Modelling State in Mind and Machine

Meurig Beynon
Department of Computer Science, University of Warwick, Coventry CV4 7AL

Abstract

This paper discusses and illustrates the application of Empirical Modelling (EM), as developed by the author and his collaborators, to computer programming. The discussion centres on the perspective EM gives on procedural and experiential vs. declarative and logical approaches to knowledge representation. The illustration focuses on demonstrating the potential for pedagogical use of EM in teaching algorithms. It is based on a family of practical models that have been developed to teach the principles of heapsort.

1. Introduction

Handling state is a major theme in computer programming. The basic theoretical model of computation - the Turing machine - is procedural in nature; it describes algorithms in terms of sequences of changes to the cells of a tape. Despite this, the problems of interpreting and reasoning about procedural methods have prompted declarative approaches (such as functional [6] and logic programming [8]) based on stateless mathematical abstractions. In practice, these have failed to compete with procedural paradigms in most applications, and have embraced uncomfortable compromises with procedural elements.

Conceptually, the challenge is not so much to eliminate state from programming representations as to represent computational state in a comprehensible fashion. Consideration of state is essential in the detailed exposition of algorithms, and in the analysis of their complexity. For instance, in the process of sorting an array, the sequence of intermediate states defined by inspecting the array as it is progressively sorted is conceptually very significant. The use of abstract data structures, as illustrated by a method such as heapsort, offers one means to address this problem. It enables the algorithm designer to think in terms of state-based models that are more easily apprehended and interpreted. For instance, the array of values in Figure 1 is conceived as the heap depicted in Figure 2, where elements are associated with nodes, and significant order relationships between elements are recorded visually in the edges of a tree. To understand heapsort it is essential to appreciate the significance of the pattern of state changes that takes place in the tree representation as the array is sorted. In a conventional implementation of heapsort, the actual state changing activity exhibited by the program is more complex, and involves intermediate states that have no interpretation in the abstract view. For instance, when two values have been exchanged, a procedure will typically have to be invoked in order to update the status of the heap structure. The intermediate state that exists prior to this update is an artefact of the implementation that has no counterpart in the abstract description of heapsort.

The aim of this paper is to explore some aspects of the relationship between conventional computer programming and Empirical Modelling (EM) methods, as initiated through the development of the definitive ("definition-based") notation ARCA by the author in 1982. There are two sections to the paper: the first concerned with the significance of the Empirical Modelling Project in relation to paradigms for programming; the second with illustrating how our present EM tools and techniques can potentially be useful in teaching traditional programming topics.

2. Empirical Modelling and Programming Paradigms

The Empirical Modelling Project began with the design of the ARCA notation for interactive graphics. The initial motivation for designing ARCA was to seek an alternative to procedural and declarative methods of programming that combined the merits of explicit state representation with techniques to retain conceptual control over state. The definitive notations (spreadsheet-like notations with graphics) that stemmed from

this research, though introduced with this in mind, emerged as better oriented towards modelling than programming. A key concept in using such notations is exploiting the computer to construct cognitive artefacts that metaphorically represent observables in the external world, and reflect the way in which these observables are bound together through dependency relations when state-changing actions are performed. In this context, it is significant that

- definitive scripts primarily serve to represent a *state*, rather than to represent a behaviour
- the situated nature of these scripts is central to their interpretation as representations of state and potential behaviour.

(Consider the way in which the spreadsheet captures a state, but the possible states encountered in spreadsheet use - especially if we admit the possibility of extension and refinement through addition of new cells and definitions - are uncircumscribed.) *Prima facie*, it seems that definitive scripts do not provide an alternative paradigm for computer programming, but serve a totally different function, primarily concerned with human-computer interaction in connection with design and experimental activity.

When combined with agent-oriented modelling concepts, definitive scripts provide a basis for concurrent systems modelling and simulation. The fundamental elements in this approach to modelling are agency, dependency and observation. A most important characteristic of the models we construct using these principles is that the primary way to introduce behaviour is to directly simulate the interventions of the agents within the system. That is to say, the modeller explicitly decides what agent actions to select in a given state in order to make an appropriate transition of the system. In principle, this allows the modeller to design system behaviours, experimenting with potential agent interactions, and subsequently automating agent action where circumscribed patterns of state change can be identified. It is on this basis that the term *Empirical Modelling* (EM) was adopted.

As our understanding of the significance of EM has developed, the notion of observable has evolved to an unanticipated degree of sophistication. Initially most of the observables introduced in our models were visual features (like corners, doors and walls in rooms). It was apparent that quantities that were not directly observable, such as physical attributes of objects (weight, length) and other "scientific observables", that cannot be directly experienced (voltages, currents) were required to represent the interaction between agents within the model. It also emerged that the role of human agents in our models had in general to refer to what might be described as *conceptual* observables (e.g. "the train is ready to depart", "the cricket match has been won", "being in credit").

The whole idea of a conceptual observable is paradoxical. From a traditional empiricist perspective, *observables* might be identified as the most primitive elements for experience. For instance, one possible characterisation for these (cf. [5]) is as "discrete sensory particulars". In contrast, *concepts* are seen from a traditional empiricist perspective to be derived from the primitive elements of experience, and to involve more sophisticated cognitive processes. An empiricist perspective that better suits our modelling framework appears to be that of William James, whose "Radical Empiricism" [7] adopts a different stance on what is empirically given. For James, the givens are more than discrete sensory particulars; they embrace conjunctive relations that in particular account for direct apprehension of identities ("this is a different value of the same observable", "this is still the same cricket match"), and for knowledge as represented by relationships between experiences.

One of the key themes in James' work is that there is a profound distinction between the realm of 'pure experience' and the conceptual framework within which knowledge of preconceived reliable patterns of behaviour is specified and communicated. The direct and open representations of state associated with EM (and likewise with spreadsheets) are similar in character to James' world of pure experience. They are naturally associated with the uncertain, possibly personal and particular world with which empirical activity engages. In contrast, both procedural and declarative programming paradigms rely upon preconceived knowledge of behaviour. Procedural programming involves the circumscription of agency. Functional and logic programs are specified by making assertions about the relationships between states in these

circumscribed contexts. Though EM operates in a different context from either procedural or functional programming methods, it nevertheless can supply significant connections between the two paradigms that are otherwise obscure. The fact that the mental models of computer programs such as heapsort are framed in terms of highly abstract observables of just the kind that have already arisen in modelling human interaction in an EM framework is one indication of this.

It is now clear why it has taken so long to re-engage with the agenda that motivated the first research on EM. To be able to relate procedural and functional programming in a unified framework, it was necessary to first discover and recognise the significance of conceptual observables. What is more, the natural habitat for these observables is quite different from the circumscribed state-based worlds of conventional programming. In point of fact, EM principles and conceptual observables have much more to do with the cognitive processes that precede conventional program construction than with programming as traditionally practised (cf [4]). This theme will be illustrated in the next section of the paper.

2. Heapsort from an EM Perspective

The potential role of Empirical Modelling in developing computer programs can be illustrated by considering the heapsort algorithm. For brevity, some familiarity with the basic techniques of EM will be assumed (the EM website at http://www.dcs.warwick.ac.uk/pub/research/modelling can be consulted for more background). In particular, states of a system are modelled by definitive scripts, in which observables and dependency relations are respectively represented by variables and their definitions. For similar reasons, the details of heapsort will be omitted from this exposition (these can be found in any standard text on data structures, such as [1]). The aim of our modelling exercise is to show that the concepts involved in learning heapsort can be captured in an EM framework, and represented in such a way that a heapsort program can be derived from the model constructed to convey these concepts.

Three stages in the heapsort process are to be investigated using EM:

(1) constructing a visual model of a heap with which the user can experiment in order to understand the heap concept;
(2) constructing state-based models to represent the stages in the heapsort process, allowing the user to trace the steps involved in heap-building and sort extraction through a sequence of manual operations;
(3) introducing automatic mechanisms to carry out the appropriate sequence of steps.

It would be possible to give an account of the model construction that dealt with issues (1), (2) and (3) in sequence, but this is not the most appropriate organisation to adopt. From the EM perspective, the most effective way to present the model construction is to systematically introduce the underlying concepts as they might have been encountered in the discovery of the heapsort algorithm. (Compare the way in which the development of OXO models is described in [3].) For instance, a suitable account of heapsort addresses the following concepts in turn:

- using a complete binary tree on 7 elements to represent a 7-element array;
- identifying techniques for establishing the heap condition at every node of the tree;
- generalising the heap representation to associate array intervals with tree segments;
- identifying the pattern of heap transformation that is characteristic of heapsort.

The remaining sections of the paper outline the construction of models to address each of these issues. The result is a family of models that can be used to illustrate a variety of aspects of heapsort: as an environment for testing a student's understanding of the concept of a heap, and of the procedures used in heapsort; as a visualisation aid for the exposition of the heapsort algorithm; as a prototype for the implementation of heapsort in a conventional programming paradigm; as a platform for investigating variants of the heapsort algorithm. Each model is defined by a particular organisation of scripts and agents. The description of these models as given below is closely based on a realisation constructed with the tkeden interpreter [2,4], but makes use of an informal idealised syntax for definitions and agents that can be more readily understood by a reader unfamiliar with the notations used in practice.

2.1. Constructing a Visual Model of a Heap on 7 Elements

A student of heapsort who inspects Figures 1 and 2 has first to understand the relationship between the disposition of elements in the array in Figure 1 and the geometry of the associated tree in Figure 2. That is to say, the first element of the array is represented by the root of the tree, and the elements indexed by 2i and 2i+1 in the array are represented as the L and R children of the node represents the array element indexed by i. To make this correspondence accessible to the user, a dependency is established between the values of the array and the elements of the tree, so that changing the value of an array element simultaneously changes the value of the corresponding node. For instance:

val[root_of_tree] = array[1]
val[R_child_of_root] = array[3] etc.

In this simple environment, the relationship between array values and tree values can be investigated experimentally, so that the student can gain knowledge empirically through operating on the array elements, and observing the effect on the tree. (The issue of generalising the model constructed in the context of Figures 1 and 2, where there are only 7 elements in the array, to deal with arrays of arbitrary size, is beyond the scope of our immediate concern. One possible approach involves the concept of higher-order definition, but the primary function of the models we consider here is pedagogical, and the consideration of very large arrays is infeasible.)

In the tree, the basic observables are nodes and edges that metaphorically represent array elements and order relations between array elements that are significant in determining whether the tree satisfies the heap condition. A richer level of observation involves examining the values that are associated with the nodes in the tree, and the nature of the order relationships (<, =, >). In deciding whether the tree is a heap, it is further necessary to consider whether the heap condition is satisfied at each node - that is to say, whether the value associated with each node is at least as great as that associated with each of its children.

To model observation of this nature in a definitive script requires variables to represent the index and value of each node, to record the order relation that pertains on each edge of the tree, and to register whether the heap condition holds at each node. The index and value of a node are defined by explicit values, whilst the order relation and heap condition variables have values that depend on these. For instance, for the node with index i, the heap condition variable would be defined by:

HC[i] = (val[i]>=val[2*i]) **and** (val[i]>=val[2*i+1])

(subject to a suitable convention to deal with nodes with fewer than 2 children). Likewise, an order relation variable for the edge that joins the nodes indexed by i and 2*i would be defined by:

OR[i,2*i] = **if** (val[i]>val[2*i]) **then** 1 **else** (**if** (val[i]<val[2*i]) **then** (-1) **else** 0).

In our EM modelling environment, additional dependencies can readily be introduced to establish suitable visual conventions for representing these abstract conditions. For instance, the circle that represents a node and the edges between nodes can be coloured so as to reflect whether or not the heap condition is satisfied at a node, and to reflect the nature of the order relation associated with an edge:

colour_of_circle_at_node[i] = **if** HC[i] **then** black **else** white
colour_of_edge[i,2*i] = **if** OR[i,2*i] >=0 **then** black **else** white.

The virtue of such a model is that it allows the user to experiment with the assignment of values to nodes, and register visually the status of just those observables that are significant in understanding the heap concept. For instance, Figure 1 represents a heap if and only if all the nodes of the tree are coloured black. Such a condition can be independently monitored by attaching another high-level observable, defined by

is_heap = HC[1] **and** HC[2] **and** HC[3] **and** ... **and** HC[7].

In effect, the computer model developed in this way serves a similar function to the animation that a lecturer might sketch when informally explaining the heap concept. For instance, it is usual to demonstrate how the heap condition is affected by changing the value at a node, or exchanging the values at adjacent nodes. (Compare the EM development process described in [3].)

2.2. Establishing the Heap Condition: Control and Agency

Because user interaction with the model is unrestricted, it addresses issues broader than those encountered in a narrow study of the heapsort procedure. For example, the operations performed in heapsort serve only to rearrange the elements in the initial array, and indeed only involve the transposition of pairs of elements in the array, whereas the user can introduce new values at nodes in an arbitrary manner. A student who is unfamiliar with the heap concept will find the freedom to assign arbitrary values helpful in exploring what it means for a tree to be a heap. A student who wishes to understand heapsort has to be able to identify more restricted patterns of interaction with the tree model that can be reliably used to establish and maintain the heap condition at all nodes.

A simple technique that can be used to establish the heap condition at every node is: repeatedly exchange the value at a node with that of a child that has greater value, until no such exchange can be made. To prove that this technique is effective is an exercise in induction that is significant in formal understanding of the heapsort principle. An obvious optimisation involves exchanging the value at a node with that of the child whose value most exceeds the value at the node. To model this, it is necessary to introduce new observables to record the index of the child with greater value at each node i:

ICGV[i] = **if** val[2*i] > val[2*i+1] **then** 2*i **else** 2*i+1.

The introduction of observables such as ICGV[i] is associated with issues of control and agency that are complementary to the simple state model. ICGV[i] is concerned with a particular goal for manipulation of the tree model - viz. establishing the heap condition at node i. The role of such observables can be developed in different ways. One possibility is to introduce a dependency relation, so that the node with index ICGV[i] is distinguished visually in Figure 1. By such visualisation techniques, the student can 'learn' how to establish a heap by following a set of abstract rules based solely on the perceived colours of nodes and edges in the diagram. Note that this automatic mode of rule-based execution does not demand any deep insight into the significance of the process of heap creation. A student who executed the heapsort algorithm with the aid of such visual cues would be acting no more intelligently than a car driver who drove off on a green light even though the road was obstructed. Observables such as ICGV[i] are nevertheless significant both as a way of drawing a student's attention to what is to be understood, and in monitoring automatically whether such understanding has been gained.

In the modelling environment currently used for EM, it is possible to introduce triggered procedures that are invoked when the values of variables are changed. These can be used to simulate the actions of agents in response to changes in the values of observables. By such means, it is easy to connect user selection of tree nodes via the mouse with operations on the associated values. It is also possible to attach an agent to each node of the tree in such a way that it performs an appropriate exchange of values at nodes i and ICGV[i] whenever the heap condition HC[i] at node i is violated. The use of these simple control devices in various combinations supplies numerous ways to give insight into the process of heap construction. For instance, the model can be set up to establish the heap condition automatically whenever a value on the array is changed, either in such a way that all the necessary exchanges at all nodes are performed without user intervention and without visualisation of the intermediate states, or in such a way that one exchange occurs each time the user clicks the mouse. Under one control regime, the selection of the next node at which to perform an exchange is made by the user, who need not be constrained to select an appropriate node (viz. one at which the heap condition was violated). Alternatively, the choice of node might be non-deterministic, according to which of several appropriate nodes was selected for carrying out an exchange of values at each step.

The construction of these different control models is significant even though the ultimate pattern of control in heapsort is automatic, relatively simple and deterministic. The different control regimes illustrate important observations that help to inform the control mechanisms adopted in the heapsort algorithm.

2.3. Heap Generalisation: Associating Tree Segments with Array Intervals

To refine the model for application in the heapsort process, it is necessary to introduce bounds on the segment of the array within which the heap condition is assessed. This can be modelled by adding new observables first_of_heap and last_of_heap to the script described above, and modifying the definitions of the current observables to take account of whether the indices of nodes referenced lie between these two limits. A useful refinement of the visualisation to accompany this eliminates from the tree those edges incident with a node outside the range of the heap. The construction of such a model involves a systematic revision and extension of the previous script.

The primary modification of the model of the 7 element heap is the adjunction of new observables to record whether an index is currently in range:

in_heap[i] = (first_of_heap <= i <= last_of_heap).

The definitions of the heap condition, and of the index of the child with greater value at node i have to be revised. For instance, the new heap condition is defined by

```
HC[i]  = not in_heap[i]
         or (
              in_heap[i]
              and (  ( not in_heap[2*i] ) or ( in_heap[2*i] and OR[i,2*i] >= 0 )  )
              and ( (  not in_heap[2*i+1] ) or ( in_heap[2*i+1] and OR[i,2*i+1] >= 0 ) )
         ).
```

The attributes of edges and nodes are likewise modified to reflect membership of the heap:

colour_of_circle_at_node[i] = **if** in_heap[i] **then** (**if** HC[i] **then** black **else** white) **else** invisible

colour_of_edge[i,2*i] = **if** in_heap[i] **then** (**if** OR[i,2*i] >=0 **then** black **else** white) **else** invisible

The alternative definitions for HC[i], ICGV[i], colour_of_edge[i,2*i] etc needed to change the bounds of the heap can be stored as files of dependencies between observables. By including different files, it is possible to transform between simple and more complex models. This is especially useful when trying to trace errors in scripts, making it possible to carry out experiments in two different contexts.

From a pedagogical viewpoint, one of the advantages of the constructive nature of EM is that it allows experimentation with a wide variety of objectives. It is possible to explore the effect of changing the range of heap and the values at nodes both in order to test that the correct definition of the heap condition has been developed, and in order to confirm that the concept of restricting the heap to an array interval is correctly understood. In this context, there is a strong parallel between 'debugging the model' and setting up a scientific instrument. As in developing instruments, experimentation is significant when checking that observables and dependencies have been correctly identified, and are being appropriately registered.

EM also resembles the experimental method in science in other respects. There are analogues for principles such as isolating different experimental parameters, attaching additional instrumentation, and exploration of state beyond the range of interaction eventually found to be of practical interest:

- particular subsets of observables can be extracted for independent experimentation. This allows a separation of concerns, so that (for instance) the window layout, tree layout and visualisation conventions can be examined independently;
- script fragments and additional agents can be attached to the model to monitor particular aspects of its state and behaviour, for instance, to flag the occurrence of an intermediate state in which a particular sequence of array elements occurs in sorted order, or to record how many times the heap condition is violated at a particular node;
- the values of parameters that are introduced to the model in order to gain insight are not necessarily required in subsequent application of the model. For instance, the values of first_of_heap and last_of_heap can be set to satisfy 1 < first_of_heap <= last_of_heap < 7, even though this condition is never realised during the execution of heapsort.

2.4. Constructing State-based Models of the Heapsorting Process

Once the model of the heap described above has been constructed, it becomes possible to trace the steps in the heapsort algorithm. This is a two stage process: an initial process of progressively constructing a heap on the range [first_of_heap,7], where the first_of_heap ranges downwards from 4 to 1, followed by a process of sorted sequence extraction, where a heap is maintained on the interval [1, last_of_heap] and last_of_heap ranges from 7 down to 0. The extraction of an element from the heap is performed by decrementing the last_of_heap index, and exchanging the value formerly indexed by the last_of_heap with the value at the root. Heap construction and maintenance is performed using a sequence of operations that defines the *heapsift* procedure. The key concept in heapsift is that, when necessary, the heap condition can be established at a node by exchanging the value at the node with that of the child node whose value is the greater. The model of the heap admits all the transformations that the user needs in order to simulate the heapsort algorithm by a systematic process of exchanging values at nodes following a predetermined protocol, and changing the range of indices in the heap under consideration.

When the heapsort algorithm has been identified in this fashion as a deterministic process that can be automated, the observation of the heap is no longer confined to examining the local effects of changing a single value or modifying the range of the heap. In effect, the process of tracing the algorithm manually corresponds to identifying a new regime of observation of the heap, in which the pattern of state changes is viewed as a whole, as a computational behaviour that can be prescribed to follow explicit rules. Once this regime has been identified, the user has understood heapsort as an algorithmic process, and is in a position to automate the pattern of state changes in accordance with suitable protocols for redefinition. For instance, in the phase of the algorithm that builds the heap, there is a pattern of state change defined (4 >= k >= 2) by:

if is_heap[k,7] **then** first_of_heap = k-1
at all nodes i **in the range** [k-l, 7]:
    if not HC[i] **then** exchange val[i] with val[index_of_greater_child[i]].

The idea behind patterns of state change as sophisticated observables is well illustrated by considering the observation that distinguishes an expert Rubik cube solver from a novice. The expert who inspects a configuration of the cube can directly apprehend a sequence of operations that can be used to transform the current state of the cube towards the solution state. For the novice, only the latent primitive operations upon the cube are self-evident. The experimental process by which Rubik cube expertise is developed resembles the process by which the user of the heap model is led to discover the heapsift procedure. A simple experimental context that might be used to introduce a student to heapsift would involve substituting new values for the root element in a 7 element heap, and inviting the student to restore the heap. There are many ways in which the restoration could be carried out, but none more simple and generic than heapsift, whereby the value at a node i that violates the heap condition is successively exchanged with the value indexed by ICGV[i]. It is characteristic of this process that the primitive exchange step for heap maintenance is invoked in a systematic way, at a sequence of nodes on a path emanating from the root. In this respect, it is an algorithmic process quite different in character from the parallel non-deterministic invocation of agents that perform primitive exchanges at arbitrary nodes at which the heap condition is violated. Analysis of this kind is necessary to expose the empirical processes that operate in developing both the control and the state aspects of an algorithm. It also highlights hidden assumptions upon which an algorithm such as heapsort relies: for instance, the fact that the values attached to nodes are not subject to change through independent influences, such as the passage of time.

The distinction between local observation of heap state, and the state of the heap as viewed in the context of executing the heapsort algorithm can also be illustrated by the addition of a new observable to the heap:

is_sorted = (val[1] <= val[2]) **and** (val[2] <= val[3]) **and** ... **and** (val[6] <= val[7]).

By establishing a dependency relation between the label of Figure 1 and the observable is_sorted, it is possible to arrange for the annotation of Figure 1 to read "Sorted array of elements" whenever the array

elements are in sorted order. Logical as this seems, it is not appropriate to label the array as sorted in all circumstances. For instance, when an array of values that is coincidentally sorted is submitted as an input to heapsort, the execution of the algorithm perturbs the sorted order. The two modes of observation invoked here distinguish between two semantically distinct assertions: "the array is coincidentally in sorted order" and "the array has been sorted through the execution of the heapsort algorithm".

3. Concluding Remarks

As presented here, the main interest in the application of EM is pedagogical rather than practical. That is to say, the EM process constructs an animation that serves a useful purpose in demonstrating the heapsort principle, but does not directly provide an executable variant of the algorithm for general use on a large input array. Though it is beyond the scope of this paper to examine practicality issues in more detail, there has been relevant research in this direction. Once the pattern of interaction involved in updating the model has been appropriately circumscribed (i.e. the visualisation requirement is discounted, and the scope for the user to perform experiments - as in departing from the prescribed steps of the algorithm, or redefining the heap condition - is eliminated) there are techniques for analysing dependency and agency that can be used to derive conventional procedural programs semi-automatically [2]. A promising alternative approach to efficient implementation involves implementation of the dependencies at a much lower level of abstraction. This is possible using the architecture of the Definitive Assembler Machine (DAM), where dependencies between machine words can be established and maintained. This implementation strategy involves techniques such as compiling dependency relations between integers and booleans into families of simpler dependencies that use simple low-level operators to establish definitive relationships between the contents of machine words. An interesting feature of such compilation is that it generates low-level code, but nonetheless generates machine states and transitions in execution that are conceptually isomorphic to those that are introduced at the highest level of abstraction.

The research described here is also of interest as a first experiment in collaborative use of a development environment based on running a client-server network of tkeden interpreters. In this respect, it shows promise both as a way of sharing knowledge of EM principles, and of studying and teaching heapsort.

Acknowledgments

I am indebted to Simon Yung for developing the tkeden interpreter and for implementing heapsort on a virtual low-level definitive architecture, to Richard Cartwright for implementing DAM on an Acorn Risc PC, to Steve Russ, for many stimulating discussions on the subject of EM and programming, to Patrick Sun for constructing a client-server variant of tkeden, and to Amanda EM Wright for her collaboration and help in developing the family of models that has been outlined in this paper.

References

1. A V Aho, J E Hopcroft, J D Ullman *Introduction to Algorithms and Data Structures* Addison-Wesley, 1982
2. J Allderidge, W M Beynon, R Cartwright, Y P Yung *Enabling Technologies for Empirical Modelling in Graphics* Dept of Computer Science RR#329, University of Warwick 1997
3. W M Beynon, M S Joy *Computer Programming for Noughts and Crosses: New Frontiers* Proc PPIG'94, Open University, 1994, 27-37
4. W M Beynon *Empirical Modelling for Educational Technology* Proc Cognitive Technology '97, IEEE 1997, 54-68
5. Graham Bird *The Arguments of the Philosophers: William James* Routledge and Kegan Paul, 1986
6. P Henderson *Functional Programming: Application & Implementation* Prentice-Hall International, 1980
7. William James *Essays in Radical Empiricism* Bison Books 1996
8. R Kowalski *Logic for Problem Solving* North-Holland, 1979