Paradigms for Programming

Meurig Beynon

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

ABSTRACT

This paper presents a personal perspective on some of the issues raised at a Workshop on Functional and Logic Programming Languages at the University of Stirling on April 25th, 1986. It is particularly concerned with the question: Are purely declarative principles sufficient for general purpose computing applications? The consideration of notions of "computational" and "ambient" state in a programming system suggests possible limitations of purely declarative methods where interactive applications are concerned. An alternative framework for general purpose programming systems, based upon the author's work on "definitive notations", is briefly outlined.

Paradigms for programming

Prologue

"Try to grow straight, and life will bend you" - Chesterton

There are many parallels to be drawn between "programming principles and practices" and "moral codes and human behaviour". In both contexts, there is the same tension between pure ideals and adulterated actions, the same accumulation of bodies of dogma and superstition purporting to universal and exclusive truth, the same uncomfortable practical triumph of sinner over saint. Although it may be easy to recognise the inadequate ideological foundation for much commercial computing, its pragmatic methods have produced some impressive results. And amongst programming paradigms, such as procedural languages, functional and logic programming languages, constraint-based systems, object-oriented programming and spread-sheets, it is not those with the purest interpretations which have so far proved to be most effective.

The virtues of declarative approaches to programming have been preached over a number of years. For many, it is an article of faith that procedural elements are absent from the programming languages of heaven.* Here on earth, notwithstanding the ministries of many apostles of the declarative school, it would appear that the procedural heresy is far from overthrown. Even one of the earliest and most respected disciples, whom we may for the sake of anonymity call Peter, has been known to suggest that "all declarative programs compute factorial(n)" - an observation which finds some confirmation in scripture. Certainly it must be conceded that if procedural principles are of the devil, then most of contemporary programming practice is damnable, and those products of declarative ideals that are presently most useful are themselves tainted with sin.

In this brief epistle, speaking as one of uncertain faith, with modest acquaintance with the canon of scripture, and little experience of discipleship, I shall review some aspects of this controversy, and indicate some sources of doubt for the naive mind.

---

*There are of course those who say that there are no heavenly programming languages, and yet others that there is only one.

At a recent Workshop on Functional and Logic Programming Languages at the University of Stirling, the relative merits and points of similarity between functional programming and logic programming were vigorously discussed. A subsidiary theme was the unification of these alternative approaches, combining the advantages which each had to offer. Some of the contributions hinted at a hidden agenda: a concern that functional languages were not universally perceived as a suitable medium for general purpose computing, or (evil thought) perhaps were not. It was suggested, for instance, that there were practical problems in dealing with input/output when using a declarative formalism. At any rate, the functional programmer's fond hope for major software applications was as yet unfulfilled. It seemed that programming languages inspired by logic programming principles had found wider and more general use, prompting the question: can some of these principles be used to enhance the scope and utility of functional languages?

Naturally, not all the faithful shared these misgivings about the status of functional programming. The established status of Lisp was cited as a possible counterexample, but discounted on the grounds that practical Lisp systems are not purely declarative. There were also those who questioned whether the principles of applicative programming had been correctly applied, and others who attributed the relative success of so-called logic programming languages to the procedural elements they inadvertently embodied, or the financial endorsement they had received from the Japanese. The suggestion that purely declarative notations might have intrinsic limitations was strongly opposed.

To put the discussion in context, it is important to consider the criteria by which adequacy of a programming notation is to be judged. As Alan Robinson rightly pointed out, by a naive criterion of expressive power, functional and logic programming, and FORTRAN for that matter, are "equivalent", and the decision against programming with a minimal basis for describing Turing computations is a matter of taste and convenience. A better framework for comparison of functional programs and logic programs is required. Human psychology is surely involved at this point, but there may be some principles at work which can be identified without recourse to experiment.

At Stirling, the merits of logic programming and functional programming were informally set

out by David Warren and John Hughes respectively (c.f. [D1]). The principal advantages of logic programming seem to be its suitability for handling partial and incremental specifications, its generality ("relations are more general than functions"), its flexibility with respect to interpretation of values as inputs and outputs, and its connections with non-determinism. Those of functional programming seem to lie in the use of higher-order functions, and the relative efficiency and quality of its implementations. In effect, such limitations as are imposed by working with functional specifications are redeemed by higher-order features for which (in the absence of suitable unification algorithms) there is no analogue in logic programming, and there is a compensatory gain in ease of implementation. It can also be argued that a functional programming language augmented by a set construct facility (such as HOPE-with-unification [D2]) combines the advantages of both approaches. Although, as declarative formalisms, both logic programming and functional programming can claim clear semantics, functional programming implementations reflect this quality more faithfully than practical logic programming implementations incorporating non-logical features.

The position to which the discussion tended to converge was that there was a perceived need to eliminate procedural elements from logic programming systems, and that pure functional and pure logic programming systems were complementary declarative formalisms, most desirable on account of their clear semantics, each offering characteristic advantages in use. Advocates of the functional approach argue that the task of eliminating impure features of logic programming systems is likely to be difficult, that higher-order functions offer additional power, and that functional programming with appropriate set constructs captures all that pure logic programming can offer. On the other hand, advocates of logic programming plead that satisfactory implementations will be developed in time, and that programming with relations is more general than programming with functions. For the most part, neither seems to question whether purely declarative notations represent a general ideal, but only which flavour of notation is the more appropriate.

The arguments for purely declarative notations have been well-rehearsed over many years. There is clearly merit in referential transparency when reasoning about programs, and in the

modularity which the use of higher-order functions allows. There is no question that defining and evaluating functions, or specifying and solving a system of equations, is a good paradigm for programming in many contexts. The crucial idea is the description of an algorithm without reference to "computational states". It will be helpful to examine the concept of the state of a computation critically at this point, since in this way possible limitations of a purely declarative approach are manifest.

Traditionally, a state of a computation has been interpreted as one of the machine states through which a computer passes in carrying out that computation. Procedural programming is perceived as spelling out the changes of state associated with a computation, in general with respect to an abstract computer in which the state is represented by the current values of a set of variables, possibly of complex types, and each step of the computation assigns a new value to one of these variables. In a functional programming system, the generic statements (namely function definitions and evaluations) are to be viewed purely as mathematical specifications of functions and values, and if no presumption is made about the nature of the evaluation process, there is no place for the concept of computational state. The fact that "incremental specification" of relations or functions is possible within a pure logic programming framework indicates that there is a more comprehensive notion of state which may be important even in a purely declarative programming system. This concept of "total state" reflects both the computational state (as appropriate) and the "ambient" states of the programming system, that is, states of the system in which no computation is in progress. For a declarative programming system, the ambient state is determined by what definitions of functions, or equations, have taken place in the course of a user dialogue. This is closely related to the state of the system as perceived by the user through the history of interactions, in that definitions of functions and equations affect the state of the programming system (that is, they enable certain evaluations and solutions), and function evaluation and equation solution are without side-effects upon this state. In general terms, we can think of computational states as associated with partially *evaluated* functions or partially *solved* equations, and the ambient state as associated with partially *defined* functions or systems of equations. From this perspective, specifying an equation constraining the variable x to satisfy a predicate P(x) alters the ambient state,

whilst a strategy for subsequently evaluating x (such as a backtracking search) invokes a concept of computational state.

For a procedural programming system, the ambient state is determined by what variables have been declared, what procedures have been defined, and the current values of variables. The computational state, as defined by suspension of a procedure execution, includes state information of a very similar form, but has semantically complex ingredients, involving detailed knowledge of how local variables within procedures are bound and assigned values.

For a functional programming system, the ambient state is determined by what functions have been defined, and the computational state by the function evaluations in progress. To interpret computational state in this context, we must have some evaluation strategy, such as lazy evaluation, in mind. Even then, our notion of total state is by no means as explicit as that associated with a procedural programming system.

For a logic programming system, the ambient state is determined by what logical variables have been defined and what equational constraints have been put upon them by predicates. In this state, there will in general be logical variables which have been constrained to explicit values, and others which are not fully ground. The computational state depends as before upon our choice of evaluation strategy; typically, as in the suspension of a search procedure, it will be defined by constraints on logical variables resembling those in the ambient state (and consistent with them), together with semantic information implicit in the search strategy.

From this discussion, it would appear that the suitability of a particular kind of programming system for a specified task can to some extent be evaluated by considering what kind of interaction with the user is required.

In certain respects, purely declarative principles are especially well suited for programming in which user interaction plays no part. In an single interaction, the user gives a formal specification of the programming task, and an appropriate computation is constructed automatically. Neither ambient nor computational states have a significant role here.

The purity of such a programming paradigm is very seductive. Mathematical reasoning about a static description of a computational problem is far easier than reasoning about an explicit

computational process. The potential difficulties are by now fairly well identified. Controlling the complexity of the algorithm which results from translating a formal specification is one important concern, and whilst there may be scope for exploiting parallelism, there are many technical problems to be solved. The sceptic can argue that the absence of a well-defined computational state is a practical obstacle to defining efficient algorithms, and that an evaluation strategy which provides an interpretation of computational state may also introduce sequentiality. Much contemporary research is focussed in this area, and the future development of program transformation techniques and alternative architectures is sure to supply some answers.

The philosophy behind the declarative approach to programming puts considerable emphasis upon correct initial specification of the programming task, and does not provide for subsequent revision of an approximate specification. To some extent, this presumes that the user knows in advance all that is required of the computation. There are some subtle issues here, and it is not wholly clear what this precondition entails. It must be acceptable for a program to require data which is to be specified at a later stage, for example. Interesting examples arise naturally in logic programming, where (in what might be called "lazy specification") an incomplete specification is refined incrementally. In other tasks, such as editing a text or a picture, interaction and feedback have a more obvious role.

Interaction naturally invokes ideas which are most conveniently viewed procedurally, in that significant interaction must potentially affect a "change of state". There is a proper suspicion of using interaction needlessly to accomplish a task for which we have sufficient prior knowledge, when a declarative approach is most appropriate. (In this context, the experimental approach to programming commonly practised by unscrupulous programmers might be viewed as a degenerate form of "procedural programming" at a very high level.) On the other hand, there are circumstances in which interaction is entirely appropriate, as in the systematic revision of a written text.

The above arguments suggest that purely declarative notations are most appropriate where there is little call for interaction, either because complete information is available initially, or because partial information can be subsequently supplemented with additional information in a consistent fashion. In fact, interactive systems so far represent a problematical area for applications

where functional languages are concerned (c.f. [A1] p.290). At the Stirling Workshop, for instance, Peter Henderson referred explicitly to the difficulties encountered in writing functional specifications for processes in which complicated interaction with the user is involved, and the apparent limitations of modelling input/output via streams. There are two issues here: the suitability of a purely declarative notation for dialogue, and the suitability of functional methods for implementing systems for dialogue. With regard to the first, it appears likely that declarative notations offer no satisfactory means for representing the current state of a dialogue, a contention argued in more detail below (see also [B1]). With regard to the second, it may be argued that the implementation of a system for dialogue must itself incorporate a satisfactory representation of the current state of a dialogue. Abstract as these arguments are, they are advanced as plausible reasons for the relative lack of success of functional methods in interactive applications. Indeed, from the description of the ambient and computational states in a functional system given above, it appears that such a system is appropriate in interactive use only if the state of the dialogue is conveniently representable by the suspended evaluation of a function under some evaluation strategy.

The traditional response to the problems of adapting functional programming systems for interaction has been to insist that static description is of paramount importance, since no alternative approach offers as clear semantics or as firm a basis for mathematical reasoning. This has led to the introduction of streams for modelling time-dependent processes, and the use of a static "history over time" as a means of representing a dynamically changing "state". The development of LUCID (c.f. [A2]) is a particularly interesting example of this approach, and one which illustrates a perceived need for a good operational as well as denotational semantics.

As was discussed at Stirling, concurrency creates special problems for stream models of data (c.f. [A1] p.291). It was interesting to hear research workers who are actively investigating these problems at Stirling proposing the use of CCS or CSP as a metalanguage sitting on top of functional implementations; an indication that, at least for the present, pure functional models are not coping adequately with concurrency. In this context, the need for interaction between concurrent processes is also suggestive.

It may be that the case for stream models has been overstated; it is debatable whether

reasoning with streams is necessarily easier than reasoning with time-dependent processes. To take an extreme example, it is surely easier to reason about a well-written FORTRAN program than about the streams of values generated at the relevant computer addresses during execution. The importance of choosing a conceptually appropriate model must be taken into account. On these grounds, it is hard to accept that an editing process is appropriately represented by the history of edit commands, when provision for forgetting and overwriting is so deeply embedded in the conceptual process. As Steve Matthews suggested at Stirling, experience with LUCID indicates that a close relationship between the operational and the denotational view is the most significant concern, and the formalism which is most appropriate for mathematical reasoning is not necessarily the best medium for programming.

Whilst research into general applications for functional systems is well-established, pure logic programming systems are still at an earlier stage of development. Some of the concerns about the utility of purely declarative principles when modelling interaction expressed above apply equally to functional and logic based systems. An interesting, and perhaps significant difference, is that the computational states in a logic programming system may provide a better model for the state of a dialogue. Indeed, there is a strong resemblance between a set of logical variables subject to a set of constraints, and the computational framework which underlies one of the most practically successful paradigms for interaction - the spreadsheet.

On the evidence cited above, there seems to be a case against a purely declarative approach to interactive applications. Even if it should ultimately prove to be effective, it would arguably be by accident rather than design. After all, there is no obvious reason why partial definition or evaluation of a function, or partial specification or solution of a system of equations, should be a good method of describing the state of a dialogue. In a previous paper [B1], I have explained why I believe the principles used in spreadsheets offer a sounder basis for dialogue, and will give a brief summary at this point.

The essence of a naive spreadsheet is a set of variables, typically representing values of type integer, together with definitions which serve to specify the values of some variables in terms of others in such a way that circularity is avoided. Thus the value of the variable x may be defined by

the formula y+z, so that an evaluation of x returns the result of evaluating y and z and adding the results. A generic system of this kind, which I have called a "definitive notation", is based on an appropriate algebra of data types and operators relating data types, and comprises a set of variables of various types whose values are to be specified in terms of other variables using formulae over the underlying algebra.

The technical details (see [B1] and [B2]) are unimportant here. The significant aspect of a definitive notation is that it allows the state of a dialogue to be modelled in a way which is conceptually simple but which has advantages over alternative approaches. Within a definitive system, there is no useful concept of computational state, but only an ambient state in which certain variables have been given explicit values, and other variables have an implicit value (not necessarily well-defined) specified by an associated defining formula. This framework provides a model for the state of a dialogue which appears to have pleasing psychological characteristics. For instance (c.f. [B1]), it is possible to describe "persistent relationships" between variables (such as: file O is the result of compiling file S, or: the position of the big hand in relation to the digits on the clock face is determined by the absolute position of the small hand) and "transient values" (such as: the current contents of the file S, or: the position of the small hand). What is more, the ambient state of a definitive system is explicit, in that the current definitions of variables are easily inspected, and it requires no additional semantic knowledge to interpret the state. As explained in more detail in [B1], both declarative and procedural principles have a role in such a system.

I have argued that, in a general purpose programming system, an effective method of handling dialogue is needed. This might take the form of an inherent definitive notation, but there are other possibilities. These include an equational rather than functional approach to specifying constraints between variables, as illustrated in more sophisticated spreadsheets.

The discussion of a particular application will help to clarify some of the issues. Several different paradigms have been proposed for the description and manipulation of planar diagrams. These include commercial packages (such as the DOGS drafting system [P1]) predominantly based upon procedural principles, constraint-based systems (such as Sketchpad [S1]), and experimental systems using a functional language (such as Peter Henderson's functional geometry system [H2] -

c.f. [H1] p255) or a definitive notation (ARCA [B2]).* Both constraint-based and functional approaches offer good abstractions for diagrams, such as an equation for a locus, or a recursive specification of a picture. If elegant specification of an image is the principal objective, and no subsequent modification or manipulation is required, a functional description may be ideal. In general, more interaction is appropriate, and a constraint-based system allows the user to experiment with parameters within the context of specified equational constraints. There are nevertheless difficulties. The ambient states defined by a system of equations, and values of constrained variables, but it is not always possible to interpret such a system (solubility may be undecidable, for instance), and the effect of introducing an additional equation is to narrow rather than simply modify the specification.

In this application, the major difficulty is to allow the user sufficient flexibility for changing specifications whilst preserving an abstract representation. When a definitive notation is used, the constraints between variables have to be functional rather than equational. One consequence is that the user may have to choose a representation for a diagram lacking a symmetry which can be naturally expressed equationally. In effect, representations of diagrams take the form of explicit parameterisations which are more laborious to specify but easier to monitor and modify. Perhaps there is scope for notations for interaction in which definitive principles are augmented by constraint-based ideas. Unification theory, which deals with the relationship between equational and parametric specifications, may be helpful here.

In this paper, I have questioned whether purely declarative programming systems are suitable for general purpose computing. I am led to this position not from a lack of appreciation of the value of purely declarative principles, but through consideration of apparent limitations in respect of interactive applications. If my criticism is well-founded, the need for some procedural elements must be admitted, and the onus is then upon the user to make prudent use of assignment in the interest of clear semantics. It seems that our conceptual models of our environment incorporate features reflecting several different programming paradigms, and it may be that a general purpose

*There are significant differences between the types of diagram considered in each case, but for the purpose of this discussion an abstract view is adopted.

programming system requires similar characteristics. One approach to synthesising functional, procedural and logic programming concepts within a single framework is to develop a sophisticated definitive notation which allows the user to augment the underlying algebra by introducing additional data-types and operators (using functional programming principles), and to specify axioms for the underlying algebra or equational constraints between variables (using logic programming / constraint-based principles). There are many technical obstacles to the successful development of such a system, some of which might be resolved by a judicious choice of underlying algebra. I have discussed the potential for special-purpose systems exploiting these ideas in previous papers ([B1], [B2]), but believe there may also be scope for a general purpose system. Indeed, if the underlying algebra comprised lists, and operations on lists, the resulting system would resemble existing Lisp systems in many respects.

Epilogue

The fundamental issue addressed in this paper is whether procedural elements are necessary in an effective general purpose programming system. It is an issue over which it is difficult to compromise, in that introducing the simplest form of assignment in principle opens the door to procedural practices in all their gory glory. Even so, it may be that there are programming systems incorporating procedural elements which offer appropriate alternatives to these excesses. Perhaps it is also too much to expect more: that we cannot have absolute purity and adequate power.

Acknowledgments

I am grateful to Peter Henderson for allowing me to invite myself to the Workshop at Stirling, and to Steve Matthews for some stimulating discussions.

References

[A1] Abelson, H. and Sussman, G.J., *Structure and Interpretation of Computer Programs*, The MIT Press, 1985

[A2] Ashcroft, E and Wadge, W.W., *LUCID, the Dataflow Language*, Academic Press, 1985

[B1] Beynon, W.M., *Definitive notations for interaction*, Proc. hci'85: "People and Computers: Designing the Interface", CUP, 1985

[B2] Beynon, W.M., *ARCA as an archetypal definitive notation*, RR#**, Dept of Computer Science, University of Warwick, June 1986

[D1] Darlington, J., *Unification of Functional and Logic Languages*, Department of Computing, Imperial College, 1983

[D2] Darlington, J., Field, A.J., Pull, H., *The Unification of Functional and Logic Languages*, Department of Computing, Imperial College, 1985

[H1] Henderson, P., *Functional Programming: Application and Implementation*, Englewood Cliffs, N.J., Prentice-Hall, 1980

[H2] Henderson, P., *Functional Geometry*, Proc. 1982 ACM Symp. Lisp and Functional Prog., ACM, NY 10036, p179-187

[P1] PAFEC Ltd., *DOGS user manual*, PAFEC Ltd., Strelley Hall, Strelley, Nottingham, UK

[S1] Sutherland, I.E., *SKETCHPAD: A Man-Machine Graphical Communication System*, TR #296, Lincoln Laboratory, MIT, 1963

Appendix

The Workshop on Functional and Logic Programming at Stirling was introduced by Alan Robinson, who gave an overview of the principal issues for consideration. I include a few cryptic notes based on this introduction, partly because they have been an important source of ideas in compiling this report, but chiefly because they are of interest in their own right.

Features

first-order vs higher-order logic

predicate vs lambda calculus

typed vs untyped languages

relative vs absolute set-constructs

denotational vs reduction semantics

metalinguistic closure vs separate metalanguages

Computational themes

non-determinacy : the use of logical variables

completeness : occurs check, search cutoff, etc.

non-logical features : cut, clause order, etc.

complexity : computing system / theorem prover?, Kanellaki's theorem

extended unification : Colmerauer's PROLOG III, Houet's theorem

lazy working : infinite objects, streams, object-oriented style, state

Implementation

parallel architectures for declarative systems

backtracking : want / need / hide ?