Towards an Introductory Formal Programming Course

Kung-Kiu Lau, Vicky J. Bush, and Pete J. Jinks
Department of Computer Science, University of Manchester
Manchester M13 9PL, England, United Kingdom
kkl,vjb,pjj@cs.man.ac.uk

1 Introduction

It is now well established that programming can be regarded as a science rather than an art. The formal basis for programming given by Hoare [7], and the subsequent application of this formalisation to program development, most notably by Dijkstra [2, 3] and Gries [5], have left little doubt that this is indeed so.

Therefore it is disappointing (and surprising) that most introductory programming courses still adopt the 'traditional' intuitive approach. The (formal) syntax of the chosen programming language, often Pascal or C, is explained in great detail, but the semantics of the language is only explained informally; and little (if anything) is taught about how to develop a program to do a specified task, apart from 'hand-waving' principles such as 'divide-and-conquer', 'structured programming', 'top-down step-wise refinement', and so on. Consequently, beginners learn the hacker's way of programming, which has no formal basis and hence cannot even begin to deal with crucial issues such as program correctness.

There may be many reasons for this unfortunate fact.[1] The 'old' view of programming as an intuitive activity seems hard to change, let alone replace. It is seen to be adequate, in the sense that real programs and software systems can be produced using the intuitive approach. However, acceptance of bugs as the norm and the high costs of maintenance point to its unacceptability.

New programming paradigms, particularly functional programming, have diverted the attention on program correctness to simple declarative semantics. These declarative paradigms, though attractive because of their simple and clean semantics, are still supported by the traditional von Neumann computer architecture (for example, see [10]). They are therefore implemented in the imperative paradigm within which the Hoare-Dijkstra-Gries method has evolved. Most students will meet the imperative paradigm early in their professional career, so the issue cannot be avoided.

Unfortunately, there are no widely available introductory programming courses (with good software support) specifically designed to support the effective teaching of the science of programming. Like our colleagues at the City College, City University of New York [9], we are convinced that the Hoare-Dijkstra-Gries method should be the basis for all introductory programming courses. We have therefore set about designing a course and a programming environment suitable for teaching beginners the key concepts of *formal programming*. That is, programming in a formal manner, with a view to enabling them to reason formally about programs and program development.

This course aims to teach students to think about programs in a formal way and to be able to reason about their correctness with respect to a formal specification. It also takes on board the reality that many students learning programming for the first time cannot necessarily see the rationale behind such an approach until they have gained more experience. They often do not have the mathematical maturity necessary for difficult proofs. To try and overcome these difficulties, an integral part of the course is a supportive programming environment which hopefully will take much of the drudgery out of their work and allow them to concentrate on the main concepts. It should also make it possible to tackle larger problems than would be possible without machine support for the proofs. This is important for students who may have already written some substantial programs in an intuitive way and would not be motivated by 'toy' programs. It is often the case with such students that it is enjoyment of this activity which has attracted them to a computing course in the first place. They do not want their experience to be devalued.

In this paper, we outline the course material and sketch the role of the software support environment for teaching this material. The aim of the paper is to get timely feedback from colleagues with similar or related views and/or experience.

[1] See [4] for an interesting debate.

SIGSCE 94- 3/94, Phoenix, Arizona,USA

2 Course Components

The course has two main components: the course material and the software support environment.

Course Material

We believe that one course cannot teach the entire Hoare-Dijkstra-Gries method. The learning process takes time and practice. Also, software support for the whole method will take a long time to develop (as illustrated by [6]). So we have aimed for just the key basic concepts of the method. For teaching practical programming using this method, we have designed a simple Pascal-like language with some functional language features in order to achieve a simple semantics. We call this language **L** (for 'learners'). We have also designed a specification language for expressing pre- and post-conditions. We call this language **S**. In practice, these two languages will appear to students to be parts of a single design and implementation language **SL**.

Software Support Environment

For teaching the course material successfully, it is essential to have a good software support environment in which the students can (i) use on-line tutors or tutorials to learn the course material at their own pace; (ii) develop programs with the aid of necessary software tools such as structure editors. We have begun to implement such an environment with a WIMP interface. So far, the environment consists of a compiler for **SL**, an assertion checker, a tutor for **L**, and windows for constructing and running **SL** programs. We intend to add a structure editor, to simplify the problem of creating syntactically correct specifications and program code. We also intend to include an on-line help system and tutorials to consolidate the students' understanding of all parts of the course material. We expect that other facilities may be considered in the light of future experience, such as runtime animation and debugging.

In Section 3, we describe the course material in more detail, and in Section 4 we will illustrate developing programs in **SL** using the software support environment.

3 Outline of Course Material

In the course material, it is of crucial importance that basic concepts are properly identified and defined. In this section we list some of the key concepts which we think are necessary for teaching formal programming, in the order in which they might logically arise. We shall also give examples in **SL**. Rather than describe the syntax and semantics of **SL** in detail, we will present a flavour of the language by explaining the salient features used in the examples. Those interested in the full language details should consult the authors.

Computation: The fundamental concept of a *computation* can be defined as a process on a *computational device* with an *instruction set*, which starts with given initial *input* 'values' for some specified objects and then performs a series of operations. The resulting 'values' of some specified objects may be regarded as the *output* of the computation.

Computational model: The equally fundamental concept of a *computational model* can be defined as an abstract computational device with a specified instruction set capable of operating on specified objects. Thus we can define a computational model as a pair $(\mathbb{O}, \mathbb{I})$ where $\mathbb{O}$ is the set of objects in the model and $\mathbb{I}$ is its instruction set.

Program: This leads immediately to the concept of a *program*. For a given computational model, a *program* defines a computation in this model.

Syntax and semantics of a formal language: The concept of a *formal language* can be motivated by considering how to write complicated programs for a given computation model. To write such programs we need to make up a *formal language*, by defining a *syntax* and a *semantics* – a set of mappings which map every permitted syntactic construct to its meaning, or *semantic denotation*.

We can then introduce the *Backus-Naur Form* (BNF) and, as an illustration, use it to define the syntax of a simple formal language (e.g. that of arithmetic expressions).

Next, the principle of *denotational semantics* can be explained using the same language as an example.

Abstractions: To write more general programs which allow for more general computations which depend on arbitrary values (that is, functions with arguments), we need to have a means of constructing *abstractions*. To this end, we introduce *λ-expressions*.

Programming language: Abstractions lead naturally to programming languages. In order to apply an abstraction repeatedly to different concrete arguments, we need to be able to create, name and remember an abstraction. A formal language which allows the user to define (create, name and remember) and apply abstractions is called a *programming language*.

For example, the following could be a program incorporating the definition and an application of an abstraction, 'product'.

```
def_prog example1:=lambda(a,b)=(c)
        def_fun product:=lambda(x,y).x*y
        modify c:=product(a, b)
endprogram
```

Here, `def_prog` is a keyword denoting a program. `example1` is a program identifier defined to be a function that maps the input variables `a` and `b` to the output variable `c`. `lambda(a,b)=(c)` defines this function, where `lambda` is a keyword that denotes a λ-expression. The reserved symbol `:=` means 'becomes defined to be'.

`def_fun` is a keyword denoting a function definition, and `modify` is a keyword that denotes the assignment command. `endprogram` is a bracket keyword.

Specifying computations: In a programming language, computations are always explicitly defined. However, often we cannot define a computation explicitly because we only know its result, but not the computation itself. For example, if we want to compute the square root of an integer x, but do not know an algorithm for computing square roots, then we cannot write the necessary program. Therefore, we need to introduce the concept of implicitly *specifying computations*, that is, only in terms of what we know about their results.

As an example, a 'square root' function could be specified by:

```
spec_fun sqrt := lambda(x) = y
{pre x>0}
{post x=(y*y)}
```

Here, `spec_fun` is a keyword denoting a function specification. The *pre-condition* `{pre x>0}` and the *post-condition* `{post x=(y*y)}` are properties that we define for the *input* and *output* of `sqrt`. Such properties are in general best expressed as logical conditions.

Program Development: The concept of *program development* now follows naturally. For a computation that is only specified implicitly, we need to develop a program that when executed will produce the specified computation. In general, constructing a program for a specified computation is the usually nontrivial task of finding a computation that produces the result that we have in mind. In other words, if the program's input meets the pre-condition of the specified computation, then the program's output must satisfy the post-condition.

4 A Programming Example

In this section we sketch a glimpse of how a student might use the software support environment to learn to program in **SL** using the Hoare-Dijkstra-Gries method. The example chosen is the problem of how to write a program which calculates the factorial of a given number. For simplicity, we will assume that all values are of type integer.

The specification for this problem is:

```
spec_fun fac:=lambda(i)=(f)
{pre   i>=0}
{post  ((i=0) and (f=1))
       or ((i>0) and (f=i*fac(i-1)))}
```

This is the usual recursive definition of factorial written in the syntax of **S**.

```
spec_fun fac:=lambda(i)=(f)
```

defines `fac` to be a function that maps `i` to `f`.

```
{pre i>=0}
```

is a pre-condition requiring `i` to be nonnegative, and

```
{post  ((i=0) and (f=1))
       or ((i>0) and (f=i*fac(i-1)))}
```

is a post-condition that defines `fac(0)` to be `1`, and recursively defines `fac(i)` (i.e. `f`) to be `i*fac(i-1)`.

To write a program to compute `fac(n)`, the student can use the support environment to construct a 'shell' of the program with the pre- and post-conditions inserted (during program development $\cdots$ is used to denote as yet unknown code, although it is not part of **L**'s syntax):

```
def_prog example2:=lambda(n)=(a)
{pre n>=0}
      ...
endprogram
{post a=fac(n)}
```

The next step is for the student to think about the design of the program. In larger examples, there may be some splitting up of the program into parts. The goals of each sub-part would have to be understood and expressed by way of an assertion in the specification language. As code is written to achieve the sub-goal, this would then be proved by the assertion checker.

In this case, the program is small and does not need splitting up. The main loop construct in **L** is the `while` command, and since `fac` is defined recursively, it may be obvious that a `while` command can be used to compute `fac(n)`. In fact, we will use it to calculate successive factorials, terminating when we arrive at `fac(n)`. However, in **SL**, the user must supply a loop invariant for a `while` loop, which defines the meaning of the loop. In this case, the recursive definition of `fac(i)` itself provides this invariant, since for any `i`, i.e. for any iteration of the loop, we must have `fac(i)=i*fac(i-1)`. Thus the student might add the shell of the loop to the program as follows (we use `>>` here to highlight newly added/changed code, it is not part of **L**'s syntax):

```
def_prog example2:=lambda(n)=(a)
{pre n>=0}
>>      def_var i
>>      modify i:= ...
>>      while i ... n do
>>      {invariant a = i*fac(i-1)}
>>                ...
>>                modify i:= ...
>>                ...
>>      endwhile
endprogram
{post a=fac(n)}
```

Now the student must try and construct the body of the loop to have the semantics defined by the loop invariant. Clearly, the program must compute `fac(0)=1`, `fac(1)=1*fac(0)=1`, `fac(2)=2*fac(1)=2*1=2`, etc. until `fac(n)` has been computed. Now (from the specification) we know the loop need not be executed for

n=0 and n=1 (since fac(0)=fac(1)=1). So we can deal with these two base cases as follows:

```
def_prog example2:=lambda(n)=(a)
{pre n >= 0}
        def_var i
>>      modify a:=1
>>      modify i:=1
>>      while i < n do
        {invariant a = i*fac(i-1)}
                ...
>>              modify i:=i+1
                ...
        endwhile
endprogram
{post a=fac(n)}
```

Now we must compute fac(i) for i>2 using the while loop. Again, clearly the loop invariant says that each loop is meant to compute fac(i) by computing i*fac(i-1), starting ¿from i=2. Therefore, the body of the loop needs to simply multiply i and fac(i-1). So the user can complete the body of the while loop as follows:

```
def_prog example2:=lambda(n)=(a)
{pre n >= 0}
        def_var i
        modify a:=1
        modify i:=1
        while i < n do
        {invariant a = i*fac(i-1)}
                modify i:=i+1
>>              modify a:=a*i
        endwhile
endprogram
{post a=fac(n)}
```

Up to now, the user has been reasoning in **SL** informally, while constructing the above program. Now she or he can 'run' this program to verify the correctness of her or his reasoning and hence the correctness of this program. The assertion checker in the support environment will first check that the loop body satisfies the loop invariant. If this check is successful, then the assertion checker will check that the invariant will hold if the pre-condition is satisfied. Then it will check that the whole program satisfies the post-condition. In this case, this is straightforward. After the loop finishes, we have a=n*fac(n-1), which clearly proves the post-condition.

Obviously, in general, program design is a creative process which cannot be taught directly. Finding suitable loop invariants will not be easy for beginners, but at least in formal programming, students have to understand (and define) precisely the semantics of each loop, which will guide them in designing the loop construct. Loop termination is another issue that is potentially difficult for beginners. Again, having a loop invariant will discipline their thinking about the loop construct, and 'force' them to take care that the loop terminates. However, the students can get support from the programming environment to overcome all these difficulties.

Finally, one aim of the programming environment is to enable the handling of larger examples than the factorial example shown here. Larger programs have a corresponding increase in the size and complexity of their proof obligations. In order to circumvent the work associated with this, it is envisaged that a library of already proven, relevant abstract data types and functions would be made available to the students. Then, just as in a more realistic industrial setting, they could build more interesting and useful programs, without having to start from scratch.

5 Conclusion

We have outlined the course material and the software support environment for teaching formal programming (based on the Hoare-Dijkstra-Gries method) to beginners. In the past, at the authors' institution, beginners have been taught to understand, not write, formal specifications in VDM. The design process was informal. Teaching the construction of specifications and formal derivation of proofs was deferred until the following year. This proposal is a way of making a formal approach accessible right from the start. It is recognised that it will be effective only if the programming environment provides enough support to guide students through the process in a way which supports correctness proving and reduces the amount of drudgery involved. We have already started work on building this environment and are hoping to gain feedback from others working in the area as to their experiences of the effectiveness and/or pitfalls of the approach, before we complete our implementation.

Of necessity, on the whole, our approach tackles 'programming-in-the-small', but this lays a firm foundation for more realistic software production ('programming-in-the-large'). To quote David Gries [5]:

> "One cannot learn to write large programs effectively until one has learned to write small ones effectively."

References

[1] H. Abelson, G.J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[2] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[3] E.W. Dijkstra and W.H.J. Feijen. *A Method of Programming.* Addison-Wesley, 1988.

[4] E.W. Dijkstra *et al.* A debate on teaching computing science. *Comm. ACM*, 32(12):1397–1414, 1989.

[5] D. Gries. *The Science of Programming.* Springer-Verlag, 1981.

[6] M. Heisel. Formalizing and implementing Gries' program development method in dynamic logic. *The Science of Programming*, 18:107–137, 1992.

[7] C.A.R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–583, Oct 1969.

[8] C.B. Jones. *Systematic Software Development Using VDM.* Prentice Hall, 2nd edition, 1990.

[9] D.D. McCracken. Programming languages in the computer science curriculum. In *Proc. 23rd SIGCSE Technical Symposium*, pages 1–4. ACM Press, 1992. ACM SIGCSE Bulletin, vol 24, no 1, March 1992.

[10] S. Peyton-Jones and D. Lester. *Implementing Functional Languages.* Prentice Hall, 1992.

[11] J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 2nd edition, 1992.