New Paths for Programming in Theory and Practice

Meurig Beynon

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

An informal paper for the IBM Warwick Software Development Laboratory 24/9/92

ABSTRACT

Computer Programming has been under development for four decades. This period has brought great changes both in programming methods and in computer hardware, but major issues remain unresolved. In particular, a tension between foundational principles and engineering practice is evident in many areas of computer science. This paper argues that this stems from an inadequate formal concept of computation, and briefly introduces a research programme, established at the University of Warwick, that is aimed at developing a new scientific and practical framework for programming.

Introduction

Such is the scope of modern Computer Science that is hard to see connections between different areas. The development of the subject resembles the exploration of a large continent with great geological and cultural diversity. The theme of this paper is most easily appreciated through this geographical analogy - it traces a source of conflict that traverses the subject like a geological fault. It concerns a tension between principles and pragmatism, manifested in different ways in different areas, that has its origin in our present inadequate understanding of the concept of computation.

This paper is in two sections. Section 1 discusses the implications for programming of conflicts that arise in four research areas:

abstract programming paradigms,
complex software systems development,
programming techniques for AI and for end-user applications,
mathematical foundations of programming.

Section 2 briefly outlines theoretical and practical progress we have made in developing a new principled approach to programming.

1. Theory vs Practice

1.1. Programming Paradigms

Programming paradigms are the concern of the programmer and the program language designer. In this area, there is a rift between formal and pragmatic approaches.

Forms of programming that make use of stateless abstractions, based upon classical mathematical foundations, have been advocated as a formal ideal. Pure declarative languages, such as idealised variants of functional or logic programming, are examples. John Backus, in his 1979 paper "Can programming be liberated from the Von Neumann style?" [1], analysed some of the key issues involved in exploiting declarative programming principles in practice. Backus observes in particular the need for history sensitivity in a programming environment, and argues that this cannot be achieved without introducing some form of state. Subsequent developments in this area have not addressed Backus' concerns effectively. The adaptation of declarative languages for practical use (as e.g. in conventional Prolog programming) has involved compromises that fall far short of the purist formal ideal.

Pragmatic approaches to programming have in contrast gained ground in recent years. In 1979, Backus addressed the theme of functional programming as a potential successor to traditional procedural approaches. In 1992, the concept that a particular programming paradigm could be universally applicable is out of favour. The modern range of computer applications and architectures is partly responsible for this. The practical programming tools that dominate the market are based on variants of procedural - state-based - programming. They include concepts such as object-oriented models and spreadsheets with no formal pedigree. Attempts to formalise such techniques retrospectively (as e.g. in giving formal semantics to concurrent object-oriented models) give little insight into their practical power and utility.

It is no longer fashionable to quibble about the choice of programming paradigm in the context of modern software development. A paradigm is simply regarded as a means to specify a computation, selected on the basis of convenience rather than principle. In practice, the glib philosophy that particular paradigms suit particular problems ignores the fact that all kinds of problem are typically represented in a large scale systems programming task, and that the integration of different paradigms is often conceptually and technically unsatisfactory.

This paper questions current orthodoxy in two respects. We shall argue that

the concept of computation that has guided the development of programming paradigms in the past has been too narrow
the issue of finding a programming paradigm that meets Backus' criteria cannot be evaded; if ignored, it is re-encountered in the programming process (§2.2).

Our position suggests a cynical explanation for present-day suspicion of a universal programming paradigm, with its emphasis upon program design rather than programming style. Unless our concept of a programming paradigm is broad enough to embrace "programming as a process", and until wehave developed a paradigm that is powerful enough to simplify this process, the problems of integrating programming styles in a principled fashion seem trivial beside the complexity of the process itself.

1.2. Complex software systems development

Issues of software development are the concern of system designers and software managers. In this area, there are yet other conflicts between formal and pragmatic approaches. These operate at a higher-level of abstraction, and relate to programming as a design process.

Formal approaches to large-scale software specification centre around the use of mathematical formalisms for specifying the computation to be performed by the components of a system. It is widely acknowledged - see for instance [3] and [7] - that current techniques have yet to meet the challenge of software specification for (in particular) reactive systems, and that the development of such specifications necessarily involves an iterative process of preliminary design, simulation, analysis and revision. The most significant problem in the use of mathematical formalisms for specification is the need to circumscribe the system behaviour, i.e. to arrive at an entirely unambiguous and suitably comprehensive behavioural model. It is apparent that the static abstractions (albeit circumscriptions of behaviour) that constitute a formal specification are quite different in nature from those we should need to describe the steps in the design process. (It may be useful here to consider the similar distinction between the formal statement of a proof and the exposition of this proof.)

Pragmatic approaches to software development usually focus on those aspects of the process that precede circumscription. They typically rely upon informal methods of state-based modelling of the requirement. A state-based modelling technique that is appropriate for the design process would ideally have several characteristics:

scope to represent exceedingly complex state-transition models,
means to model a system in a way that is intelligible to the designer and conforms so closely to the system as conceived as to have explanatory power,
power to represent incomplete models so that they are easy to enhance or revise.

A major issue for a state-based modelling technique is reconciling an unambiguous description of behaviour with intelligibility and flexibility in design. For instance, Petri net models have an unambiguous operational interpretation, but are difficult to relate to the application in general. In contrast, concurrent object-oriented models may be closely related to the application, but have an ambiguous operational interpretation.

The role that formal methods can play in software development remains controversial. The comprehensive use of formal specification is seen by many as unrealistic, but more modest formalisation can be viewed as a vehicle for stimulating disciplined analysis of requirement issues. In a recent paper [7], Harel emphasises the need for formality, but at the same time focuses almost entirely upon the role that visualisation, simulation and testing can play in resolving the problems of reactive systems specification. A major concern about techniques that rely on state-based exploration as a means to comprehend complex systems is the absence of good criteria for discriminating between one model and another. In software requirements, as in programming, it is not usual to argue the existence of a principled state-based modelling technique that is fundamentally better conceived than any other. As Davis explains in [4], many state-based techniques are proposed, there is no consensus about which are the most appropriate, and his general advice is to be eclectic and indulge in what Smith describes in [10] as "promiscuous modelling".

There are many parallels between the old debate over programming paradigms, and the new controversies over requirements specification. What is most significant is that the attention has shifted from the operational semantics to the real-world content of programming models. This is particularly

relevant to the role of programming in AI research, and with research aimed at making programming tools more accessible to the non-specialist user.

1.3. Artificial Intelligence and Applications

In AI and applications, the programming issues raised are user-oriented. They include:

how "intelligent" do the responses of a program appear to be?
how can programming activity be made more accessible to the non-specialist?
how can programs be written so as to be more easily understood?

1.3.1. Artificial Intelligence

The conflict between formal and informal approaches to AI is well-established as the "neat" vs "scruffy" controversy.

Neat AI has its basis in logic. The logicist thesis - that intelligence is rooted in sophisticated forms of reasoning - has been the subject of vigorous debate [9]. To make the logicist thesis plausible, more and more sophisticated variants of logic have been developed, some of which have characteristics (such as non-monotonicity) quite unlike classical logic. None of these variants can remain formal, in the accepted meaning of the term, without presuming circumscription of the problem domain in some form Such circumscription is the basis for criticism of a logicist position. It is implausible that preconceived patterns of response, however ingeniously constructed, can mimic intelligent behaviour. The concept of intelligent behaviour itself suggests an unforeseen element, as when we use a familiar object in a unexpected role (e.g. a knife as a screwdriver).

Scruffy AI is characterised by ad hoc representation techniques and programs. Its products include techniques and programs that have been very successful for specific classes of problems or in narrow domains of application. The weakness of the scruffy AI position is that it relies upon modes of representation that follow no coherent general pattern, and are typically informal to the point of vagueness. In defence of scruffy AI, it can be argued that the practical tools that originated from neat AI (such as expert systems and theorem provers) involve many extra-logical ingredients.

The neat vs scruffy controversy has serious implications for the agenda of AI research. Logicism can be seen as promoting a fundamental misconception about the nature of intelligent knowledge, whereby contemplation of a circumscribed object displaces open-ended experience. The ability to observe and experiment are an essential part of the behaviour of an intelligent agent - the fact that we cannot represent experiential knowledge formally at present should not divert us from its fundamental significance. In this context, it is important to recognise that the predictions of a theory, however subtle, can never confound expectations based on previous experience of its use. In contrast, the results of observation and experiment can in principle be arbitrarily unexpected. The fact that many observations and experiments are reliable in practice is a basis for faith, not a matter of reason.

1.3.2. Applications

In programming for applications, it is necessary to bridge the gap between data representations that can be processed by a computer, and those that can be appreciated by a non-computer-specialist familiar

with the application. This issue has been the central focus of research into data-base and user-interface design.

Constructing good formal models of data-bases is exceedingly challenging. In 1978, Kent [8] devoted an entire book to the problems that beset traditional hierarchical, network and relational data-base models. In his conclusion (pl93) he observes "Perhaps it is inevitable that tools and theory never quite match." and (p.vi) "implicitly suggests [that] there is probably no adequate formal modelling system [for the representation of information on computers].". Amongst more modern developments in this area, it is those approaches that incorporate spreadsheet-like data representation principles (cf the database RL/1 [6]) that have most promise.

The difficulties of adapting declarative approaches to programming for interaction are well-recognised [12]. They are connected with two issues:

- declarative input-output abstractions for computation are not well-adapted for representing the state of input-output devices,
- it is hard to circumscribe the computational activity in user-computer interaction.

At a more fundamental level, the traditional formal concept of a program is ill-matched to data-base interaction. The most important state-changing activity in a data-base is user-driven, and corresponds to updating the model in response to external events (such as the return of a book or the acquisition of a new book in a library data-base). In this context, the real-world interpretation of data models inside the computer is the most significant concern.

Pragmatic approaches to real-world data representation and manipulation have made considerable progress since Kent's 1978 analysis. The relationship between conceptual and computer data models is much better reflected in integrated data-bases and spreadsheets in business software and in object-oriented data-bases. The widespread availability of windowing environments has had a major impact on the way in which data models are represented to the user. The most significant aspect of this development is the shift of emphasis from "understanding through contemplation of a document" to "understanding through exploration of an environment".

The latent weakness and danger in these pragmatic developments is the absence of an adequate explanatory framework. Modern applications software implicitly invokes new modes of data reference and abstraction associated with new concepts of views, agents, and computation. In some way, the foundations of programming have to be extended to account for this.

1.4. Foundations

The foundations of programming are the concern of the philosopher. The concept of formality has been so well-established through the classical work of mathematical logicians that its invocation in connection with programming is rarely questioned. Our analysis indicates that as we focus upon the essential relation between a program and its real-world context, formal approaches tend to impotence, and pragmatic approaches to anarchy. Better foundations for programming demand principles beyond the scope of our current notion of formality.

The philosopher and computer scientist B. C. Smith is a strong proponent of this view. In his contribution to the logicism debate [10], Smith argues the need for a reconstruction of classical logic

that respects some basic principles concerning form and content. The characteristics required of such a reconstruction include:

- context dependent interpretation of utterances,
- richer interaction between content and form,
- models that are more narrowly discriminated (cf §1.2).

Smith's contention that "formality, in the end, reduces to notions of physical realisibility" is particularly relevant to our thesis in this paper.

The history of mathematics offers additional insight into ideas of formality. The arithmetisation of geometry and analysis in the 19th century is recognised to have changed the character of mathematics, creating a new distinction between its "pure" and "applied" aspects. The intuitions surrounding variables prior to this process were unlike those that mathematical logic has imposed. Newton's use of fluents exemplifies this. In our view, programming must reinstate variables with richer associations. Indeed, we shall argue that pragmatic programming methods, such as the use of spreadsheets, have already indicated how this may be done in a principled manner.

In our current computer science culture, two complementary attitudes obstruct progress towards better foundations for programming. Armed with modern mathematical logic, formalists tend to be complacent about the need for any more expressive foundational framework. Encouraged by practical progress, aware of the theoretical challenges, irritated by the limitations of current formalism, pragmatists tend to be sceptical about the prospects for useful foundations. That neither conventional formalism nor blind pragmatism is itself sufficient is suggested by the following unsolved research problems, each of which depends crucially upon a better account of the relationship between program form and content:

- write programs that are easy to interpret,
- write interactive programs that adapt to a user,
- integrate requirements analysis and specification,
- model design activity, where the user introduces knowledge incrementally,
- program a robot to relate the symbols in its internal model to sensory input.

2. Towards a new programming paradigm

The issues raised above provide the context for a research programme that has evolved at Warwick University over the last decade. Despite the foundational - even philosophical - nature of our concerns, our programme has from the first been guided and motivated by practical software development. The technical aspects of our work and its relationship to fundamental programming concerns are described and illustrated in detail elsewhere (see the attached bibliography). In this section, we give a brief resumé of current progress and status.

The kernel of our programme has been the development of a paradigm for user-computer interaction that has essential ingredients in common with the spreadsheet. In this paradigm, a set of definitions (or definitive script) is used to specify the state of an interaction between the user and the computer. These definitions link the values of variables in the same way that the values of spreadsheet cells are linked by formulae. The variables in a script typically designate quantities that are depicted on the screen: for instance, parameters that determine the length or location of geometric entities in a picture, or that specify the mode of presentation of a picture in a window. To change the state of the display, the user

enters a definition, whether to introduce a new parameter, or simply to redefine an existing one. As in a spreadsheet, the redefinition of a single variable conceptually entails instantaneous synchronised changes to the values of all dependent variables.

The principle of using systems of definitions to represent the state of a geometric model, as demonstrated in our first software prototype - the ARCA interpreter - in 1982, was first introduced by Wyvill in 1975 [11], and has since been widely used in interactive graphics and design. Our contribution has been to exploit the representation of state by definitive scripts in a systematic and general manner. This generalisation has two aspects:

the design and development of definitive notations,
the use of definitive state representations to specify interaction between arbitrary computational agents.

For convenience, abusing the word "definitive" to mean "definition-based", we have adopted the term definitive programming to describe our approach. Its characteristic feature is the principled use of definitive state representations to describe computation.

A definitive notation is a simple formal language in which definitive scripts can be formulated. The algebraic expressions that appear as defining formulae in such scripts are evaluated over an algebra of data types and operators - the underlying algebra of the definitive notation. The variables that appear in a definitive script have an informal semantics quite different from the variables that are used in conventional programming languages. In the programming process, they denote values that correspond to such observations as the programmer might make in building a model of the requirement. In this respect, they differ in nature from traditional data types in programming languages, which serve as private internal data representations.

Definitive programming is subtle and unconventional, both in concept and process. As suggested in §1.1, it conflates state changing activity associated with program design with conventional program execution. The practical advantages of this in systems for geometric modelling and animation are well-recognised [12]. In essence, we describe all modes of computational activity by models in which many agents concurrently redefine variables in a definitive script.

The principles that underlie our programming method distinguish it from existing approaches, including those that are otherwise "agent-oriented". In our framework, programming is associated with a form of system modelling that leads to models between which we can discriminate objectively to an unusual degree (cf §1.3). To formulate our programming concept, we must examine the fundamental premises of computation more critically.

Computational activity is not confined to computers. A child may be taught to multiply 3 by 9 by turning down the third finger of her left hand and interpreting the result. In reactive systems, the interaction between human agents, sensory devices, computers and electro-mechanical components involves many transactions between state-changing agents where similar protocols for observation and conventions for interpretation have to be designed and implemented. The computational agents in the classical theory of computation are devices - like the Turing machine - with a repertoire of state-changing actions that is carefully preconceived, entirely circumscribed, and reliably implemented. The conventions for interpreting its state-changing actions are established once-and-for-all prior to computation. The traditional theory of computation presumes the prior existence of computational agents together with established protocols for observation and conventions for interpretation.

The formidable problems of reactive systems specification referenced in §1.2 stem from the need to embrace dynamic specification of computational agents and their interaction in the programming process. To achieve this, we have to describe how each computational agent should react to another in terms of how its state is changed or it changes the state of other agents. This involves expressing the effects of one agent's actions upon another in explicit state-based terms, i.e. by specifying how state change is observed and communicated. Most crucially, if our model is to be faithful to the practical capabilities of the computational agents, it must reflect what observations one agent can make of another and how these observations are indivisibly bound by functional dependencies in transition. It is in this role that definitive state representations are indispensable.

The nature of our agent-oriented modelling approach to programming can best be understood by analogy with a conventional engineering exercise. In designing and assembling a complex device, such as an engine, we may first assemble and test combinations of components independently. To be satisfied that the design will work, we may perform experiments on these assemblies, providing certain stimuli and observing the responses. When the engine is finally assembled, these same stimuli may be given by other autonomous components. By exploring the behaviour of subcomponents at first-hand in this fashion, we can then account for the behaviour of the system as a whole.

This approach strongly resembles the object-oriented design strategies due to Booch [2] and Deutsch [5], but has subtle and significant differences. By dealing with observations rather than objects, we can take account of synchronisation issues crucial in the semantics of assemblies and components. In effect, by expressing how changes in observations are coupled in change - as in a mechanical linkage, we represent the indivisible operations associated with each computational agent. The distinction between our model and an object-oriented model is best appreciated by considering the design of a entirely new engineering product, where we have first to conceive the functions for subassemblies and the observations that are essential in interpreting their cooperative behaviour. In contrast, an object-oriented model presumes sufficient knowledge of the system decomposition and interaction to allow encapsulation.

The use of a definitive script to represent user-computer interaction can be viewed as a form of "single-agent" programming. This is a programming paradigm similar to that used in spreadsheet packages, where all the significant state-changes are user-driven (cf §1.3.2). The generalisation from single-agent to multi-agent programming is essentially explained by the strategy for system development we have described. The design of a complex system reduces to a series of experiments, each documented by a definitive script, representing the designer acting in the role of participating agents. Such programming activity is in effect as much machine design as software development.

Figure 1 is an sample screen from a vehicle cruise-control simulation that has been developed using our agent-oriented modelling principles. The visualisation aspects of the simulation are described using two definitive notations: DoNaLD - a definitive notation for line drawing, and SCOUT - for screen layout. Definitive scripts are used to link internal parameters, such as the speed to be registered on the speedometer, to the location of the speedometer needle on the screen display. They also specify the positions and characteristics of the windows. For example, the speedometer window is specified in such a way that if the speedometer is redesigned to display speed on a different circular segment, the window frame will be appropriately adapted.

In simulation the display is updated by the autonomous computation associated with the dynamic model of the vehicle. The sensitivity of windows is defined in such a way that the buttons and sliders initiate

appropriate transitions under the control of the user. The effect of introducing this user-interface is to provide a restricted environment for the simulation user that precisely matches the perceptions and privileges of the driver. When a button is pressed, an appropriate re-definition is eagerly generated by an agent. In developing the simulation, variable redefinition was at first carried out by direct textual input into the definitive script , and this option for intervention beyond the scope of normal use of the simulation still exists as an alternative to direct manipulation in the final version of the prototype. For instance, it is possible to intervene to redefine the geometry of the vehicle whilst the simulation is in progress.

In the model of the vehicle dynamics, the components of the engineering model, such as the speed transducer, the cruise cutout and the vehicle itself are represented by agents. The equations of motion of the vehicle are represented by a definitive script in which the variables represent analogue quantities. Such variables illustrate a process of idealisation involved in the use of definitive scripts, whereby Platonic entities (such as points and lines) are represented on the display to a degree of approximation that can in principle be chosen with arbitrary accuracy. In the simulation, the values of these variables are updated by integration with a step-size that - conceptually - is a parameter that is chosen to be arbitrarily small. Other aspects of the simulation, such as the specification of the speed transducer, also involve integration - in this case to reflect the process by which the actual speed of the vehicle is converted into a sampled signal. In this context, the step-size of the integration is a design feature of the model, chosen to satisfy the requirements of the engineering model with respect to feedback control for the cruise speed maintenance. It is on the basis of such explicit modelling that our simulation acquires explanatory power (cf §1.2).

Our simulation program convincingly demonstrates the expressive power of definitive state-transition models (cf §1.2). It is significant that the development of the original model of the vehicle dynamics predated the visualisation by several months and that these programming activities were entirely independent to the extent that :

they were carried out without direct collaboration by two different programmers,
the specifications interact only through the use of common variable names,
subject to knowing names, the order of specification development was irrelevant.

The following dictionary definition of 'agent', drawn from the Chambers 20th Century Dictionary, illustrates the extent to which our simulation encompasses significant abstractions associated with agents:

agent, n.
  a person or thing that acts or exerts power:
  any natural force acting on matter:
  one authorised or delegated to transact business for another ...

Each of these usages is represented in our specification: representative agents include the simulation designer / the speed transducer: the force of gravity as specified in the equations of motion: the agent that mediates between the simulation user and the model in direct manipulation of a button or slider.

Acknowledgments

The author is greatly indebted to many people who have made essential abstract and practical

contributions to the programme of work discussed in this paper. He is particularly grateful to Yun Pui Yung and Ian Bridge for their indispensable contribution in developing the simulation depicted in Figure 1 and to Yun Wai Yung, Mark Norris and Mike Slade for developing the underlying tools.

Conclusions

The research reviewed in this paper has laid the foundation for a new approach to programming that we believe - with proper attention to principled elaboration - can significantly address the problems of reconciling theory and practice. Our software experiments demonstrate that our methods have considerable potential for further development, some outside the scope of the applications we have so far addressed. Possible areas of future interest include:

automatic generation of procedural programs from definitive specifications,
the use of definitive specifications for generating portable software,
the implementation of definitive interpreters on parallel architectures,
the development of definitive notations for geometric modelling and CAD,
building environments for developing definitive programs,
abstract programming languages based on definitive languages (e.g. to support a definitive implementation of a definitive interpreter),
unification of programming paradigms in a definitive programming framework.

The fact that we shall need far greater programming capability to pursue all the avenues that are programme has opened up is a good measure of the success of our project.

References

1. J Backus Can programming be liberated from the Von Neumann Style? CACM 21(8), p613-641, 1978
2. G Booch Object-oriented development IEEE Trans SE 12(2), 1986, 285-292
3. F P Brooks, Jr No silver bullet: essence and accidents of software engineering Computer 20:4 (1987), 10-19
4. A M Davis Software Requirements Analysis and Specification Prentice-Hall, Englewood Cliffs, NJ,1990
5. M S Deutsch Focusing real-time systems analysis on user operations IEEE Software, Sept 1988, 39-50
6. S van Denneheuvel Constraint-solving on database systems: design and implementation of the rule language RL/1, CWI Amsterdam 1991
7. D Harel Biting the silver bullet: towards a brighter future for system development IEEE Computer, Jan 1992
8. W Kent Data and Reality North-Holland, 1978
9. D McDermott A critique of pure reason Comput Intell 3, 151-160, 1987
10. B C Smith Two lessons of logic Comput Intell 3, 151-160, 1987
11. B Wyvill An interactive graphics language PhD Thesis, Bradford University, 1975
12. M Chmilar, B Wyvill A software architecture for integrated modelling and animation New Advances in Computer Graphics, Proc of CGI'89, p257-276
13. W W Wadge, E A Ashcroft Lucid, the dataflow programming language, Academic Press, 1985