DoNaLD: a line-drawing system based on definitive principles

Meurig Beynon
David Angier
Tim Bissell
Steve Hunt

Department of Computer Science
University of Warwick
COVENTRY CV4 7AL

ABSTRACT

The DoNaLD line-drawing system has been designed to illustrate how a generalised spreadsheet (or "definitive notation") can provide a simple medium for interactive graphics. Novel features of DoNaLD are: the use of variables based upon the recursive specification of a line-drawing as a union of points, lines and subdrawings, and a complementary user interface based upon windows of definitions.

A definitive notation for line drawing

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

ABSTRACT

The merits of "definitive" (definition-based) notations for interaction were first set out by the author in [1]. Further development of the ideas, and solutions to the technical problems in dealing with complex data types considered in [1], are discussed with particular reference to the design of DoNaLD: a Definitive Notation for Line Drawing. Features of DoNaLD are: the use of a recursive data type **shape** to represent line drawings, and an unusual user interface based upon a hierarchy of "context windows" for describing line drawings at different levels of abstraction.

Background

The notion of using definitive (definition-based) notations for interaction was first described in [1]. The essential principle in developing such a notation is to devise an "underlying algebra" of data types and operators which reflects the universe of discourse, and to introduce appropriate variables to represent values in the underlying algebra, either explicitly or implicitly through a defining formula. In such a system, a program - or **dialogue** - essentially consists of a sequence of variable definitions and evaluations. Perhaps the simplest example of such a notation is obtained by choosing the underlying algebra to be traditional arithmetic, when the dialogue resembles the use of a spreadsheet, stripped of its tabular interface.

As explained in [1], definitive notations appear to be very well-suited for dialogue. They make it possible to represent the state of a dialogue effectively, since the combination of implicit and explicit definitions allows both persistent relationships, and transient values, to be recorded. An important feature of a dialogue over a definitive notation is that all the information needed to determine the current state of the dialogue is automatically stored, and can be recovered by interrogating the variables to obtain their current definitions.

The focus in [1] was upon describing the advantages which the development of "generalised spreadsheets" might offer in interactive applications. The illustrations in the paper were based upon the definitive notation ARCA, for the display and manipulation of combinatorial graphs. In this paper, which is complementary to [1], some of the developments arising from subsequent research into definitive programming principles are illustrated with particular reference to another example of a definitive notation for interactive graphics - DoNaLD: a Definitive Notation for Line Drawings. The contrasting characteristics of ARCA and DoNaLD reflect quite different abstractions of geometric structure: for instance, the edge of a combinatorial graph is viewed in ARCA as "defined by a pair of vertex indices", and in DoNaLD as "the line segment between two geometric points". In effect, the DoNaLD notation gives a less abstract representation for a planar diagram, and is aimed at less specialised graphical applications than ARCA. Two other features of this work of especial interest are: the development, both in ARCA and DoNaLD, of more satisfactory solutions to the problems of dealing with complex data types in the underlying algebra (cf [1]), and the novel user-interface for DoNaLD, which is based upon sets of definitions within "context windows".

The author is much indebted to several students who graduated from the University of Warwick in 1986 for their assistance: David Angier, Tim Bissell and Steve Hunt, for contributions to the design of DoNaLD, and Kevin Murray for complementary work on ARCA.

DoNaLD: basic principles

The DoNaLD notation (a "Definitive Notation for Line Drawings") is intended for the interactive display and manipulation of planar diagrams comprising points and lines. As a definitive notation, it is based upon an underlying algebra comprising five data types: **integer**s, **real**s, **point**s, **line**s and **shape**s, and a variety of simple geometric operators. In contrast with ARCA, a definitive notation for describing combinatorial graphs used for illustrative purposes in [1], the data types in DoNaLD represent geometric data in a concrete fashion. Scalar values are represented by **integer** or **real** variables, points in the plane by **point** variables, directed line segments (that is, lines defined by an appropriate pair of endpoints) by **line** variables, and line drawings comprising a multiset of **point**s and **line**s, together with a set of **real** and **integer** attributes by **shape** variables. Following the usual pattern, a DoNaLD dialogue then consists of a sequence of declarations of variables, definitions of variables of the form:

$$\textit{variable} = \textit{formula}$$

specifications of user-defined operators, and evaluations of variables. For this purpose, realising the line drawing represented by a **shape** variable is viewed as a special kind of evaluation.

The full details of the operators in the underlying algebra appear in [2]. In brief, there are standard arithmetic operators, vector operators acting upon **point**s viewed as 2-dimensional vectors, and a variety of operators on points and lines. The latter include constructors to synthesise a point in the plane from its component coordinates, and a line segment from its endpoints, and selectors to extract coordinates from **point**s and endpoints from **line**s in the usual fashion. There are also geometric operators for rotating and scaling **shap**es, and an operator for combining two or more **shape**s. Explicitly, if X1, X2, ..., XN are **shape**s, their union is denoted by

$$X1 + X2 + ... + XN.$$

It comprises the multiset union of the sets of **point**s in X1,...,XN, and the multiset union of the sets of **line**s, together with the union of the sets of attribute values.

The underlying algebra also incorporates an arithmetic "**if** ... **then** ... **else** ..." operator, which can be used to define the value of a variable so that it depends upon a predicate. This is useful for instance when the size of the drawing represented by a **shape** variable falls below a critical value, and some simplification is required for purposes of display.

It is convenient for the user to be able to define non-standard operators, and a means of extending the underlying algebra is provided. It should be noted that, when specifying such an operator, an appropriate programming paradigm for defining a function is required, rather than a medium for dialogue. Ideally, a functional programming language might be used for this purpose, but this would require a major extension of DoNaLD in respect of both design and implementation. In the current design, a simple procedural programming notation is employed for the definition of new operators. This takes the traditional form, comprising procedural variables to represent explicit **real**, **integer**, **point**, **line** and **shape** values, together with a **for**-loop, and a simple alternative (**if** ... **then** { ...} **else** {... }) construct.

In the design of DoNaLD, the medium used for dialogue is deliberately separated from that used to specify new operators, and there is no provision for the declaration of procedural variables, or the use of the **for**-loop and alternative constructs outside the body of a user-defined operator. The syntax used to specify user-defined operators is in any case of peripheral relevance; what is of primary importance is that such operators should be pure functions. (For more details, see [2].)

DoNaLD includes typed variables of several different kinds. These comprise **integer**, **real**, **point** and **line** variables, and two kinds of **shape** variables (to be described below), together with indexed variables defining arrays of variables of each sort. An indexed variable v with type T and dimension N represents an array of variables v[1],...,v[N] of type T. Each component of an indexed variable can appear as an *l*-value provided that it is referenced using an explicit index, so that a valid *l*-value associated with v must be of the form " v[E] ", where E is an integer expression containing no un-evaluated variables defining an index in the range 1 to N.

The problems associated with the complex data types **vertex** and **colour** which arise in ARCA are circumvented in DoNaLD (see [1]), since both **point**s and **line**s have but two components. However, designing the variables to be used for representing values of the complex data type **shape** in DoNaLD poses similar problems to those described in connection with ARCA **diagram** variables in [1]. The essential issue is the mode of abstraction by which a **shape** variable represents a line drawing: whether for instance a **shape** variable S is to be defined by a single expression of type **shape** (eg as the union of two other shapes) or, in a composite fashion, by defining its constituent **point**s, **line**s and scalar attributes via expressions of the appropriate type. As explained in [1], it is semantically indesirable to use these two modes of definition simultaneously in connection with the same **shape** variable, and the naive solution proposed in [1] is that **shape** variables should be respectively declared as *abstract* or *explicit* accordingly. More effective alternative solutions are described below.

In DoNaLD, the design of **shape** variables is further complicated by the difficulties of providing a satisfactory means of reference to constituent points and lines within a **shape**, and in particular by the need to identify those expressions which can denote *l*-values. It is essential to be able to refer to the constituents of an abstractly defined **shape** variable, for instance, but inappropriate to treat such a constituent as an *l*-value. It is tempting to suppose that the **point**s and **line**s asssociated with a **shape** value X are best referenced by introducing selection operators of the appropriate types. However, there is no satisfactory way of introducing such operators into the underlying algebra unless the component points and lines in a **shape** value have an intrinsic means of reference associated with their geometric values. In other words, the values in the underlying algebra would no longer be simply "line-drawings", but *labelled* line drawings, and the definition of operators acting upon shapes would be accordingly more complex. In order to avoid complicating the semantics of shape values in this manner, syntactic mechanisms for referencing all points and lines of the shape value specified by a particular shape expression are provided. This depends upon associating a set of valid labels with **shape** expressions in a manner sketchily described below: for full details, see [2].

Dealing with complex data types

The solution to the reference problem for **shape**s adopted in DoNaLD is based upon a representation of a line drawing by a **shape** variable resembling that of a file system directory as a union of files and subdirectories. Such a representation corresponds to an abstract view of a line drawing as a union of points, lines and sub-drawings. In effect, it is based upon a recursive specification of the **shape** data type, viz:

shape = set of **real / integer** attributes **+** set of **point**s **+** set of **line**s **+** set of **shape**s.

There are two kinds of variable of type **shape**; these are declared as "**shape**" and "**openshape**" variables, and are broadly analogous to "abstract" and "explicit" variables as described in [1]. The value of a **shape** variable is to be defined implicitly by means of an expression of type **shape**. An **openshape** variable, which resembles a directory, is composed of constituent **real**, **integer**, **point**, **line**, **shape** and **openshape** variables, and its value is defined componentwise by associating values with its constituent scalar attributes, points, lines and subshapes.

Each variable is either declared globally, or denotes a constituent of an **openshape** variable, which may itself be a variable of type **shape**. The value of a **shape** or **open-shape** variable which is declared as a constituent of an **openshape** variable X is a *sub-shape* of the value of X, comprising a subset of the set of **point**s and **line**s associated with X. The authentic variable name is that used to reference the variable from the global context, and is in general specified by a sequence of openshape variable names separated by '/''s to identify the enclosing openshape, followed by a local name to identify the appropriate constituent. (The syntax resembles filenames in UNIX directories.) A variable declaration thus takes the form

type *var_name*

where *type* is **integer**, **real**, **point**, **line**, **shape** or **openshape**, and *var_name* is of the form

context *loc_var_name*

where *context* is a concatenation

(*loc_var_name* '/')*

in which each *loc_var_name* references an **openshape** variable, possibly indexed.

The semantics of **integer, real**, **point** and **line** variables is straightforward. A declaration of the form

openshape S

identifies S as an *explicit* **shape** variable, which is not itself an *l*-value, but enables the subsequent declaration of attributes and components of S, as in

integer S/i; **point** S/p, S/P[4]; **line** S/t; **shape** S/V; **openshape** S/Z[3].

The subshape S can then be defined componentwise according to the normal semantic rules. In contrast, a declaration of the form

shape V

identifies V as a *virtual* **shape** variable, whose value and component structure must be defined by means of a **shape** expression, and, in particular, cannot be defined componentwise.

The introduction of **openshape** variables helps to alleviate the problems of referencing the components of line drawings, enabling components of distinct **openshape** variables to have identical names. The identifiers declared within an **openshape** variable V are referred to as the *valid labels* of V: if V is an openshape, and t is a valid label for V, then V/t can serve both as an *l*- and *r*-value. Referring to the constituent parts of a **shape** value defined by a virtual **shape** variable is a more vexing problem, and is solved by recursively defining the set of valid labels for any **shape** expression in an appropriate way. For instance, if p and q are valid labels in the **shape** expressions S and T respectively, they are in general both valid labels for the union S+T, though a special convention for disambiguation is required if p and q are identical (see [2] for details).

The hierarchical approach to the definition of shapes used in DoNaLD provides one solution to the problem of referencing complex data types in a definitive notation, and may be contrasted with the solution which has been adopted in ARCA. The essential principle used in ARCA to give maximum flexibility in defining and manipulating **vertex** and **colour** expressions (representing vectors and permutations with **integer** components), and **diagram**s (comprising lists of component colours and vertices), is to devise an auxiliary definitive notation for declaring the **mode** of abstraction through which a variable represents a value. From this perspective, the introduction of **openshape** variables represents an alternative mechanism for manipulating the mode of a **shape** variable in the sense of [4]. For example, the sequence of **mode** declarations

mode D = 'abc'- **diagram**
mode a_D = **abst col**
mode b_D = **col** 4

mode c_D = **mode** b_D
mode D!4 = **vert** 2
with int : I = 1..3 **do**
mode D! I = **abst vert**
od

might be used in ARCA to declare a **diagram** variable in which the constituent **colour**s b_D and c_D and the **vertex** D!4 were to be specified component by component, and the remaining vertices and colours were to be specified abstractly. The effect of such a **mode** declaration can be conveniently depicted diagrammatically, as in Figure 1, where the types tagging the leaves of the tree indicate the type of the formulae to be used in defining the value of the variable D. Figure 2 depicts the "mode" of the **openshape** variable S defined by the sequence of DoNaLD declarations:

openshape S, S/T, S/R, S/T/X
shape S/V, S/T/W
point S/p, S/q, S/T/a, S/T/b, S/R/c, S/T/X/d
line S/l, S/m, S/T/n, S/T/X/z

using a similar convention.

Extensions to the basic syntax

The description of the DoNaLD notation above contains all the essential concepts underlying the practical system as envisaged. That is to say, any user action can be interpreted as declaring, defining or evaluating variables, or as introducing new operators into the underlying algebra. The rudimentary notation so far introduced can be tedious to use directly, however, and it remains to explain how to develop an effective interface to the user. It will be convenient to describe this development in two stages, by first considering syntactic extensions to the basic notation (essentially the addition of some simple macros), and then indicating how these can be incorporated into a window-based interface.

The natural first step towards a simpler interface is the introduction of an analogue of "changing the directory" in a file system; the hierarchical view of a line drawing as incorporating sub-drawings can then be reflected in the manner in which it is defined and referenced.

Formally, the set of variables associated with the **openshape** variable V consists of the points and lines of V, together with local scalar variables and all variables associated with subshapes of V. Definitions of the constituents of **openshape** variables are governed by a single *scope rule*: each subshape V/S of V, and all variables associated with V/S must be defined in terms of variables associated with V. Conceptually, the actions in a DoNaLD dialogue can be viewed as taking place in the context of a single universal **openshape** variable, whose constituents are the **point**, **line**, **shape** and **openshape** variables not contained in any user specified **openshape** variable.

The construct used to specialise the dialogue to a particular context, and provide for all references to variables to be interpreted relative to an **openshape** other than Ω is:

within *context* { .... }

where a general sequence of actions, possibly including further **within**-clauses, is specified between the braces.

Reference to the scope rules reveals a feature which causes inconvenience here: a variable v within the openshape S (and necessarily not within any **openshape** subshape of S) can be defined in terms of variables defined in the enclosing context for S, rather than within S itself. To avoid having to leave the context of S in order to make such a definition, each variable associated with the enclosing context for S in the expression defining v can be prefixed by an escape symbol "\", to indicate that it is to be interpreted with reference to the enclosing context for S.

The central virtue of the definitive programming paradigm - as explained in detail in [1] - is that it allows the state of a suspended dialogue to be described solely in terms of variable definitions which can subsequently be determined for resumption. A potential disadvantage of using a definitive notation is that the process of making definitions one at a time can be tedious, and some way of enhancing the notation to allow definitions to be introduced more conveniently and efficiently is required. From a design perspective, there is a problem of ensuring that any procedural variables introduced to this end are clearly distinguished syntactically from the authentic variables in the dialogue. On the other hand, since the state of dialogue described by a set of definitions is independent of the manner in which those definitions are generated, relatively unsophisticated devices - essentially simple macro facilities - are acceptable. There is also another pretext for avoiding verbose syntactic forms: as explained below, the user input is ultimately to be entered in dialogue boxes associated with windows of definitions.

Multiple definition is based upon a simple syntactic device:

v1,v2,...,vN = f1(a1,b1,...,z1), f2(a2,b2,..., z2) ,...., fN(aN,bN,...,zN)

being synonymous with

v1=f1(a1,b1,...,z1); v2=f2(a2,b2,..., z2) ;....; vN=fN(aN,bN,...,zN),

subject to the semantic correctness of this sequence of definitions.

Parametrised expressions - essentially macros - can also be used to specify sequences of definitions. In such expressions, the parameters may represent values of type **real**, **int**, **point**, **line** or **shape**, but can only be defined explicitly. A lexical convention whereby all parameter names begin with the symbol '$' is adopted, to avoid potential confusion with variable names.

The syntax of DoNaLD includes a method of specifying a list of expressions in an abbreviated form; this can be used as a macro generating facility in definitions, aliases, parameters or expressions etc as required. In the simplest non-trivial cases, such an abbreviation for a list is given by a **range**-specifier such as

over *type par1* **in** *range1* , ... , *type parN* **in** *rangeN* **range** f(*par1*,*par2*, ..., parN)

in which the f() is an expression in the formal parameters *par1* ,..., *parN*, and each parameter *parI* is constrained to range over the values specified by a range specifier *rangeI* . Explicit range expression are used as the basis of this recursive definition; these may be explicit lists of expressions separated by commas, or can be represented indirectly by special notations. In particular, an interval of integers can be described the form

I_exp1 **..** *I_exp2*

where the integer expressions *I_exp1* and *I_exp2* represent constant values c1 and c2 respectively, and c1 < c2. If V is a shape expression, the lists of **real**, **int**, **point**, **line** and **shape** labels associated with V are respectively denoted by

reals(V), **ints**(V), **points**(V), **lines**(V) and **subshapes**(V).

It should be noted that range expressions do not represent variables of **list** type, but are simply used as an abbreviation for a list of expressions. A general range specifier then consists of a list of range specifiers separated by commas, to be interpreted as a linear list of expressions in the obvious fashion.

To illustrate the use of range specifiers in conjunction with multiple definition:

over int $1 **in** 1..7 **range** a[$1] = f, **over int** $1 **in** 1..3, **int** $2 **in** 1..2 **range** g($1,$2)

is an abbreviated - even cryptic - form denoting the sequence of definitions:

a[1]=f; a[2]=g(1,1); a[3]=g(1,2); a[4]=g(2,1); a[5]=g(2,2); a[6]=g(3,1); a[7]=g(3,2).

This example also illustrates the significance of being able to determine the effect of simultaneously introducing many definitions upon a dialogue *without* needing to consider the way in which these definitions were generated.

The user interface

A typical DoNaLD dialogue includes many definitions, not all of which can be conveniently displayed at once, reflecting different levels of abstraction in the description of a picture. For instance, a particular **openshape** variable R to describe a room layout may incorporate subshapes to represent furniture within the room. These subshapes might include an **openshape** variable R/C to describe a chair, and additional **shape** variables to represent other chairs defined on the same pattern as R/C. The user-interface for DoNaLD is designed to reflect the way in which relationships at each level of abstraction are captured by a set of definitions, and is based upon a family of windows associated with **openshape** variables.

With each **openshape** variable, there is an associated context window, in which the appropriate local definitions are displayed together with the names of any local **openshape** variables, and through which the relationships within that context can be viewed and manipulated. The context windows are organised in a tree structure reflecting the hierarchical relationships between contexts. At the highest level of abstraction, and the root of the tree, there is a global window, in which all the declarations and definitions of variables at the outermost level are displayed. Each **openshape** variable in the global context defines a subcontext at the next level of abstraction, and its associated window is a child of the global window.

The global window is effectively organised as a subcontext list comprising a list of **openshape** variables declared within the global context, and a body consisting of the set of current definitions of remaining variables declared within the global context, ordered in such a way as to reflect the dependencies between variables. Note that the body may include definitions of component **integer**, **real**, **point** or **line** variables of an **openshape** which are externally defined. The subcontext list consists of the keyword **openshape** followed by the appropriate list of **openshape** variable names. In the body, the format for each variable definition is

var_type *var_name* **[** = *current_defining_formula* **]**

where the optional defining formula is omitted for variables which have been declared but not yet defined. In this context, the *var_type* is **int**, **real**, **point**, **line** or **shape**. The ordering of definitions within the body is such that the definition of a variable always follows the definitions of those variables upon which its value depends, and the initial segment of the body comprises variables which have been declared but not yet defined, followed by variables whose values are defined explicitly. For example, a global window display might take the form:

openshape A B C D
point p
line l
shape E
int i = 3
point O = {0,0}
real x = 1.4 * i
point q = {i,k}
line r = [**rot** (p, q, x), O]
shape S = A+B
shape T = **rot** (S, O, x+1)
....

For each **openshape** context there is a context window which is organised in a similar fashion, so that the entire set of context windows forms a tree. As mentioned above, in any definition, a reference to an external variable - necessarily a variable which is declared within the enclosing context - is preceded by a '\'. For instance, the context window for the subshape A above might include an integer attribute y defined as " y = \i-1".

At any given stage in a DoNaLD dialogue, a number of windows can be open, but only one window is currently active. In general this is the window associated with the current context. In effect, all declarations and definitions made whilst a particular context window is active are interpreted as being in the scope of a **within** *current_context* clause. Each context window has a header to specify the **openshape** variable name relative to the universal context, and a footer comprising a set of "buttons" which are used for interrogation of variables or to change the current context, together with a dialogue box in which new declarations and definitions can be entered.

Declaration, definition and evaluation

A typical action in specifying a DoNaLD line drawing is the entry of a declaration or definition of a simple variable within a context via the dialogue box. The effect of such an action will be to modify the associated context window, either by introducing a new identifier into the **openshape** list, or by declaring a new variable, or by altering or augmenting the current set of definitions. The actions which can be specified in the dialogue box of a particular context are confined to declarations and definitions valid within that context, but may include multiple actions specified by means of the range construct.

The buttons in a particular context are used for all forms of interrogation of local variables , and for context switching. These are identified by the keywords:

dep eval draw open close

and are generally to be used in conjunction with an appropriately selected identifier from the context window.

The **dep**(endency) button is used to display the set of definitions upon which the value of a **int**, **real**, **point**, **line** or **shape** (but not **openshape**) variable currently depends. It should be noted that the defining formula of a variable V in general includes operands whose definitions are not within the current context, such as S/a or S/T/p, where S is a **shape** or **openshape** variable. If S is a **shape** variable, these operands can be interpreted with reference to the current definition of S, which belongs to the dependency list of V. If S is an **openshape** variable, its value is not defined by a single formula, and cannot conveniently be displayed alongside the other definitions of variables upon which the value of V depends. To overcome this problem, the names of the **openshape** variables upon which V depends are displayed, together with the complete definitions of all variables on the dependency list for V in the current context. This can be implemented simply by highlighting the relevant definitions and **openshape** names in the current context window, or creating a temporary (read-only) window, similar in format to a context window, to display this information.

For an **openshape** variable, the function of the dependency button is served by the **open** button. The effect of this button is to change the current context to that of the selected **openshape** variable, and the current window changes accordingly. When a subcontext is entered in this way, the window associated with the original context remains open, but the context window for the appropriate subshape is opened, and is currently active. In this way, the set of context windows which is open at any stage defines a prefix of the tree of context windows. At any given stage in a dialogue, any open context window can be selected as the active window. The **close** button is used to close the window; the effect is to remove this window, together with any of its descendants in the context window tree, from the display, and to make the parent context window active.

The **eval**(uation) button is used to display the current values associated with variables. The effect of evaluation is to create an image of the current context window which corresponds to replacing every formal definition by an explicit one. The virtual context window thus created resembles a context window in that it includes a subcontext list and a body. The body is the result of transforming the definitions of all **int**, **real**, **point** and **line** variables in the active context window by replacing the defining formula on each RHS by its evaluation (denoted by @ if this is undefined), and the subcontext list is the result of replacing the list of **openshape** variable names by the list of all **shape** variable names in the current context. The virtual context window is read-only, and has no dialogue box. It has two buttons: **open** and **close**; these can be used to display the explicit values of the **shape** variables as if the virtual context window had been defined explicitly, and all subshapes within the context were defined by **openshape** variables. The names for components of the **openshape** image of a **shape** variable are determined by its valid labels. In this way - at the discretion of the user - evaluation can create a tree of virtual context windows which is rooted on the virtual context window created by the initial evaluation. Whilst the user is inspecting the evaluation, only the virtual context windows within this tree can be active; the evaluation then terminates when the user closes the virtual context window at the root of the tree. At this point, the state of the dialogue is as it was when evaluation was invoked.

Some method of inspecting and modifying the definitions of user-defined operators is also required: for consistency, this could be implemented by providing a window in which the names of user-defined functions were displayed, together with a button for opening an operator definition for editing.

For **point**, **line** and **shape** variables within the active context, the **draw** button provides an alternative method of returning a value, viz by display in a graphics window which is conceptually open throughout the dialogue, and is a read-only screen. By default, the screen is cleared prior to each **draw** request, but it is possible to select a set of

point, **line** and **shape** variables for simultaneous display.

Further directions

The design of DoNaLD as described above is currently being used as the basis for an implementation, and it is impossible to evaluate the interface at this early stage.

Several extensions of the basic interface described above might be considered. A means of directly editing definitions in a context window might offer an alternative to the use of a dialogue box. It might also be desirable to allow geometric information to be entered through the graphics screen: referencing, redefining or interrogating points as appropriate.

Additional facilities associated with displaying shapes will also be required. The attributes of a diagram are intended to have a role here, and might supply parameters to a shape drawing routine eg to specify colour.

It is to be hoped that the principles outlined in this paper can be successfully applied to more ambitious applications in due course. Further discussion of possible applications in connection with computer-aided design is to be found in [4].

References

1. Meurig Beynon "*Definitive notations for interaction*" in hci'85, CUP Sept 1985

2. Meurig Beynon, David Angier, Tim Bissell, Steve Hunt
"*DoNaLD: a line-drawing system based on definitive principles*"
University of Warwick CS Research Report 86 (October 1986)

3. Meurig Beynon

"*ARCA: a notation for displaying and manipulating combinatorial diagrams*"

University of Warwick CS Research Report 78 (July 1986)

4. Meurig Beynon

" *Definitive principles for interactive graphics*"

Proc. Nato ASI "Theoretical Foundations of Computer Graphics and CAD", July

1987

5. Meurig Beynon, Kevin Murray

" *The revised ARCA definition* " <in preparation>