On Languages, Models and Programming Styles

MARIAN PETRE AND R. WINDER
Department of Computer Science, University College London, Gower Street, London WC1E 6BT

In this paper, we attempt to clarify both the terminology used to classify programming languages and the nature of the classification itself. We feel that the current terminology and classification as captured in the literature is out of balance with the way that subject programming languages are used. We consider the accessibility of the computational model and the importance of the implementation language and suggest that a re-appraisal in this context will produce a better characterisation of both the programming language styles and their labels.

Received June 1988, revised December 1989

1. INTRODUCTION

'When *I* use a word,' Humpty Dumpty said, in rather a scornful tone, 'it means just what I choose it to mean – neither more nor less'.
'The question is,' said Alice, 'whether you *can* make words mean so many different things.'
'The question is,' said Humpty Dumpty, 'which is to be master – that's all.'
Through The Looking Glass, Lewis Carroll.

In any discipline, ideas and terminology proliferate together. Unfortunately, the relationship of one to the other is not always clearly defined; labels and expressions infiltrate common parlance often before their meanings are precise, so that discussions are ill-founded and ambiguous. Computer science is no exception; terminology covering programming paradigms – even the use of 'paradigm' – is disputed.

In this paper, we try to clarify the nature and usage of programming languages in order to understand the qualities driving classification. Viewing coding as a translation process, we consider the accessibility of the models between which translations must be accomplished. Hence, we examine the computational models which underlie programming languages and consider at what level – language surface or implementation – computational mechanisms are evident and at what level algorithmic decisions are defined. This examination guides our re-appraisal of the terminology, starting with our notions of 'specification' and 'program', and culminating in a series of questions which highlight the significance of 'imperative' and 'declarative'. We find that, in this formulation, the languages form a continuum, and we present a simple map.

1.1 Terminology

We start by giving the following short, 'garden variety' definitions as a base for our examination:

Imperative: Imperative languages express sequences of operations required to achieve a calculation. These languages are 'state-oriented'; they imply an underlying machine manipulated explicitly by a programmer's commands.

Declarative: Declarative languages emphasise what is to be calculated rather than how the calculation should proceed. How the calculation is performed depends on the implementation, which embodies algorithmic information not found directly in the language. A declarative program is a statement of constraints on the solution which can be read 'declaratively' as a description of the solution set. These constraints coerce the algorithmic engine embedded in the language implementation to produce a solution or set of solutions.

'Imperative' and 'declarative' are the major classes on which we focus our appraisal. The following terms govern sub-classes or styles, and their meanings are less problematic:

Procedural: A sub-class of imperative languages, procedural languages incorporate language constructs for modularising source code in the form of procedures or functions which are called with parameters and which return control to the caller.

Object-Oriented: A sub-class of imperative languages, object-oriented languages view computation as interaction among active data objects. All data are objects, all objects are treated uniformly, and all processing is done by passing messages among objects. Each object embodies operations defined for it, and 'control' takes the form of requests ('messages') sent to data objects to transform themselves. Data abstraction is sustained, and the internals of objects are hidden. Similar objects can be grouped in a hierarchy of classes, with classes able to 'inherit' common attributes form their superclasses.

Logic: A sub-class of declarative languages, logic programming languages provide constructs for defining atomic relationships among values by asserting facts and rules about them in the form of 'implications' in which at most one conclusion derives from the conjunction of zero or more conditions. The logic used is of first order; the clauses that constitute a program are mutually independent but cannot be treated as objects in their own right. Programs are 'interrogated' to elicit truths about individuals and their relationships. The engine for 'resolving' and 'unifying' these relationships is wholly within the language implementation. Programs generate sets of answers, not necessarily single answers. Serialisation of the set generation is usually handled by the user interface.

Functional: A sub-class of declarative languages, functional languages specify output as a function, in the mathematical sense, of the input. These languages are 'value-oriented'; there is no implied state and hence no changes of state or 'side-effects'. These languages incorporate 'referential transparency'; the value of a function is determined solely by the values of its arguments, so a function called with the same arguments will always yield the same values.

Applicative: Treated here as a synonym for 'functional', this term emphasises that effects are achieved by composing functions and applying them recursively. This term is used elsewhere as a stricter label the sense of 'purely applicative'; that is, referential transparency is maintained for all expressions.

1.2 Computational Model

Underlying all programming is some notion of a machine. However a solution is pursued, the vehicle is the computer. Abstraction from specific machine operations – the evolution of high-level languages – has made it possible to characterise different models of computation compatible with actual processing. Embodied in each language is such a computational model, a view of the actions and interactions by which solutions are achieved. Moreover, some of the structure originally imposed at the hardware level – actually 'wired in' – is now provided within language implementations, so that the machine-level model is simpler, and sophistication is introduced in the computational models underlying high-level languages. We will examine our terminology for languages in terms of the distinctions or similarities among the embedded views of the computational world.

Between conception and computation are translations; of strategies to code, of source code to machine code, of instructions to actions. The introduction of levels of abstraction, i.e., intermediate models, implies additional translations. These are introduced to bring the model of computation closer to the user by reducing the translation distance between the top layers of translation. In compensation, the translation distance between underlying layers – between the language implementation and machine instructions – lengthens. In fact, in recognition of the improvements in language technology, machine instructions have become more rudimentary. Less structure is imposed at the level of machine instructions, making the hardware more amenable to the imposition of various computational models, with the consequence that the machine language is even further from the user. The lengthened

Computation **Conception**

1. In the beginning, there was machine code:

⟨.................. **Machine** ....⟩ ⟨...................................... **Mind** ....⟩

2. Assembly language provided a computational model closer to the programmer.

⟨.................. **Machine** ....⟩ ⟨ **Assembler** ⟩ ⟨............................ **Mind** ....⟩

3. The high level languages further reduced the user-end translation distance:

⟨.................. **Machine** ....⟩ ⟨ **Assembler** ⟩ ⟨ **Compiler** ⟩ ⟨.................. **Mind** ....⟩

4. Attempts were made to make the hardware simpler but without changing the boundary between hardware and software. An extra level was introduced: microcode programs were held in ROM and called firmware:

⟨......... **Machine** ....⟩ ⟨ **Microcode** ⟩ ⟨ **Assembler** ⟩ ⟨ **Compiler** ⟩ ⟨.................. **Mind** ....⟩

5. Attempts were also made to make the hardware much more sophisticated and so directly able to support the high level languages:

⟨............................ **Machine** ....⟩ ⟨ **Compiler** ⟩ ⟨.................. **Mind** ....⟩

6. The current RISC idea is a return to the simple hardware of 4 but requiring the compiler to provide the sophistication for the user; there is no manufacturer-supplied-software level:

⟨......... **Machine** ....⟩ ⟨ **Assembler** ⟩ ⟨........... **Compiler** ....⟩ ⟨.................. **Mind** ....⟩

7. The above are all compilation models. There are also interpreters which are software machines, programs that act like hardware computers:

⟨......... **Machine** ....⟩ ⟨........... **Interpreter** ....⟩ ⟨............................ **Mind** ....⟩

Figure 1. Translation distances

translation gap is bridged by the high-level language compiler. Figure 1 shows these changes of translation distance over time.

The priorities of programming language development are clear: the translation distance from the user mind to the user language must be minimised; the rest is technology. The distance of the hardware model from the user is irrelevant, as long as a translation path which preserves semantics exists from the user's computational model to the machine model. It is the translation distance between the top layers – between the user's perception of the language model and the computational model embedded in the language – that is critical.

Efficacy of programming depends on the continuity of translation between these top layers and so is concerned not only with how well the model captures computation but also with how accessible the model is to the user. In order for programs to succeed, the programmer's perception of the model must reflect the language with some accuracy.

Some tolerance is required in this discussion. The perimeter of a language is not strictly defined, and so the underlying computational model, although stable in principle, has blurred edges. Mechanisms such as procedure and data abstraction define new words which become part of the language. They enable extensions to a language in the language, which confuses the borders of both language and model.

Moreover, a program implies a model which is related to the language's computational model. A program selects from the language model in the sense that it exercises only portions of the model. Yet it also extends the model of computation by composing and defining new items using language constructs.

Thus, it is not a single computational model which governs effective programming, but the interactions of several. The compatibility of the model embodied by the program, the model inherent in the language, and the programmer's perception of both of these are issues of translation distance.

2. CHARACTERISATION OF MODELS UNDERLYING STYLES

Our willingness to classify languages (i.e., to exercise the terminology under discussion) implies that we acknowledge some commonality of models among languages of a style or class, even if we reject that those computational models are equivalent. Thus, models inherent in languages considered declarative share characteristics not found in models embodied in imperative languages, and so on. For the purposes of this discussion, we will treat a style as having an inherent computational model which is the general distillation of attributes shared by models underlying the languages of the style.

2.1 The Imperative Model

The imperative computational model is reflected by the notion of the von Neumann machine, which was for many years the model for all digital computers. The basic von Neumann machine has two components: a memory and a single processor. Programs, like all other data, are stored in memory, from where they are retrieved for execution for the processor. The processor performs two sorts of functions: it can access any memory location to retrieve or modify the contents, and it can execute instructions, which exercise its simple logic operations, one at a time in a sequence defined by the program. Thus, memory and a single processor combine into a model of a global environment which undergoes incremental changes.

The imperative model incorporates these von Neumann machine characteristics in the notions of assignment, state and effect. Values are assigned to variables, which are seen as 'boxes' whose contents are mutable. Procedures operate by modifying their parameters or global variables. Hence, the machine comprises objects (variables) which can change over time. Collectively, the values of the variables at a given time describe the state of the machine. This model of operation by change of state and by calculation and alteration of variable values, yields the notion of 'computation by effect'.

Under this model, algorithms are conveyed as a consequence of changes of state. The subject languages contain explicit control structures for guiding the flow of execution.

The closeness of this computational model to the hardware model is a matter of evolution. Programming languages began as strings of machine-specific binary codes corresponding to individual machine operations. These machine language instructions comprised two parts, resembling the von Neumann model: operation code and memory location. Next, the assembly languages introduced mnemonics to represent the binary instructions and removed the need for programmer control of storage locations but necessitated translation of this symbolic code into machine code and 'assembly' of the variously stored program components (subroutines). Although some mnemonics represented more than one machine instruction, translation remained mainly 1:1. The so-called high-level languages abstracted from assembly language, becoming machine independent and incorporating composite constructs. These languages require more sophisticated translation ('compilation') from their portable form into machine-specific code. These high-level languages, as abstractions from machine codes, still reflect basic machine operations. Developments of structure and style reflect the consideration of programming languages in their own right instead of as versions of machine code, that is, as mere extensions of hardware.

This closeness of language to machine model is reflected in practical ways, e.g., the good control afforded by many imperative languages over machine aspects such as memory allocation and I/O. However, the strength of correspondence means that imperative languages embody hardware-based restrictions, so that pragmatics may intrude upon expression. It may be impossible to express structures whose bounds are not defined or whose characteristics may change at run-time.

2.2 The Declarative Model

The declarative model is divorced from the von Neumann machine, from explicit sequential control, from state and from what John Backus[1] labelled 'the von Neumann bottleneck' – assignment. This is a model of 'computation by value'. Functions, rather than causing the 'effects' of modifying parameters or variables, return values. Data items are immutable; there is no sense of updatable memory accessible by instruction.

There is no sense of instruction, instead there is a 'script' which defines what is to be computed. The manipulations by which this objective is achieved are not explicitly part of the model. Indeed, David Turner[3] introduced the term 'script' for the programs written in his functional languages to emphasise that such programs were qualitatively different from their imperative counterparts.

Distance from machine operations is achieved by declarative languages at the cost of certain practicalities, those which govern performance. Input/output can be awkward, and matters such as garbage collection and efficiency cannot be addressed directly by the programmer. What is gained in exchange is release (in concept and notation) from some hardware-based restrictions, so that it is possible to express explicitly structures which potentially cannot be evaluated. This useful conceptual freedom is said to facilitate reasoning about strategies without pragmatic clutter.

The interpreter adopts the burden of pragmatism. It intervenes to constrain the program into conformance with the underlying (imperative) machine model, so that what is declaratively conceived becomes imperatively computable.

3. DECLARATION IMPLIES SPECIFICATION

The crux of the distinction between declarative and imperative languages is the 'declarative reading' afforded by the former in addition to, or instead of, the operational reading provided by the latter. The declarative reading epitomises the shift of emphasis of 'program' from prescription of operations to be performed, to definition of the objects to be computed, that,is, from computer behaviour to solution properties.

The intention, in the terms used in the declarative programming literature, is 'to separate logic from control components,[2] so that the solution logic or properties can be investigated thoroughly without commitment to a particular realisation and without reference to the behaviour of the machine. 'Control' or computational issues governing machine behaviour are handled within the language implementation. The declarative reading of the program is essentially a solution specification.

The literature treats 'specification' as related to but distinct from the 'program' that results from it. 'Specification' is taken to mean characterising what the solution entails, whereas 'program' means determining how the solution is to be reached. It seems as though declarative programming blurs the distinction between specification and program. Rather, the alignment of declarative program with specification reflects the disalignment of declarative program and imperative program; it is the use of the term 'program' which has lost precision. The consequence of 'separating logic from control' – of relegating control of computation from program to language implementation – is that the declarative language implementation injects information not in the program. Whereas an imperative language compiler does a reasonably direct translation between program and machine instructions, a declarative language translator disambiguates the program and chooses among possible realisations. In declarative programming, operational or algorithmic decisions are not avoided but are deferred to the language implementation.

Just as the declarative program is not complete without the language implementation, the language surface – the bits of language visible as lexicon and syntax which are used by the programmer – does not reflect completely the computational model underlying the language. Since imperative programs are explicitly a list of instructions, where syntactic units correspond roughly to psychological ones, imperative languages imply the computational model in the syntax. In contrast, aspects of a declarative model are inaccessible from the syntax and reside solely in the implementation. The 'declaration' alone cannot anticipate behaviour or performance; it has no 'hooks' into acutal computation.

Declaration, by nature, excludes algorithm: the 'process or rules for (esp. machine) calculation . . .' How, then, can we characterise languages that bear both a declarative and an operational reading? Clearly, programs in these languages entail more than declaration; they must contain some expression of algorithmic intent.

3.1 Levels of Information

It is possible to view these issues in terms of an approach to programming, so that classification reflects successive levels of information captured in a complete program. In this view, the programmer considers a problem and establishes the properties of its solution set. Methods for actualising the solution are appraised, and some strategy is elaborated. The programmer then fits the algorithm to the framework of control employed by the machine.

Consider as an analogy, the problem: 'Find me an aardvark'. Under the above view, the searcher first asks: 'What is an aardvark? That is, what constitutes a solution? An aardvark is a nocturnal mammal, native to the grasslands of Africa, with long ears and snout, that feeds on termites. Next the searcher asks: 'How can I find an aardvark?' That is, what strategy produces the solution? To find an aardvark, its habitat must be identified and located, its spoor distinguished and followed until an individual is found. Finally, the searcher expresses the strategy in terms of the available 'machinery', in this case, acquiring funding, making arrangements for visas, transportation, guides and so on.

Each stage of the approach adds a level of information, so that the progression can be viewed as a shift of orientation from solution specification to solution pursuit, or as an incremental translation that gradually constrains intention into conformance with operation. With respect to terminology, this view emphasises the continuity from declaration to instruction, so that the questions guiding language classification are: How much of the computational model is explicit in the language surface? Where is the transition from specification to instruction undertaken? Is the solution critically dependent on the language implementation?

Under this view, languages bearing both declarative and operational readings occupy a logical middle ground between declaration alone and machine control. They express both the solution space and algorithmic intent. Programs in such languages are specifications that inform execution; in the continuum between specification and program, these may be called executable specifications. The computational models underlying these languages are partially hidden; the general algorithm is available in the code, although it employs mechanisms concealed in the language implementation.

3.2 The Need to Reason about Behaviour

If we accept that declaration implies specification and excludes algorithm, how appropriate is the category 'declarative language' for languages which afford both declarative and operational readings? Does calling them 'declarative' obscure their nature, emphasising that they have a declarative reading at the risk of implying that they have only a declarative reading?

The value of executable specifications (which enhance declaration with algorithmic information) lies in the fact that, just because a language may reduce the degree to which a user must conform to the underlying machine and may hide matters of machine control, doesn't mean that all keys to program behaviour are expendable. Actually, there are computational issues (including aspects of efficiency) that are critical to the programmer, so that reasoning about algorithm behaviour is part of reasoning about the solution. A specification that admits only a declarative reading is in this sense incomplete.

Further, the issue extends beyond what the language affords, to how it is used. In a local, informal survey of functional programmers, we found that their typical approach is process-oriented. Although these programmers recognise and defend the declarative semantics, all agree that they exercise the operational semantics as well. Typically, they write their programs algorithmically and discuss them in operational terms.

We give an example drawn from experimentation, where (not unusually) the programmer demonstrated his operational use of the functional language Miranda by exploiting his knowledge about a discrepancy between the language definition and the implementation used: although the order of evaluation of guards is not defined in the language, the implementation causes a textual order evaluation. The example function determines whether a given year is a leap year or not:

```
leapyear x = True,  x mod 400 = 0
           = False, x mod 100 = 0
           = True,  x mod   4 = 0
           = False, otherwise
```

The definition is clearly intended as an ordered sequence; indeed, if the evaluation occurs in any other order, the function probably gives an incorrect result. Further, there is a hint of operational bias in the language design; the 'otherwise' implies a final catch-all. Only when all others have been tried and failed, choose this one.

Even with the more orthodox version, where all guards are mutually exclusive:

```
leapyear x = True,  x mod 400 = 0
           = False, x mod 400 = 0 & x mod 100 = 0
           = True,  x mod 100 = 0 & x mod   4 = 0
           = False, x mod   4 = 0
```

the guarded definitions are read as a sequence, perhaps with a tendency towards the if-then view, and the whole definition is read as a process of testing to determine which result is appropriate.

It may be worth noting that there is a process bias even in mathematical discussion, as in a formal proof. Whereas the mathematician recognises that all relationships hold at once (i.e., recognises the declarative nature of the proof), it is usual to read the proof as a process.

3.3 Declarativeness and the Accessibility of the Computational Model

The declarative reading is a by-product of hiding portions of the computational model in the implementation so as to shift the emphasis of 'program' from solution strategy to solution specification. Those languages which also afford an operational reading, admit the expression of algorithmic intent so that less is hidden in the implementation and more of the computation is available at the language surface. The basis of the distinction between declarative and imperative languages (and hence the appropriate basis for classification) lies in the significance of the particular implementation, that is, in where the computational model resides and in the closeness of the computational model to the language surface.

We propose that the directness of accessibility of the computational model (the degree to which it is reflected in the language surface) is the important dimension for classifying languages. It is more instructive to treat 'imperative' and 'declarative' as poles on a language class continuum than as strictly distinguished categories.

We have accumulated a set of related questions that draw out the relative placements of languages along this imperative–declarative continuum: how great is the translation distance between the language that the programmer sees and the computational model that makes it executable? How hidden is the computational model? Where is algorithmic intent introduced? How difficult is it to deduce the principal computational mechanisms from the language surface? How complex are the computational mechanisms provided in the language implementation? The more important the language implementation, the greater its role in providing algorithmic information and realising the program, the farther the language is along the continuum toward the declarative extreme.

4. THE CONTINUUM OF LANGUAGES

In this section, we explore this continuum of languages by reviewing critical features of a few popular programming languages. We show where these languages are sited on this continuum and present some examples coded in the various languages to highlight our argument. For the examples, we have chosen the problem of calculating the factorial of a number. The examples illustrate, in the shift from imperative to declarative style, the withdrawal first of explicit control and then of algorithmic information, leaving the injection of such information to the implementation.

Before offering the examples, we should point out that factorial is only defined for non-negative integers. Also, in many implementations, there will be restrictions caused by the finite range of computer numbers. For instance, our C solutions work only if the result is less than the maximum unsigned integer the machine will hold.

FORTRAN: FORTRAN is about as close as a high-level language can get to assembler; it has few varieties of control structure, few data types, and poor data structuring tools. FORTRAN is definitely an iterative, imperative language; recursion is not permitted. FORTRAN is strongly typed but allows variables to be declared implicitly by usage on the left hand side of an assignment. FORTRAN treats functions as distinct from data items.

Factorial: A FORTRAN solution.

```
      integer function factorial (number)
      integer number
      integer loopcounter, totaliser
      if (number .lt. 0) then
         factorial = 0
         return
      endif
      totaliser = 1
      do 10 loopcounter = 2, number
         totaliser = totaliser + loopcounter
10    continue
      factorial = totaliser
      return
      end
```

C: C is a flexible imperative language that affords good machine control, including input–output, good data structuring features, and (to a certain extent) higher level features, such as the construction of higher-order functions. C is a typed language, but there are many exceptions and faults. C supports recursion and pointer manipulation. In consequence of the range of expression offered, much C code is resistant to machine verification. Also, C programming depends heavily on the construction and use of good libraries.

Factorial: An iterative solution in C (a decidedly imperative solution):

```
unsigned int
factorial(number)
unsigned int number ;
{
    unsigned int result = 1 ;
    unsigned int count ;
    for (count = 2 ; count <= number ; count++)
        result *= count ;
    return result ;
}
```

Factorial: A recursive solution in C (which looks more like a functional program):

```
unsigned int
factorial(number)
unsigned int number ;
{
    return (number == 0) ? 1 : number * factorial(number
        - 1) ;
}
```

Although the above solution appears 'functional', control remains explicit in the 'return' statement.

LISP: LISP is a language based on function evaluation but which employs imperative constructs to control the process of expression evaluation. The list is its essential data structure. There are good function abstraction features but poor data abstraction features. Modern variants of LISP, e.g., Scheme, have introduced much stronger type checking and also lexical scoping of names to help in the production of modular software.

Factorial: A recursive solution in Scheme:

```
(define (factorial x)
  (if (= x 0)
    1
    (* x (factorial (- x 1))))))
```

Making use of the 'setq' (in Scheme 'set!') feature, the iterative solution can also be coded, but most LISP programmers would use the above functional algorithm.

Miranda: Miranda is a strongly but implicitly typed functional language with good pattern matching facilities. Functions use recursion and guarded commands. It also has implementation features such as tail recursion and lazy evaluation which programmers often use explicitly to make scripts cause efficient execution. Unfortunately, input–output is achieved using functions which are not referentially transparent.

Factorial: A Miranda implementation (definitely functional) based on guarded commands:

```
factorial x = 1                      , x = 0
            = x* factorial (x - 1) , otherwise
```

An alternative Miranda implementation (equally functional) based on pattern matching:

```
factorial 0       = 1
factorial (x + 1) = (x + 1) * factorial x
```

The first implementation in Miranda, whilst functional, exhibits control features directly. The guards force explicit choice between the options available, and this is exhibited explicitly. Selection is hidden in the second Miranda example. The mechanism of selection is pattern matching; the implementation chooses the relevant part of the definition of factorial by a mechanism not explicit in the code. The programmer is responsible for writing mutually exclusive patterns, hence the '(x + 1)'. If this is not the case, an implicit rule, such as textual ordering, is invoked, so that the program can be executed. We consider this latter implementation of factorial more concise and more declarative.

PROLOG: PROLOG is inspired by first order predicate logic. The major computational controls lie within the PROLOG implementation, whose computational model is unification. Backtracking is also required, as the essential feature of unification is searching. PROLOG has many control features for the programmer (for example, the cut operator) which allow the programmer to intefere with the searching of the implementation for particular pieces of software. Further, the user interface supplies implicit control of the program and, inscrutably, governs input–output. Procedure call facilities to other languages are usually provided. The order of rules is important to control; PROLOG systems always search rules in textual order. Also the and operator ',' provides sequencing. For example, all values must already be known before they can be used in an 'is' expression; the order of the comma-separated expression in the example is critical.

Factorial: Perhaps the only the implementation in PROLOG:

```
factorial(0, 1).
factorial(X, F):- XP is X − 1, factorial(XP, FP), F is X * FP.
```

This example highlights a number of hidden control-oriented features:

(1) The unnecessary PROLOG variable 'XP' appears because PROLOG does not allow expressions as parameters to functions.
(2) All arithmetic expressions must be evaluable, i.e., must contain no unknowns. This leads to the necessary feature that, although ',' is the Boolean and operation, it also provides sequencing: the predicate on the left must be dealt with before the predicate on the right.
(3) There are two assignment predicates: '=' and 'is'. The difference between the two is that the 'is' predicate forces evaluation and assignment immediately, whereas with '=' it may be delayed.

Obj: Obj is a executable subset of Clear. Obj specifications are written with the knowledge that the order of statements is important to the underlying implementation; statement order determines search order for the underlying term re-write system. Obj specifies functions in terms of their inputs and outputs and their relationships to other functions. Specification statements may use recursion and guarded commands.
The following Obj specifications assume an object Natural, defining the sort 'nat' and the functions 'succ', 'pred' and 'mult' and including an object Boolean, defining the function 'not'.

Factorial: An Obj specification using guarded statements:

```
obj Factorial/Natural
ops
  factorial: nat −> nat
vars
  n: nat
eqns
    (factorial(n) = 1                          if   (n = = 0))
    (factorial(n) = mult(n, factorial(pred(n)))if not(n = = 0))
jbo
```

Factorial: An Obj alternative using pattern matching:

```
obj Factorial/Natural
ops
  factorial: nat −> nat
vars
  n: nat
eqns
  (factorial(0) = 1)
  (factorial(succ(n)) = mult(succ(n), factorial(n)))
jbo
```

VDM: VDM is a specification language based on predicate logic which captures specifications in terms of pre- and post- conditions. A computational model is defined, and VDM specifications can be made executable. However, the specification itself makes no statement about how the function is implemented, only about the state of the computation before and after the function has been executed.

Factorial: A VDM specification:

$$\text{factorial}: N \rightarrow N$$
$$\text{pre-factorial}(n) = TRUE$$
$$\text{post-factorial}(n,r) \triangleq r = n!$$

It is important to note here that the definitions of the predicates are statements in mathematics, not in a computer language. In particular in this example, the ! (factorial) symbol is the mathematically defined one.

Z: Z is a specification language based on set theory. It is a mathematical system for reasoning about things for which no computational model has yet been constructed, although we understand such a thing is possible. Z specifications are captured in terms of pre- and post- conditions rather like VDM.

Factorial: A Z specification:

$$\begin{array}{|l}
\textit{factorial}: N \rightarrow N \\
\hline
\textit{factorial}\ 0 = 1 \\
\forall n \in N | n > 0 \bullet \\
\quad \textit{factorial}\ n = n * \textit{factorial}\ (n - 1)
\end{array}$$

Clear: Clear is a mathematical system for reasoning about things based on universal algebra. This specification language includes constructs considered mathematically sound and useful but for which a computational engine cannot be defined. Therefore, only a subset (Obj) of this language can be made excutable.

Figure 2 presents a map, siting these example languages on the continuum between 'imperative' and 'declarative'.

The specification languages (Clear, VDM, Z, OBJ) are concerned with solutions rather than with computation. They are based on models formalised in mathematics. These models are not really available at the language surface but must be learned elsewhere. Further, since they are concerned only with the outcome of computation, these borrowed models are incomplete; usually they do not embody strategies to drive the computation. The languages can be made executable by the addition of some computational model. Hence the user must apply his or her own additional computational information, or, to enable execution, the model as interpreted and embedded in the language implementation must be enhanced (e.g., by combining set theory with some set resolution mechanism) or constrained in order to construct a computation engine. Such enhanced models are usually complex and different from the models inherent in the language.

The Z specification given above can be shown to be satisfied by all the language implementations. The specification, in this form, describes neither the time/

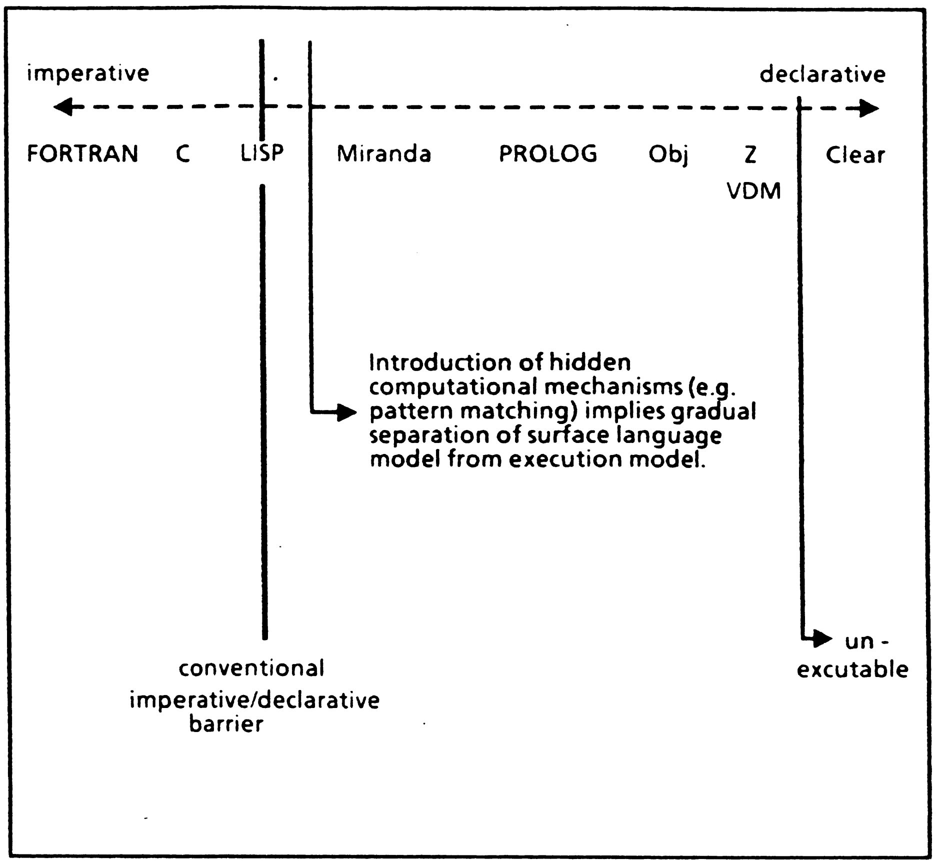
imperative
declarative
FORTRAN
C
LISP
Miranda
PROLOG
Obj
Z
VDM
Clear
Introduction of hidden computational mechanisms (e.g. pattern matching) implies gradual separation of surface language model from execution model.
un-excutable
conventional imperative/declarative barrier

Figure 2. The continuum of languages.

space behaviour nor the efficiency of the implementation. Therefore the specification cannot help us choose between implementations.

As with the specification languages, Miranda, LISP and PROLOG are based on a computational model distinct from the von Neumann machine, i.e., lambda calculus and first order predicate logic. Unavailable at the language surface, these models can nevertheless be learned from other sources without necessitating forays into language implementation details. Confusion may arise however, where the borrowed model has been adapted for use in the language, and the mechanisms which drive computation may still be obscure.

5. CONCLUSION

The declarative philosophy, as expressed in the literature and reflected in current terminology, is, we believe, mis-oriented. The basic difference between programming styles lies in the hiding of the computational model. The distinction between imperative and declarative, so evangelistically touted, is not exclusive but gradual, and the characterisation of both terminology and classification, reoriented in this way, is more powerful and more satisfying. Moreover, it emphasises that programmers will not be freed from implementation details until language designers provide comprehensive, workable reasoning models of their languages, models which accommodate reasoning about program behaviour.

6. ACKNOWLEDGEMENTS

We would like to thank John Campbell, Nigel Chapman, Paul Otto and Karen Paliwoda for conversations which helped to clarify many of the ideas in this paper.

REFERENCES

1. John Backus, Can Programming Be Liberated From The von Neuman Style? A Functional Style And Its Algebra Of Programs. *Communications Of The ACM21*, pp. 613–641 (1978).
2. Robert Kowalski, Algorithm = Logic + Control. *Communications Of The ACM22*, pp. 424–436 (1979).
3. D. A. Turner, Miranda: A Non-strict Functional Language With Polymorphic Types, in *Proceedings Of The IFIP International Conference On Functional Programming Languages And Computer Architecture*, Springer Verlag Notes In Computer Science – 201, Nancy, France (1985).