in progress, testing better ways to teach multi-thread programming.

It would take at least a half-day workshop to explain the kernel language in sufficient detail to fully appreciate its concise elegance and remarkably wide scope. In this paper, as for Dijkstra's theory, we will show the show the syntax and skip over the semantics of the kernel language and we will have room for just a couple of examples on how the kernel language definition clarifies some programming concepts that C and Java programmers find puzzling or error-prone.

The Basis of the Kernel Language

Dijkstra's theory is built around the assignment statement and integer data values. The kernel language has a different and wider base. Variables are declared without value or type. Strong typing is dynamic. Variables acquire a value and with it a type in a binding operation (denoted by "=") that is similar to unification of logic programming.

As for *final variables* of Java, the values of kernel language variables cannot be changed. This simplifies reasoning about concurrent and parallel programming. It is surprising how many problems do not need assign-many-times variables. Default use of assign-many-times variables unnecessarily complicates reasoning about programs that could easily be programmed with *final variables* only.

Unbound variables may be used in expressions. The expressions are still strongly typed because type compatibility is checked dynamically when such an unbound variable is bound to a value in another thread of computation. What happens when computation of an expression reaches an unbound variable? The rule is:

> *The execution of an expression suspends if it has to use a variable that is not bound to a value. Execution of this expression resumes and continues when the unbound variable is bound to a value in another execution thread.*

This turns multithread computing into an integral and natural part of the programming language instead of treating threads as an external, library-accessed activity as in C++ or as a complicated language activity with special *run, start* and *stop* methods as in Java. The kernel language invokes threads with a simple thread-statement

thread <statement> **end**

that executes the enclosed statement (which may be a sequence of any number of statements) in a separate execution thread and then terminates that thread. With the execution suspension rule and the thread-statement we can construct, study and compare synchronization mechanisms, race conditions, deadlocks, producer-consumer problems and other aspects of concurrent and parallel computations directly and in full generality. After that we can turn our attention to the implementation peculiarities of threads in Java and C++. Separation of concerns increases depth of understanding and decreases the effort required to master the concepts.

The Syntax of the Kernel Language of Van Roy and Haridi

<statement> ::=

skip	Empty Statement
<var1> = <var2>	Variable to variable binding
<var> = <value>	Value to variable binding
<sta1> <sta2>	Statement sequence
local <var> **in** <sta> **end**	Declaration of variable
if <var> **then** <sta1> **else** <sta2> **end**	Conditional statement
case <var> **of** <pattern> **then** <sta1> **else** <sta2> **end**	Pattern matching
{ <var> <arg1> ... <argN> **}**	Procedure call
thread <sta> **end**	New computation thread
try <sta1> **catch** <var> **then** <sta2> **end**	Exception handler
raise <var> **end**	Raise exception
{NewCell <var1> <var2>}	C-like Assign-Many-Times var
{Access C X}	Bind X to content of C
{Assign C X}	Set content of C to value of X

Basic Data Types of the Kernel Language

<value> ::= <number> | <record> | <procedure>
<number> ::= <int> | <float>
<record> ::= <literal> | <literal> **"("** <feature1> **":"** <var1> ... <featureN> **":"** <varN> **")"**
<procedure> ::= **proc "{" "$"** <arg1> ...<argN> **"}"** <statement> **end**
<literal> ::= <atom> | <bool>
<feature> ::= <literal> | <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? The kernel language makes it clear. The statement

X = 1230

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

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

binds the procedure value to the variable *Cube*, which can then be used as the name of the procedure. 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, we can also bind or use or call a procedure. This is what is meant by *"a first class value"*.

Object Oriented Computations

A *class* is a data structure. It is a first class value that specifies objects and methods of that class. An *object* is a data structure that encapsulates an explicit state as defined by the class of which the object is "an instance". Cells are used to represent state because the values referred to by cells can be changed as many times as needed. The state encapsulated by an object can only be accessed in a controlled way by *methods,* which are procedures that are defined in the class definition.

Conclusion

The binary search example showed us how a programmer's theory of programming can help us to create programs with prdictable behaviors. The Kernel Language allows us to study difficult programming concepts, such as thread synchronization and race conditions directly, in their most general forms, unconstrained by paradigm, tradition or the whims of a particular programming language designer. The Kernel Language is a small language with precisely defined syntax and semantics that allows us to acquire a good understanding and working knowledge of the most important concepts of many programming paradigms in a surprisingly short time.

For the first time we have a promising programmer's theory that spans all programming. Let us hope that it will help us to design and implement CAPS systems that actually work predictably.

References

[1] Dijkstra, E.W., "Notes on Structured Programming", pp. 1-82, in Dahl & Hoare & Dijkstra, Structured Programming, Academic Press (1972)

[2] Dijkstra, E.W., "A Discipline of Programming", Prentice Hall (1976)

[3] Floyd, R., "Assigning Meaning to Programs", pp.19-32, Mathematical Aspects of Computer Science XIX, American Mathematical Society (1967)

[4] Hoare, C.A.R., "An Axiomatic Approach to Computer Programing", CommACM, pp.576-580,Vol.12#10,(1969)

[5] Naur, P., "Proofs of Algorithms by General Snapshots", BIT Vol.6 pp.310-316, (1969)

[6] www.mozart-oz.org (2002)

[7] Van Roy, P., Haridi, S., www.info.ucl.ac.be/people/PVR/book.html (2002)

Biographical Information

Juris Reinfelds received his PhD from the University of Adelaide, South Australia, in 1963. Through the University of Edinburgh, University of Adelaide, Marshall Space Flight Center, University of Georgia, CERN in Geneva and the University of Wollongong, he is now Professor of Computer Engineering at New Mexico State University.