Definitive Principles for Interactive Graphics

Meurig Beynon
Dept of Computer Science
University of Warwick

Abstract

A definitive (ie definition-based) programming paradigm for interactive graphics is investigated. Two contrasting examples of graphics notations for line-drawing applications based upon definitive principles are described. The advantages of using a definitive notation for interactive design, and as a medium for representing several kinds of abstraction are considered. Some relationships between definitive programming and other paradigms for graphical applications are identified, with a view to developing abstract models and formal methods for computer aided design.

Keywords: graphics languages, user-interface design, human-computer interaction, data types and data structuring, formal methods, computer aided design

Introduction

Many different paradigms for computer graphics systems have been proposed and developed. Amongst these are the procedural, the constraint-based, the functional and, most recently, the object-oriented approaches. This paper examines yet another paradigm: the use of definitive (ie definition-based) [B1] notations for computer graphics and computer aided design.

Each programming paradigm for a graphics language has its characteristic emphasis. In a definitive programming notation, the emphasis is upon interaction [B1]. The general principle is to regard "drawing a diagram" as a design process in which the intended semantics of the diagram has an important role, and to provide an appropriate framework within which the design dialogue can be conveniently represented. Essentially, a definitive notation consists of an underlying algebra of data types and operators (an abstract data type) together with a set of typed variables whose values are either given explicitly, or defined implicitly by means of formulae in terms of other variables. Each stage of the design process is then represented by a suitable set of variable definitions, which, in the context of a graphics notation, can be used to describe the abstract structure underlying a picture in much the same way that the definitions of fields in a spreadsheet impose abstract relationships upon scalar values. Perhaps the principal advantage of this approach is that partially completed images are recorded in such a way that the manner of construction is irrelevant, and that the user has considerable flexibility in choosing and modifying the underlying conceptual model.

Two contrasting definitive graphics notations have been developed by the author: DoNaLD - "a definitive notation for line drawings" [B3], and ARCA, for the display and manipulation of combinatorial diagrams [B2,B4]. This paper includes an overview of both notations (§2), with an emphasis upon illustrating the theoretical principles underlying their design.

DoNaLD has been designed as a popular graphics system illustrating definitive

principles. The underlying algebra is based upon **real**, **integer**, **point**, **line** and **shape** variables, where **point** and **line** values are geometric Cartesian or polar coordinates and line segments in the plane, and a **shape** value is a line drawing composed of a set of points and lines. Specifying an appropriate mode for defining the value of a variable of a complex data type presents a problem in a definitive notation [B1], in so far as (for instance) the value of a **shape** variable may either be specified directly in terms of other shapes, or by specifying its component points and lines independently. To overcome this problem, DoNaLD includes two kinds of **shape** variable: virtual **shape** variables whose value is defined by a single formula of type **shape**, and **open shape** variables, which comprise a set of **point**, **line** and **shape** variables which are used to define component points, lines and subshapes. There is a strong analogy between "**shape** and **open shape** variables" in DoNaLD, and "files and directories" in UNIX. This is reflected in the user-interface for DoNaLD, in which there is a window associated with each **open shape** variable. Editing the definitions within a window then resembles working within a subdirectory.

ARCA is a more sophisticated notation than DoNaLD, and may have more interest as an experiment in software design than as a practical programming medium. It is primarily aimed at the design and manipulation of a class of combinatorial diagrams with a rich and clearly defined mathematical semantics. The underlying algebra comprises **integer**, **vertex**, **colour** and **diagram** data types, which respectively represent scalars, Euclidean coordinates, abstract incidences, and realisation of coloured digraphs which semantically can be viewed as finite automata. In this context, the problem of defining variables of complex data types is resolved by introducing an auxiliary definitive notation so that the mode in which a variable is to be used for representing a value can be suitably declared. This has the advantage that, at the discretion of the user, the mode of a variable can either be specified precisely, enabling rigorous semantic checks on subsequent definition, or loosely specified so as to permit flexible use.

Although the ARCA system, as presently developed, deals primarily with geometric aspects of diagram manipulation, the richness of the associated semantic framework naturally suggests its consideration as a prototype computer aided design system. A subsidiary theme of the paper, to be developed in §3, is that definitive principles offer a promising approach to the problems of handling many different kinds of abstraction of central importance in computer aided design. Some discussion of connections with other programming paradigms, and directions for further research, is also included.

§1. Background

1.1 Paradigms for graphics languages

Many different paradigms have been proposed for graphics languages. Amongst these are the procedural, the constraint-based, the functional, and more recently the object-oriented approaches. This paper investigates graphics notations based upon yet another paradigm: a definitive ("definition-based") approach which may be seen as a generalisation of the spreadsheet concept.

As is to be expected, the emphasis in some of the most successful and highly developed commercial packages (eg DOGS and BOXER [P1]) is upon a procedural approach. Such languages in effect provide a powerful virtual machine for graphics incorporating high-level primitive operations, such as the display or transformation

of a complex shape. The limitations of describing graphical images using such tools resemble those of describing an algorithm in a sophisticated high-level procedural language: it is hard to infer abstract relationships between components of an image from a recipe for its construction.

Historically, constraint-based graphics offers perhaps the most significant alternative to a procedural approach [N1]. Since the pioneering work of Sketchpad [S2], several experimental systems for graphics have used this paradigm. A constraint-based system deals very directly and elegantly with the problem of specifying functional relationships between geometric elements, but does not have the same flexibility as a procedural system if extended interaction is needed to create large and complicated images. It may be necessary to construct a figure which can be conveniently specified approximately by a single constraint, but to modify it in some point of detail for instance. In other situations, it may be necessary to invoke a set of constraints whose consistency is difficult to determine, or to revoke constraints in a controlled manner. Dealing with these problems can easily lead to a situation in which detailed knowledge of how an image is constructed is required to manipulate it.

A possible solution is to use a purely declarative notation: an idea explored by Henderson [H1], and developed by others with applications such as VLSI in mind (eg Sheeran [S1]). Such an approach is attractive when the objective is to display an image which can conveniently be specified statically in its entirety, and gives particularly impressive results for images which can be defined using recusrion or higher-order functional abstractions. In the functional framework, as with constraint-based systems, it is nevertheless hard to see how to describe iterative design of an image in a satisfactory way.

Object-oriented programming techniques for graphics [G1], though ostensibly procedurally based, have some of the advantages of each of the above methods. It is relatively easy to ensure that functional relationships between geometric entities are preserved by suitable protocols between objects for example, and to make incremental changes by modifying the behaviour of a single object. Such techniques also offer scope for associating semantic information with components in a natural way. The main problem in this context is the lack of a suitable semantic model in which to interpret the representation of a graphical image within an object-oriented system.

The features required of a graphics language are influenced by the intended application. It is particularly important to distinguish between contexts in which the intention is merely to construct pictorial images, and those where additional manipulation and interpretation of graphical images is needed. In the former case, a sophisticated paintbox facility may be most appropriate; a set of operations which can be freely used to paint a picture as conveniently and efficiently as possible. In the latter, a means of representing an image in a more abstract fashion must typically be devised. From an implementation perspective, this may be seen as the distinction between either defining a pattern of pixels directly, or building up an abstract data structure which encodes semantic information about an image, and in particular can be used to construct a graphical representation. This distinction is often reflected in the chosen user interface, which may involve explicit referencing of displayed elements and / or more oblique methods based upon textual input.

The historical development of systems for graphics reflects a general trend towards methods in which the emphasis is upon building appropriate underlying data structures to represent pictures, rather than simply upon drawing pictures directly.

There are a variety of reasons for this. A primary reason is that in many of the most important applications of graphics (eg circuit design, mechanical engineering drawing, architectural draughting) the graphical images have an associated semantics which cannot reasonably be ignored. It has become increasingly evident that means of binding conceptual structure to images is essential if effective manipulation and interpretation is to be possible in such contexts. Even in those applications where pictorial images might be seen as the principal objective, as in computer animation, or the layout of illustrations in a paper, it may still be preferable to work with an abstract representation. This reflects the need in general for interactive modification of images, either whilst iterating towards a suitable image, or on subsequent manipulation. Such interaction usually entails a mechanism for referencing an image in a semantically sensible fashion, and also requires a framework within which the stages of an iterative design process can be conveniently represented. It is with applications of this nature that this paper is primarily concerned.

1.2 The definitive programming paradigm

Procedural graphics languages are based upon sophisticated sets of operations which can be used to construct images efficiently. Functional graphics languages stress ways of specifying the abstract structure of images effectively. Constraint-based languages give scope for the representation of abstract structure, and at the same time permit some degree of flexibility in choice of parameters. The object-oriented approach, if used intelligently, provides a framework within which some of the advantages of all these paradigms can be gained, but does not have an entirely satisfactory semantics.

The primary emphasis in using a definitive programming paradigm is upon interaction [B1]. The general principle is that of viewing drawing as a design process, and capturing the stages of the design in a way which is transparent to the user, and conveniently represented in a computer, viz by storing a set of appropriate variable definitions. Perhaps the most significant advantage of this approach is that the details of a partially completed image are abstractly recorded so that the manner in which it has been constructed is irrelevant, and that the user has considerable flexibility in choosing and modifying the underlying conceptual model. Such an approach allows the user considerable scope for introducing appropriate means of referencing components of an image, for instance, and makes subtle forms of backtracking possible without complicating the semantics.

The principles behind definitive notations for interaction are set out at length in [B1], and will merely be summarised here. In essence, a definitive notation consists of an underlying algebra of data types and operators (an "abstract data type") together with a set of typed variables whose values are either specified explicitly, or defined implictly by means of formulae in terms of other variables. A typical action in a definitive notation dialogue is then the declaration, definition or evaluation of a variable. Simple as this programming paradigm may seem, experience suggests that it can be used as a framework within which to develop a rich variety of notations. Possible extensions and refinements to be explored (some of which are illustrated in connection with the notations ARCA and DoNaLD below) include: the introduction of user-defined data types and operators, special modes for associating values with variables for the purposes of definition and evaluation, concurrent use of two or more definitive notations in counterpoint, and context switching through changing the underlying algebra. It may be noted in particular that a functional language is designed to address just one of these issues, viz the specification of user-defined data

types and operators, so that, in some variants, definitive programming is more general than functional programming.

A spreadsheet - stripped of its tabular user interface - provides the simplest example of a definitive notation, in which the underlying algebra is traditional arithmetic. It may be seen that the formulae used to define fields in a simple spreadsheet provide a conceptual model which underlies the scalar information explicitly displayed, and allow the user to manipulate this information in a manner which reflects an abstract view. The theme of this paper is that the same programming paradigm can be used effectively when specifying geometric information in an abstract fashion. Indeed, it will be argued - in this context with specific reference to graphics applications - that definitive notations are a powerful means of describing many different kinds of abstraction. Practical evidence to support this view is based on the design of two contrasting definitive notations for graphics: the ARCA notation, intended for the display and manipulation of combinatorial diagrams, and DoNaLD, a definitive notation for describing line drawings. The principal features of these notations are described in §2 below. For more details, the interested reader should refer to [B2,B3,B4].

§2. Two examples of definitive notations for graphics

2.1 DoNaLD: a definitive notation for line drawings

The DoNaLD notation is designed to demonstrate the use of definitive principles in a simple graphical application. The purpose of the notation is to describe and manipulate line drawings composed of points and lines in the plane. As will be illustrated in 2.2 below, it is not necessarily appropriate in all applications to conceive points and lines as concrete geometric entities - a more abstract view is desirable in some contexts - but in DoNaLD the underlying algebra is based upon data types for representing planar drawings in the most direct and unsophisticated fashion. In all there are 5 data types in this algebra:

real, **integer**, **point** (= **real** × **real**), **line** (= **point** × **point**),
shape (= **set of points** × **set of lines**),

which are respectively used to represent real and integer scalar information, geometric points and line segments in the Euclidean plane, and line drawings composed of a multiset of points and a multiset of lines. The operators of the algebra include standard arithmetic operations on scalars, projection operators to select component coordinates from points, and endpoints from line segments, and simple geometric operators for specifying the line segment joining two endpoints, the point of intersection of two line segments, or the result of interchanging the endpoints of a line segment. There are also operators which act upon shapes, combining two shapes into a single shape, or rotating one shape into another.

The use of **real**, **integer**, **point** and **line** variables in DoNaLD is very straightforward. The type of each variable is declared before use, and the value is subsequently defined by a formula - possibly an explicit value - of the corresponding type. As explained in [B1], few semantic rules govern variable definitions: it is only necessary to ensure that no circular definition is introduced. It may also be noted that undefined values can be handled very easily within a definitive notation, using a simple lazy evaluation strategy.

Variables representing complex data types within a definitive notation present some syntactic and semantic difficulties. To an extent, these are evaded by the restriction to *planar* line drawings. Were it possible to define points in dimension > 2 in

DoNaLD, it might have been necessary to provide a mechanism for specifying the components of **point** variables independently. As discussed at length in [B1], this would probably have necessitated the introduction of two kinds of **point** variable: *abstract* variables, whose values are specified by a single algebraic expression returning a **point**, and *composite* variables to be viewed as arrays of variables of type **real**. Since a DoNaLD **point** has only two components, the exclusive use of "abstract point variables" does not cause too much inconvenience.

The design and use of **shape** variables is more problematical. Recall that the value of a **shape** variable is a multiset of geometric points and lines, so that the need for some mode of incremental definition cannot reasonably be ignored. A naive approach is to introduce abstract and composite **shape** variables, where an abstract **shape** variable has its value defined by a **shape** expression, and a composite **shape** variable by giving appropriate definitions for its component points and lines. One major problem arises: the dichotomy between abstract and composite **shape** variables is crude, and proves too clumsy in practice. It may be convenient to specify a line drawing by describing some component points and lines explicitly, and the remainder by abstractly defined subshapes for instance. A simple and elegant solution for this problem is to replace the concept of a composite **shape** variable, which in effect comprises a family of constituent **point** and **line** variables, by a more general notion for which the term **open shape** variable has been devised. An **open shape** variable resembles a composite **shape** variable in that it incorporates constituent **point** and **line** variables, but may also include constituent **shape** variables (the "constituent subshapes") which may themselves be **open shape** variables. (A recursive definition of the data type **shape** as

$$\textbf{shape} = \textbf{set of points} \times \textbf{set of lines} \times \textbf{set of shapes}$$

might be viewed as the basis for the concept of an **open shape** variable.)

The introduction of **open shape** variables plays a major role in the design of DoNaLD. In the first place, it simplifies the problems of referencing component points and lines within a **shape** very considerably, by making it possible to organise related components of a line drawing into a single subshape. (By way of illustration, an **open shape** variable to represent the plan of a building might include the points and lines defining the overall configuration of rooms, together with a component **open shape** variable to represent the layout of each room in greater detail. Within each room there might be points and lines to represent significant locations, and further component **shape** and **open shape** variables to represent fittings and furniture. Typical definitions might specify the location of a light fitting with reference to the corners of the room, and describe the structure of several identical desks via **shape** variables defined either by a single **open shape** variable - or a user-defined nullary operator - representing an archetypal desk.) The referencing mechanism used syntactically resembles the use of directories in the UNIX file system: a **point** p within the **open shape** T which is a subshape of the globally declared **open shape** S is denoted by S/T/p relative to the global context. The semantics of **point** and **line** references is a little subtle: in effect the declaration of **point** and **line** variables within an **open shape** T introduces labels which can be used for the selection of the corresponding points and lines from the line drawing associated with T. To ensure that there are appropriate ways to refer to the points and lines of line drawings defined by general **shape** expressions (eg the expression defining abstract **shape** variables), it is necessary to establish conventions for determining labels for the components of such expressions, which may in some cases involve mechanisms for disambiguation. A full discussion of these issues, which is related to the question of why the data type **shape** represents line drawings rather than *labelled* line drawings, is beyond the scope of

this brief overview [B3].

In view of the analogy with directories of files mentioned above, it is natural to introduce some device corresponding to "changing directories". It is convenient to think of the global context as associated with a "universal open shape" Γ, so that the declarations of variables in the global context are interpreted as specifying components of Γ. The user interface for DoNaLD is based upon a hierarchy of windows, and rooted upon the global window in which the declarations and definitions of the globally declared points, lines and abstract subshapes appear together with the list of **open shape** variable names. There is one such "context window" associated with each **open shape** variable, which can be displayed after selecting the **open shape** variable name in the traditional fashion. A virtue of this interface is that variables can be referenced relative to their current context, thereby avoiding excessive use of variable references from the global context in much the same way that the use of directories can eliminate "absolute path names". Full details of this interface are left to the readers imagination, but it is worth noting that the scope rules in DoNaLD ensure that the definitions in each window reference variables lying within the immediately enclosing context, so that syntactic conventions such as the "../../../" in UNIX are not required.

The design of **shape** variables in DoNaLD offers an attractive solution to the problems of managing hierarchical abstraction of a kind which is very common in graphical applications. The list of **open shape** variable identifiers within a particular context window may be viewed as specifying the components of the associated line drawing which lie at the next level of abstraction. Such a facility for specifying different levels of abstraction can be exploited both in defining and displaying a diagram. (In this context, a diagram display is interpreted as a special kind of evaluation, so that several different modes of evaluation are seen to exist.) It is particularly important to recognise that the set of definitions within a context window captures the current state of a drawing design dialogue in a very effective manner, and is not subject to the same problems of interpretation as an extract from a procedural program might be (cf [B3]). The order of the definitions within a window is not significant, for instance, and the list of **open shape** identifiers within the context window acts as a index to the ingredients of definitions which are not explicit at the "present" level of abstraction. The comprehensive nature of the semantic model supplied by the context windows can be exploited in the design of additional features for manipulating definitions. To avoid tedious or otherwise inconvenient multiple definitions, DoNaLD incorporates array variables of each of the basic types, and some simple methods for iterative definition. The use of these mechanisms is to be viewed very much as a means to an end: they are simply convenient ways to edit the sets of definitions which alone describe the current status of the line drawing under development.

DoNaLD includes an important additional feature which deserves a brief mention. In view of the very rudimentary set of operators in the underlying algebra, some provision has to be made for user-defined operators. Since the syntax of the definitive notation which is the essence of DoNaLD is not really appropriate for the specification of such operators, the manner in which user-defined operators are specified is not of great relevance in this context, and can be viewed as a separate concern. Ideally, a simple functional language over the basic data types might be used, but the solution proposed in [B3] is a simple procedural notation. The significant fact is that a user-defined operator defines a pure function, and can have no side effects.

2.2 ARCA: a definitive notation for describing combinatorial diagrams

Like DoNaLD, the ARCA notation is used for describing and manipulating graphs. Unlike DoNaLD, the ARCA notation is designed with abstract graphical applications in mind, and this is reflected in the nature of the underlying algebra. The idea of viewing a graph as an abstract combinatorial object comprising vertices and edges satisfying particular incidence relations is central to the design of ARCA. This means for instance that concepts such as edge traversal and connectivity can be captured within the semantic model of an ARCA diagram. A vertex of a diagram in ARCA is associated with an index, rather than identified with its location in a particular realisation, and the edges of the diagram are specified via relations between vertex indices rather than by geometric line segments. Indeed, the edges have additional attributes, including a colour and a direction, so that an ARCA diagram can easily be interpreted as a finite automaton. A full discussion of the background to this semantic model, which is derived from a class of finite automata studied by Arthur Cayley in connection with group theoretic investigations in the 19th century, appears in [B2]. In this context, it will be enough to conceive an ARCA diagram as describing a deterministic finite automaton in such a way that a vertex represents a state, and an edge of a particular colour a state transition labelled by a particular input symbol. This particular semantic model is significant as a common interpretation for a graph, which underlies applications in which the edges of a graph are used to represent relations on the set of vertices (cf the transport networks in §3).

There are three primitive data types in the underlying algebra for ARCA. These are **integer**, **vertex** and **colour**, which are repectively used to represent scalar, coordinate and incidence information. All scalar information - including the coordinates of geometric points - is interpreted as discrete, and the versatility of the **integer** date type is enhanced by the introduction of an associated modulus for each **integer** variable. Integer values of modulus >1 have their traditional interpretation as residues, and are useful as indices when referencing cycles of edges within a diagram. The "ordinary" integers are viewed as "integers modulo 0", whilst the scalar components of coordinates are expressed in terms of special purpose geometric units which can be conveniently regarded as "integers of modulus 1". Coordinate information - represented by values of type **vertex** - comprises an array of integers of modulus 1, and may be of any dimension >1. In effect, the value of a variable of **vertex** type is a vector comprising integer components representing a "sufficiently accurate" approximation to a geometric point under an appropriate convention. The representation of incidence information is more subtle, and corresponds to "simultaneous specification of all the transitions associated with a particular input symbol in a deterministic finite automaton" rather than to the "independent specification of each transition". A variable of type **colour** accordingly has a value which is a partial permutation of an appropriate degree ("the number of states") and is primarily used in connection with a particular specification of vertex indices within an ARCA **diagram** (see below).

There is one complex data type - the **diagram** - which may be seen in some ways as analogous to the DoNaLD type **shape**. A value of type **diagram** is a realisation of a combinatorial graph: an indexed set of vertices with associated coordinates (possibly in dimension >2), and a family of edges partitioned by colour which define incidences between these vertices.

The underlying algebra for ARCA includes a rich variety of operators, and is far more sophisticated than that of DoNaLD. It includes all the standard operators relating scalars, vectors and permutations, and special purpose operators introduced to

assist the problems of specifying explicit values for permutations, vectors and vertex indices in **diagram** expressions. There are also operators to construct a join and product of diagrams, making it possible to address quite sophisticated semantic aspects of ARCA diagrams within the notation.

The problem of defining the values of variables of a complex data type alluded to in connection with **shape**s in DoNaLD occurs in an even more acute form in ARCA. The fact that both **vertex** and **colour** variables can have an arbitrary number of components precludes the exclusive use of abstract variables of these types, and the intricate form of the **diagram** data type poses additional difficulties. It may be convenient to define the value of an ARCA **diagram** variable by a **diagram** expression, or by specifying explicit lists of **colour** and **vertex** expressions to define component variables of a composite **diagram** variable, and it may be appropriate that these component variables themselves be abstract or composite variables. For instance, it can be useful to define the coordinates of a **diagram** by supplying an independent definition for each coordinate of the first vertex, and to define those of the other vertices in terms of the first vertex by abstract **vertex** expressions. The problems posed by the diversity of abstractions from a **diagram** value which may arise cannot reasonably be solved simply by devising a mechanism for typing variables once-and-for-all on declaration. Such an approach is too inflexible. Nor is the solution adopted in DoNaLD appropriate in this context, since it is not generally convenient to specify a finite automaton recursively. (There is however a connection between the solutions adopted in DoNaLD and ARCA, in that the use of **open shape** variables in DoNaLD is effectively an explicit mechaism for specifying the mode of a variable to represent a **shape** value.)

The key idea which leads to a satisfactory resolution of these difficulties is that of dynamic variable declaration: the introduction of a mechanism for declaring variables interactively. As might be expected, this takes the form of an auxiliary definitive notation for specifying the mode of abstraction by which a variable is to represent a value. The underlying algebra for this definitive notation is essentially the same algebra as that used for specifying the values of ARCA variables viewed at a higher level of abstraction, so that operators are defined on "abstract value types" or variable *templates* rather than on explicit values. These templates can be viewed as precisely describing the levels of abstraction at which the components of a complex data value are specified.

The details of this moding mechanism can best be indicated with reference to an example. As mentioned above, there are many different ways in which a **diagram** variable can be used to represent a **diagram** value. The simplest uses an abstract **diagram** variable whose value is defined directly by means of a diagram expression. An appropriate mode declaration for such a variable takes the form:

mode D = **abst diag**

- a declaration which precludes the existence of component **vertex** and **colour** variables for D. For the latter to exist, D must be declared as a composite variable, and it then becomes possible to give independent definitions to the component vertices and colours of D. An appropriate declaration in this case might take the form:

mode D = 'abc'-**diag** 15

indicating that D has component colours a_D, b_D and c_D, and vertices D!1, ..., D!15. As suggested above, it might be that the component coordinates of the **vertex** variable D!1 were then to be defined by independent **integer** expressions, but that the vertex D!2 was to be defined by an single expression of type **vertex**. To provide for this, the declarations:

mode D!1 = **vert** 4 *and* **mode** D!2 = **abst vert**

might then be used. The elaboration of this concept of variable mode should not be difficult to infer from these examples, and has a number of interesting ramifications.

It may be observed that the mode definitions used above are explicit: that is, mode expressions such as

'abc'-**diag** 15

represent values in the algebra of templates. It may also be convenient to make the declaration of a variable mode implicit, declaring

mode D!1 = **mode** D!2

for instance, to indicate that the template for the component variable D!1 is to be that specified for D!2. Indeed, there is a rich underlying algebra of templates which includes images of all the operators in the underlying algebra of ARCA values, as determined by relations such as

vert 2 + **vert** 2 = **vert** 2 *and* **abst col** . **col** 3 = **abst col**

expressing the fact that the vector sum of two composite vertices of dimension 2 defines a composite vertex of dimension 2, and that the product of an abstractly defined permutation and an explicitly defined permutation must be regarded as abstractly defined. This algebra is essentially a homomorphic image of the underlying algebra of ARCA values, in that an ARCA expression defines a value conforming to a template ("the mode of the expression") built up by applying operators to the templates of its constituent parts. Other features of this algebra are lattice operations representing the "least common abstraction" and the "greatest common refinement" of templates which are compatible in so far as they can represent the same value, and relations such as

vert 2+ **vert** 3 = @ *and* **vert** + **int** = @

(where @ represents an undefined value) which embody semantic rules concerning type information.

The introduction of an auxiliary definitive notation has other implications. A potential semantic problem is that any value definition of a variable such as

v = [2,3]

can be conceived as implicitly defining information about the variable mode. The function of a declaration is of course to constrain the manner in which the value of a variable can be defined, so that there are necessarily semantic rules to relate the definitions of mode and value. In essence, a single semantic rule suffices: in a definition of the variable v by the expression E, v and E must have compatible templates, and the mode of v must be at least as abstract as that of E. (In the ARCA interpreter currently under development, the slack nature of this constraint between declaration and definition is being exploited to admit conventions for variable typing which can be "arbitrarily" weak and strong at the discretion of the user.)

The ARCA system illustrates ways of capturing abstraction within a definitive notation framework which are complementary to the hierarchical abstractions in DoNaLD. The original motivation behind its development was the creation of a suitable context within which to display and manipulate Cayley diagrams of groups: a class of finite state machines which effectively supply geometrical models for the multiplication tables of groups (cf [B2] for details of this connection). The richness of the underlying algebra, and the complexity of the **diagram** data type reflects the complicated semantics of a Cayley diagram, which can be viewed both as a geometric object, and as an encoding of a sophisticated algebraic object. Contexts in which the same object admits quite distinct abstract interpretations in this way abound in computer aided design, and the counterpoint between two or more such interpretations is

often of central importance. The manner in which the definitive programming paradigm can be used to capture such "orthogonal abstraction" will be considered in §3 below, and is well illustrated by the ARCA system.

In the context of this paper, it is inappropriate to attempt to explain fully the variety of issues relating to Cayley diagrams and their properties which can in principle be addressed in a consistent and integrated fashion within the framework of ARCA, but the example to be considered in §3 below, based upon a more familiar and concrete domain, may give some insight. (The reader with a sophisticated algebraic knowledge will appreciate that the semantic domain addressed by ARCA is especially - perhaps even artificially - well-suited for the application of definitive principles. For instance, within the ARCA system, it is trivial to abstract the group multiplication table from a Cayley diagram, to exhibit the multiplication of elements graphically, to trace out relations on the diagram, and to compute the orders of elements represented by vertices of the diagram. The significant feature of such a system is not "being able to carry out these operations" - which are computationally easy - but being able to perform these operations by direct reference to the graphical display via the underlying semantic model which may be seen as an analogue of the group-theorist's interpretation of the diagram.)

§3. Evaluating definitive principles for computer aided design

3.1 Definitive principles for abstraction

Experience gained from research on ARCA and DoNaLD suggests that definitive principles are particularly useful as a means of capturing forms of abstraction commonly encountered in computer aided design. It will be helpful at this point to identify these forms of abstraction, and to examine in general terms how definitive notations can aid their description. This exercise is instructive both in respect of the definitive programming paradigm, and of the design process.

Three pervasive kinds of abstraction can be identified: *hierarchical*, *generic* and *orthogonal*.
It is difficult to give formal definitions for these concepts, but the following informal characterisations are proposed:

hierarchical abstraction refers to the process of viewing a single complex object at different levels of detail, as in viewing an electronic device as a black box, or as a system of interconnected functional cells, or as VLSI circuit layout;

generic abstraction refers to the process of associating many related objects into a single class, as in identifying 1, 2 and 3 as "integers", and <1,2>, <1,2,3> and <2,3,4,5> as "lists";

orthogonal abstraction refers to the process of interpreting a single object, or a single class of objects, in several independent ways, as in viewing a label on a picture as a pattern of pixels, or as a string of letters, or as a meaningful name.

Hierarchical abstractions can be captured in a definitive notation in a number of ways. The form of the sequence of definitions used to specify a value might be used to indicate a hierarchy of different views for instance. Thus the sequence of definitions

```
profit=sales-costs
sales=carpetsales+curtainsales
costs=tax+rates+expenses+wages
expenses=services+supplies
```

services=gas+electricity+telephone
.....

suggests many levels of detail at which the profit made by a shop could be specified. A single form of hierarchical abstraction is commonly used in connection with definitions of values within a particular class, as in the specification of a hierarchical data type such as the ARCA **diagram**, or the DoNaLD **shape**. Such abstractions are both hierarchical and generic, in the sense to be explained below. The recursive definition of the data type **shape** which underlies the **shape** variables in DoNaLD illustrates a technique which can be used to introduce hierarchical abstraction at arbitrarily many levels, as might be appropriate (for instance) in the definition of fractals or recursive list structures.

Generic abstractions are most directly represented within the framework of a definitive notation by the data types within the underlying algebra, which are associated with a class of explicit values. Definitions of variables in terms of undefined variables (which is essentially parametrisation of expressions) can be used to describe particular instances of generic abstraction. This is an especially important device in the design process, when it may be necessary to describe the form of parts of an object which cannot as yet be specified in greater detail. The use of modes in ARCA illustrates a method for extending the scope of generic abstractions based on variable type declaration; a fact which enhances the utility of variables whose values are undefined. With such a moding mechanism, it becomes possible to specify very precisely the form of parts of an object which are as yet undefined. Yet another kind of generic abstraction applies at a higher level of abstraction, and corresponds to the device - familiar to algebraists - of "changing the underlying algebra". The essential idea here is that the underlying algebra may itself be abstractly described in such a way that the operators and elements admit many consistent interpretations. As a simple illustration, a definitive notation based upon commutative ring addition and multiplication could serve as a calculator if the underlying algebra were the real numbers, as a context for performing residue arithmetic over a quotient of the integers, and as a medium for polynomial manipulation if the operators were viewed as symbolic. (The manner in which **open** shapes in DoNaLD can be used to define different contexts suggests a further application for the "change of algebra" concept: it might be that the nature of the underlying algebra could be context dependent, so that - for instance - the interpretations of **point**, **line** and **shape** - together with the associated incidence operators - could be chosen to describe spherical geometry within some subshape. As a simpler illustration, a context-dependent change of scale could also be construed as a change of algebra.)

Orthogonal abstraction is one of the most significant problems to be addressed in computer aided design. The idea that a single object admits several different interpretations is of central importance in the computer aided design process, and the difficulties presented by translation between semantic models are a common source of frustration. The solution adopted in ARCA supplies the archetype (even the ARCAtype) for the application of definitive principles in handling orthogonal abstraction. The key idea is that a single data type can be designed to synthesise several different semantic ingredients, in the same way that the ARCA **diagram** incorporates abstract group theoretic information encoded in the colour list, and geometric information encoded in the vertex list. In other words, an ARCA diagram represents both an "abstract finite automaton" and "a picture of the automaton" in the form of a graph whose nodes represent states and whose edges represent transitions.

It is interesting to note that this particular feature of the ARCA design has been

criticised for violating the principle of "separation of concerns". The appropriate answer to this criticism is that "juxtaposition of concerns" plays a very significant part in the design process, and what is important is that the specification of independent semantic information can be separated as necessary. It is quite possible to define and evaluate the coordinates of the vertices of an ARCA diagram without supplying any colour information, for instance; but the user has the means to integrate information about coordinates and incidences as required. As an indication of the subtlety of the interaction between semantic models of an ARCA diagram, it is possible (for example) to define the coordinates of the i-th vertex D!i of a diagram D in terms of the coordinates of D!(i+1) and those of the vertex D!j which is reached by following a specified sequence of coloured edges in D from the vertex D!i in a direction determined by the parity of i. In fairness, it must be added that the successful integration of independent semantic information illustrated by the ARCA **diagram** depends upon a suitable interrelationship between concerns. Where this is absent, the juxtaposition of semantic information within a data type effectively amounts to the parallel use of two disjoint definitive notations. As an illustration of this, it might be that in drawing a Cayley diagram solely for purposes of display the ARCA system could be used to produce a graphical image - perhaps a projection of an embedding in dimension > 2 - which could subsequently be reinterpreted as a DoNaLD line drawing and manipulated as a geometrical object. The virtue of this approach is that issues concerning incidences between geometrical edges - for instance - are more easily addressed by DoNaLD, in view of the abstract combinatorial nature of edges in ARCA.

As a footnote to this section, it is interesting to ask whether there is an alternative framework which provides a better formal characterisation of the kinds of abstraction which appear in computer aided design than the definitive programming model. It is arguably the case that neither a purely procedural nor a purely declarative programming paradigm offers the same scope for representing many varieties of abstraction. Indeed, the definitive framework for modelling abstractions can perhaps be useful both in an analytical and a prescriptive role. The example below suggests that the scope to choose and model abstractions of various kinds using definitive principles can assist the user in developing a definitive notation for a particular application.

3.2 An extended illustrative example

To clarify some of the issues considered above, it will be helpful to look abstractly at a possible graphical application, and to speculate upon ways in which definitive principles might be applied. To give a particular focus to the discussion, let us imagine that an ambitious graphics based system to provide comprehensive travel information is to be developed. This will make it easier to explain the long term objectives and motivating ideas behind the development of definitive principles than is possible solely with reference to the relatively modest graphical systems so far developed.

Central to the travel agency, there will be a graphical system for displaying geographical information in the form of an atlas comprising maps of various kinds. These maps will include an image of a 3-dimensional globe, various world maps in appropriate projections, maps of specific countries, of districts within a country, of holiday cities, towns and villages, and of localities within these holiday resorts. The relationship between these maps illustrates a very common form of hierarchical abstraction: ideally, we should like to link the complex of maps appropriately so as to reflect the fact that a map of a particular city lies within a district of a particular country, for instance. To represent such an atlas using a definitive notation, it would first

be necessary to devise an appropriate underlying algebra: that is, a set of data types and operators for describing each type of map via a suitable family of definitions. A naive solution to this problem would involve developing a data type appropriate for representing the map as a pictorial image, as in a traditional atlas, but this is only one aspect of the semantics of the map which is relevant in this context. In effect, there is a need for orthogonal abstraction, whereby places and transport connections are represented both by an image on the map, and by a combinatorial graph indicating when and how places are connected.

A possible solution borrows some of the features of DoNaLD and ARCA. Since an orthogonal abstraction for places and transport links is needed, the basic data type used to represent a *place* (which might be a country, district, city, village, or even a building) combines a conventional map, and semantic information about sites and transport connections between sites within that place. Since a hierarchical abstraction for viewing places is clearly essential, the general framework for the deign of the map data type is modelled upon the DoNaLD shape. A possible recursive specification for the underlying data types is then:

$$map = image \times list\_of\_places \times list\_of\_links$$
$$place = map \times reference \times index$$
$$link = map \times mode\_of\_transport \times index \times index$$

where each *place* in the *list_of_places* associated with a *map* M refers to a location on M with an *index* (or identifier) whose coordinates are specified relative to M by the *reference*, and each *link*
in the *list_of_links* refers to a link between places with specified indices supplied by a specified *mode_of_transport*. As an illustration, a *map* variable to represent Britain would comprise a pictorial image (*image*) together with a list of places of national interest and a list of principal transport connections between these sites. One such place would be Coventry, with its own associated *map* information (including its places of local interest, and transport connections) together with a map reference relative to the map of Britain, and a characteristic index. Amongst the transport connections in the *list_of_link*s for Britain might be the M1 motorway, from London to Leeds, and the main railway line from London to Glasgow. The purpose of the recursive specification for each transport link provided by the *map* of the link is to allow "sublinks" of a link to be represented. For instance, a *map* of the main London-Glasgow railway line would include Coventry (or perhaps Coventry railway station - a *place* within Coventry) as a *place*, and London to Coventry as a rail *link*. In a similar fashion, the *map* of a motorway would include links to represent the intervals between consecutive exits or service stations on the motorway. (Even these sections of motorway could have an associated *map* in which the places indicated emergency telephone points.)

The underlying data types above would be complemented by a set of operators designed to describe the physical relationships between places, and the physical connections between transport links. There would be operators resembling the join of DoNaLD shapes or of ARCA diagrams, for instance, which would combine the images of several maps, and the associated lists of places and links. Such operators could be used to construct maps for specific itineraries, or for the promotion of a particular holiday region. Operators to return physical characteristics of places, such as the average annual sunshine or rainfall, or the height above sea-level might be included. In the orthogonal semantic model associated with places and links, there would be operators to identify the endpoints of a link, and an operator to concatenate links, so that (for instance) a special bus service could be represented by a link variable defined by a concatenation of road links. In general, the principle within the model

would be that much of the information was encoded in predefined variables, but that the travel agent could modify or amend this information by redefining - or declaring and defining - variables. The London-Leeds link supplied by the M1 motorway might be predefined for instance as the concatenation of links associated with sections of the motorway between junctions along its entire length. Such a definition might need to be amended in the event of closure of a complete section of the motorway, by redefining a link between junctions to conform to the route for a diversion. Note that in principle this definition could take the form of a conditional expression, indicating for example that the diversion applied only on Sundays. It would also be natural for a travel agent to introduce variables to represent places of peculiar local or personal interest, such as hotels or related organisations with whom special arrangements had been made.

Within the framework sketched above, it is easy to identify many "orthogonal" ways in which information stored in the definitions of maps, places and links could be displayed. As in ARCA and DoNaLD, such displays are viewed as different modes of evaluation, and might correspond to partial evaluation (as in displaying the connections between stations on the London Underground), or exceptionally to evaluation over an alternative underlying algebra (as for instance in displaying flight paths between major airports as they would appear on the globe, rather than in a planar projection).

It remains to consider the other aspects of the travel agent's task: advising upon the scheduling and cost of journeys. A link in itself is an inadequate representation of a journey, since it describes spatial rather than spatial and temporal information. A possible data type to record a journey is a sequence of *legs*, where

$$leg = link \times transporter \times \text{dept\_time} \times \text{arr\_time}.$$

Functions returning the cost and duration of a *leg* would be available in the underlying algebra, and there would be an operator to combine two journeys into a single journey subject to feasible scheduling. In this way, a variable of type *journey* could be used to represent an itinerary for a particular client. As remarked above, the definitions of variables in this context could be framed so as to accommodate a degree of uncertainty: perhaps indicating alternative travel plans where appropriate, and incorporating parametrisation to reflect the dependence upon fuel surcharges, for instance.

The significance of the fanciful travel information agency sketched above lies not in the individual functions which it performs, but in the conceptual framework within which it has been described. All the facilities described are within the compass of a conventional system incorporating graphical, database and spreadsheet tools. The purpose of this illustrative example is to indicate how the use of definitive principles might lead to an integrated system for data definition and manipulation even in the context of a relatively unsophisticated application. In general, the benefits of adopting a definitive notation will be most evident when the underlying algebra has a richer structure than that associated with the travel information system, and the ARCA system - despite the apparent lack of a practical application - is probably more representative of the potential of the definitive programming paradigm.

Directions for further research

In this paper, the merits of the definitive programming paradigm for interaction and abstraction in computer graphics and computer aided design have been outlined. To put these ideas in perspective, it is helpful to look briefly at the connections with other programming paradigms, and indicate future directions for research.

The functional programming paradigm is essentially based upon the definition of an abstract data type (which can be viewed as an algebra of data types and operators), and the evaluation of associated algebraic expressions. A definitive notation which enables the user to specify new operators and data types in principle supplies a more general framework. In practice, the primary emphasis in the definitive notations so far developed has neither been upon enhancement of the underlying algebra, nor upon the introdution of higher-order operators into the underlying algebra, and to this extent represents a divergent direction of research. The case for using definitive principles rather than purely declarative principles in applications involving interaction or iterative design has been made at length in §3 and elsewhere [B1,B6]. It rests primarily on the need for appropriate means of representing both the state of the dialogue and the abstractions which are of central importance in such applications. It should be noted that a definitive notation for design applications for which a special purpose functional language has already been developed (eg the μFP language for VLSI design [S1]) could probably be derived by a process of enhancement in a conceptually simple manner. The issues are technical (solving the problems of variable specification and component reference for sophisticated data types) and philosophical (is it necessary, or desirable, to introduce procedural elements into a purely declarative notation?), and represent one direction for future research.

In graphical applications, there has always been a strong interest in techniques more closely related to an equational rather than a functional programming paradigm. Constraint-based systems represent an equational approach to geometry, and "intelligent" computer aided design systems make use of logic programming concepts. Naively: functional programming seeks to represent data by algebraic expressions typically involving higher-order operators, equational programming as the solution to a system of equations, and definitive programming by a system of interrelated definitions. At this stage, further research is needed to clarify the connections between these alternative representations, but some pertinent informal observations can be made.

Where geometric aspects are concerned, the relationship between the equational (ie constraint-based) and the definitive approaches can be illustrated to some extent by contrasting the specification of a parallelogram as a quadrilateral in which opposite sides are of equal length, and that given by a definition specifying the appropriate functional dependence of one vertex upon the other three. The former specification has the advantage of expressing a symmetric relation between the four vertices, so that the geometric relationship "ABCD is a parallelogram" is preserved under displacement of any vertex. In the latter case, the definition of the vertex D ensures that ABCD remains a parallelogram on displacement of the vertices A,B or C, but not on that of D itself. There is no doubt that the equational specification of constraints has aesthetic and practical advantages in this and other similar contexts, but the less elegant approach based upon definitive principles offers a better model in interactive use [B1,B6]. It is quite possible to specify a set of constraints which has no feasible solution for instance, and the constraint-based approach gives less scope for satisfactory representation of the state of the dialogue. A suggestive imprecise analogy between specification "by equations" and "by definitions" may be seen in the specification of an ellipse as "the set of solutions of a polynomial equation", or as "the locus traced by a point with parametrised coordinates", respectively. The definition of a geometric object within a definitive notation is typically a parametrised description: a possible generalisation of interest would entail the introduction of an operator to the underlying algebra which constructed a locus from an appropriate parametrisation.

A parallel relationship is observed where equational logic is concerned: the interdependences between variables specified by a set of predicates clearly resemble those which are imposed by a sequence of definitions. This tentatively suggests that definitive principles address in an explicit manner semantic issues which are implicit in a logic program. It may be that techniques such as unification, which in effect find a parametrised solution to a system of equations, supply a link between the logic programming and definitive programming paradigms. A more direct role for logic programming principles in connection with a definitive notation can be sought by introducing axioms into the underlying algebra, and exploiting symbolic manipulation techniques in the evaluation of formulae. More research is required in this area, but there is some evidence to suggest that the semantic models for abstraction and interaction supplied by definitive principles are a good basis for automated reasoning in an "intelligent" computer aided design system.

Amongst the programming paradigms for graphics, the object-oriented approach offers one of the most effective ways of dynamically maintaining constraint relationships, and integrating orthogonal semantic elements. In so far as it is easy to define variables x and y in an object-oriented programming system which are constrained so that x=y+1 irrespective of whether x or y changes its value, it is clear that such systems offer a more flexible model for handling constraints than a definitive notation. Perhaps the fact that the maintenance of the functional relationships

$$x=y+1 \quad \textit{and} \quad y=x-1$$

can be delegated to distinct objects - thereby avoiding the problems of self-reference implicit in considering the pair of definitions within the same context - is significant. Such considerations naturally point towards the development of a programming paradigm in which a network of independent processors interact via a definitive notation - an idea explored in the LSD notation for communicating systems [B5]. The investigation of this generalisation of definitive notations, and its relationship to the object-oriented programming paradigm, represents another promising direction for further research. It seems probable that the intrinsic semantic intricacies of the computer aided design process, and those of other graphically based concerns such as computer animation, require a programming paradigm of this degree of sublety. It is to be hoped that the perspective proposed in this paper can help to provide a coherent framework for its description.

Conclusions

The virtues of definitive principles for interaction and abstraction have been illustrated in connection with relatively simple graphical systems. Many of the techniques and principles described generalise directly to more sophisticated applications, but more research is required into connections with existing approaches before large integrated systems based upon a definitive programming paradigm can be developed. For instance, where computer aided design applications are concerned, it is important to relate the representation and manipulation of data within a definitive notation to the use of conventional data bases, and to consider the implications of introducing functional and logic programming concepts, such as higher-order functions, and automated reasoning, which have an important part to play in future developments.

Acknowledgements

I am indebted to many undergraduates for work which has contributed to the devel-

opment of the ARCA and DoNaLD systems over several years. I am particularly grateful to Nader Fahranak for helping to initiate the ideas on the design of ARCA which have been seminal in this research, and to Kevin Murray for programming assistance and stimulating discussions.

References

[B1] W.M.Beynon *"Definitive principles for interaction"* in Proc hci'85, CUP Sept 1985

[B2] W.M.Beynon *"ARCA: a notation for displaying and manipulating combinatorial diagrams"*, University of Warwick Computer Science Research Report #78, July 1986

[B3] W.M.Beynon, D.Angier, T.Bissell, S.Hunt *"DoNaLD: a line drawing system based on definitive principles"*, University of Warwick Computer Science Research Report #86, October 1986

[B4] W.M.Beynon, K.A.Murray *"The revised ARCA definition"*, University of Warwick Computer Science Research Report #**, <in preparation>

[B5] W.M.Beynon *"The LSD notation for communicating systems"*, University of Warwick Computer Science Research Report #87, November 1986

[B6] W.M.Beynon *"Paradigms for Programming"*, Alvey Software Engineering Mailshot, December 1986

[G1] A.Goldberg, G.Krasner, *"SMALLTALK-80: Creating a User Interface and Graphical Applications"*, Addison_Wesley, 1985.

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

[N1] G.Nelson, *"Juno, a Constraint-Based Graphics System"*, Proc. SIGGRAPH '85, Computer Graphics Vol 19(3), p235-243, 1985.

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

[S1] M.Sheeran, "μFP, a language for VLSI Design", Proc. ACM Symp. on Lisp and Functional Programming, pp104-112, 1984.

[S2] I.E.Sutherland, *"SKETCHPAD: A Man-Machine Graphical Communication System"*, TR #296, Lincoln Laboratory, MIT, 1963

Author's Biography

Meurig Beynon has been a Lecturer in Computer Science at the University of Warwick, UK since 1975. He has been an SRC Postdoctoral Research Fellow at University College, Swansea, and a Visiting Research Fellow at British Telecom Research Laboratories. He has published papers on many aspects of mathematics and computer science. His most recent research has been concerned with applications of a programming paradigm based upon definitions, and includes work on the design of languages for graphics, and for formal specification of communicating systems. He is also a member of a research team investigating VLSI design tools for regular array architectures.