20

The integration of functional and visual programming for the development of a knowledge based interface

C. Standing
Department of Information Systems, Edith Cowan University
Joondalup Drive, Joondalup, Western Australia 6027
email: C.Standing@cowan.edu.au

G. G. Roy
Department of Computer Science, Murdoch University
Murdoch, Western Australia 6150 email: geoff@cs.murdoch.edu.au

Abstract

One of the requirements of a more knowledgable GIS involves the representation of knowledge related to a number of features: the data in the system; the operations that can be performed; the processing of requests; and the presentation of results. Many of the ideas presented can be generally applied to other query language driven systems.

The approach taken to the design and implementation of a prototype high-level interface to GIS is based on the functional style of programming. Functional languages appear to offer some important properties, for example: the strong and polymorphic typing and the ease with which new types can be defined; the ability to order the knowledge base within functions; and the facility to create functional hierarchies composed of compound or higher-order functions which allow high-level operations to be manipulated as units. The paper describes how a functional solution to the problems can be represented visually through the use of a graphical user interface with direct manipulation of objects/icons.

Keywords

Functional programming, visual programming, interface, GIS

1 INTRODUCTION

The development of an easy to use interface strategy for geographical information systems (GIS) is difficult due to the complexity of the systems and the range of commands available. The paper contends that certain types of information should be made use of by the interface to detect errors at an early stage of query formulation. Through the integration of functional and visual programming approaches an interface has been designed that provides the user with an unambiguous environment to formulate queries.

S. Spaccapietra et al. (eds.), *Visual Database Systems 3*

2 THE NEED FOR KNOWLEDGE

The particular problem that this research addresses concerns the process by which a user interacts with a GIS and how the design of such a system's interface can be developed to maximise the potential value of the GIS for that user. The approach taken by the research to the design of the user interface is one which brings this programming function much closer to the problem specification.

The research proposes that the incorporation of rules and knowledge in the interface can ensure the validity of the query to be passed to the geographical information system. This validity refers to the steps involved in the process of building the query as well as the completed query.

There are a number of problems associated with using database interfaces; namely that databases give few clues about the nature of the data, little help in indicating which operations can be strung together, and even less help in indicating which commands can be applied to which data sets. This type of information or knowledge can be embedded in the interface and hence the user can be spared time and effort in constructing a query that fails because it is syntactically or semantically incorrect.

This paper considers the interface to one particular type of database application called geographical information systems (GIS). Whilst the examples used are related to one particular commercial GIS, ARC/INFO a product of Environmental Systems Research Group, the principles we feel are readily transferable to other database applications.

Types of Knowledge

To overcome the problems stated in using database systems the user interface has to address the following objectives:

- Knowledge about the data - data availability and type are clearly defined

At present most GIS maintain limited information on the data that they store. The commercial GIS - ARC/INFO for example, can provide a list of coverage names; the names, data types, widths of attributes in a coverage; and whether it is a point, line or polygon coverage is implied by the attributes and the file extension. This is a very limited amount of information. A new user to a system may have an obstacle to overcome in simply finding out what the data represents especially if much of the data is in a coded form. For example, data may be ranked but this may not be obvious from the field name. Equally data could be in nominal form but just what the code means could be lost to a new user; a code number of 1 might signify large cities but unless this is somehow recorded in the data dictionary that information is lost to those other than those who set up the files.

- Knowledge about the operations - operations available and syntax are clearly defined

It would be useful for the user of the GIS to have access within the GIS to a clear representation of knowledge about the operations it can perform. The system should know which requests are logically feasible. Many functions in ARC/INFO, as an example, are not polymorphic; they require certain types of data. So an intersect operation takes a first argument of point, line, or polygon and a second argument of type polygon. Applying the function to two line coverage arguments would be an invalid use of the operation. It is not the intention of the research to focus on putting right just the poor design features of a particular software package but rather, in this instance, to lessen the problems associated with applying commands to the correct type of data. This is a more universal problem which has to be overcome in interface

design. To solve this problem information or knowledge has to be stored in the GIS or GIS interface which represents the rules associated with each command.

- Knowledge about the sequence of processing - a query is specified in a non-procedural form

GIS are often presented as 'toolboxes', that is the user is provided with a set of available tools (perhaps expressed as commands) that enable a sequence of operations to be specified, and executed to give the required results. The effectiveness of the GIS is thus tied closely to how effectively the user can master the command language structure of the GIS. There are often long learning curves with associated significant time-costs in learning the command structure. The research is thus concerned with the processes for constructing command sequences which are required to fully utilise the powerful data processing analysis capabilities of most GIS.

- Integrated knowledge - queries are developed in such a way that are guaranteed to obey the constraints of the system both in the use of the command and the suitability of the application of the command to the type of data

It is clear there are problems in the design of GIS for the user; notably the limited knowledge represented within the systems and the time and difficulty involved in becoming a competent user because the user is effectively required to construct programs or 'macros' to perform sequences of low-level operations to carry out tasks. It is not difficult to appreciate the difficulties that the inexperienced user will get into. The required skills to become an efficient user are essentially the same as those required in writing programs in languages like FORTRAN, Pascal and C. The problems don't stop here. We also have significant problems associated with interpreting the user's *real* requirements in the formal command language. To illustrate the user's task consider this typical command sequence to produce a listing of all the schools within a 3 km radius of a particular bank:

```
RESELECT banks newbank POINT
RESELECT name = "ANZ Beldon"
BUFFER newbank bufbank # # 3 # POLY
INTERSECT schools bufbank schools_bufbank POINT
RESELECT schools_bufbank new_schools_bufbank POINT
RESELECT inside = 100
LIST new_schools_bufbank .PAT
```

Note that in this command sequence we are ignoring the fact that these commands are entered at various prompts (GIS system dependent) and that we only display the user input (to the GIS) and do not include what the GIS might display as a result.

In this command sequence the user is required to construct the commands in the correct sequence and syntactic form and to define new (intermediate) coverages or maps (eg. newbank, bufbank) along the way. The user must also be aware of the availability and type of the data so that commands can be correctly applied to the correct type of data. These are common problems in many programming tasks, especially within the procedural programming paradigm.

The above command sequence could be expressed more naturally as:

"Which schools are within a 3 km radius of the Beldon ANZ bank?"

To put this query to the GIS the user must:

- realise that the process requires the availability of the banks and schools coverages
- understand that a new coverage is required that contains just the Beldon ANZ bank
- recognise that a buffer area is required around the bank
- request an intersection of the two coverages
- select the schools that fall just within the buffer zone (inside = 100)
- list the resulting data
- determine the right sequence of steps involved
- express each command in the correct syntactic form

The knowledge that the system requires to know that the query is valid and that the data is available has to be provided within the interface or the interface has to have the capacity to collect its own knowledge, for example by interrogating the GIS database, or by requiring the user to provide some additional information. This arrangement is shown diagrammatically in Figure 1.

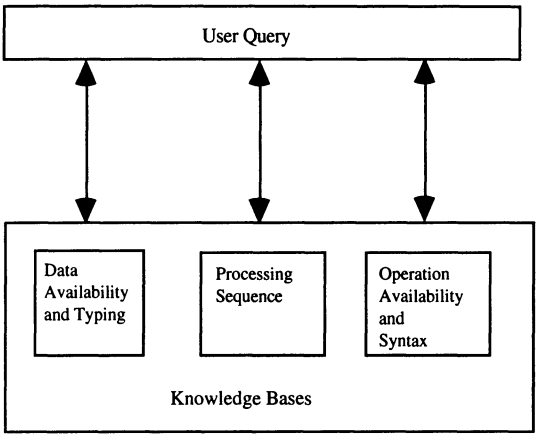


Figure 1 Knowledge incorporated within the interface.

Once we are sure that this user request conforms to the known rules and knowledge content of the GIS, then it can be formally translated into a procedural program which will provide the required results for the user.

3 FUNCTIONAL SPECIFICATIONS AND KNOWLEDGE

It is a hypothesis of the research that a functional specification along the lines as defined for standard algebraic expressions and the theory of functional analysis is a useful development method for meeting some of the objectives of the interface. This approach to problem solving is embodied in a range of programming paradigms, usually under the title of functional programming, eg. Bird and Wadler (1988), Henderson (1980) and MacLennan (1990).

What is functional programming?

As indicated by the name functions play an important role in functional programming. A function maps arguments to results. The function *half* maps an argument n to a result n/2. A program in functional programming is composed of function definitions and a sequence of function applications and therefore is rather like programming with algebraic expressions. An algebra is basically a set of values and the operations that are used on them. All functions map objects onto objects. Other functions are built using a fixed set of combining forms called functional forms. Functional forms combine existing functions or objects and return functions as a result.

Advantages of functional programming

- Referential Transparency

Better programming understanding can be achieved through programming without side effects. Imperative languages rely heavily on side effects whereas pure functional languages do not. They are not history sensitive and their computational results are merely temporary. It also means that because we are not concerned with how and when a function is called we can develop programs with more style and accuracy and which are not procedural in form. Field and Harrison (1988) explain referential transparency as:

"each expression denotes a single value which cannot be changed by evaluating the expression or by allowing different parts of the program to share the expression." (p.10).

This means that evaluating the expression changes the form of the expression but not its value. Any reference to a value is equivalent to the value itself and a reference to it from another part of the program does not change this. The problem with imperative languages such as Pascal is that global data can be used and this along with the use of the assignment statement means that a function value can change from one call to the next. It is these dynamic changes in the data, such as from true to false, that are called side-effects. It means that a function can have a different result even though the data passed as arguments are the same. Pure functional languages do not allow assignment statements as they are not concerned in the same way as imperative languages with updating a store. A functional program describes what needs to be computed not how!

- Strong Typing

The type of an object is the feature that characterises the aspect that the elements have in common. For example, red objects, blue objects, green objects have the type *colour*. Common data types used in programming are integers, characters, lists, etc.

In programming the programmer classifies data according to its type as a large number of errors can be detected by checking whether any values are wrongly used. It is this classification technique which is known as *typing* the data.

In a typed programming language, every value has an associated type. It is the type that defines how the value can be used. The numbers, both positive and negative (-1,0,1,2..) have an integer type. The values true and false are of type bool. The addition operator is defined to operate on pairs of type integer. The programmer can define a new set of values and provide a type name. The program then assures that operations use values of the correct type. Automatic type checking is seen as producing more reliable software (Reade, 1988). Strongly typed programming languages perform type checking and if this is done before evaluation then it is known as static type checking. Importantly, this allows many mistakes to be trapped early in the program development process.

- Abstraction - Higher Order Concepts

A common feature of programming is the use of abstraction techniques. Objects are classified according to similarities but they may still have many differences which are ignored for simplicity. Since the classification is made up to reflect the structure of some real world problem then it is abstract. Concepts are then combined into hierarchies that allow the problem structure to be defined.

Functional languages are often termed applicative, declarative or descriptive languages. Functional programs "can be viewed as descriptions declaring information about values rather than instructions for the computation of values or of effects" (Reade, C. 1988). With this approach we can develop solutions that allow us to describe the nature of the problem in the languages of the problem domain rather than having to be concerned with the lower level administrative features dealing with the machine architecture, a level that so many procedural languages work at.

Why a functional approach for interface design?

The method used takes a functional specification and automatically builds the procedural commands, in their correct sequence and with the correct syntax, which are sent to the GIS. An example of a functional specification using the sequence of commands shown earlier would be:

```
list(reselect(intersect (xdb "schools")(buffer (reselect (xdb "banks")(Name "ANZ Beldon")) 3))
inside = 100)
```

Although the functional query may not look especially elegant, it does express the semantics of the user query precisely and unambiguously. The GIS command sequences can be automatically constructed from this functional expression. Note with this specification the output of one function becomes the input of the next function with a function xdb that checks for the availability of the data. A system has been developed by the authors that incorporates all the rules and knowledge mentioned in a functional form taking as input a functional query as in the example just given (Standing and Roy, 1992).

We can take advantage of the properties of functional programs described.

1) In particular we can use the strong typing properties to ensure that functions are not applied to invalid data types. For example we need to ensure that a query to "list all the lengths of all points in a coverage" is detected as invalid since objects of the class "point" do not have a length attribute. In addition, we need to ensure that the data required for a function is available from the GIS database. In both these situations it is of some considerable advantage to be able to detect the problem before any attempt is made to formally execute the procedural specification.

   In a GIS there are other errors which are not picked up, but ideally should be prevented or the user warned about. A common source of error can arise from the use of an invalid statistical operation. For example requesting a mean of a set of ordinal (but still numeric) data may give meaningless results. The GIS may not be able to make this distinction, but our user interface should be aware of the problem. Providing data typing is enforced the problem of detection is greatly simplified. Thus the functional expression can be evaluated to produce a valid query, as a procedural program, before sending it to the GIS for final evaluation.

2) A function is considered to have a unique value, so that for a given set of arguments the value of the function is invariant, ie. there can be no side effects. We therefore need not

be concerned with how and when a function is evaluated, we can thus allow the reduction process to proceed with greater confidence.

3) Since we need be less concerned with how and when a function is evaluated we are freed from the burdens of a procedural style of programming.

4) If we are less concerned with how a function is evaluated (what is happening within the function) then we have a powerful abstraction technique where functions can be combined to develop higher order concepts which are nearer the language of the problem domain.

We can thus evaluate the functional expression to produce a valid query, as a procedural program, before sending it to the GIS for final evaluation.

4 VISUAL PROGRAMMING

Many of the problems of interface design discussed earlier relate to the problems of programming, that is programming in a traditional sense of being procedural and text based. Visual programming represents a departure from tradition and according to Shu (1988, p.7) is driven by the following premises:

1. Pictures are more powerful than words as a method of communicating as they can often concisely convey more meaning.
2. Pictures can help comprehension and remembering.
3. Pictures may provide an encouragement for learning to program.
4. Pictures, if designed properly can work across cultures and languages.

Our research contends that a visual style of programming development can play an important role in supporting some of the objectives of the GIS interface and in implementing the benefits of the functional strategy just described. In particular, a visual strategy has the potential to help in the following areas, data availability and type, operation options and syntax, the facility to express a query in a non-procedural form, and the use of the operation and its suitability of application to the type of data. If these objectives can be met in a visual way then validation of the program can be an inherent part of the programming process. This would then alleviate the need for the debugging stage.

At this point it is worthwhile defining the concept of visual programming as there are different interpretations of what it is. Shu (1988, p.7) takes the term *Visual Programming* to mean the "use of meaningful graphic representations in the process of programming" and she provides a useful framework for classification. However programming is a broad term that covers the language and the environment of specification; the specifications themselves; the process of validation and the display of data involved in the execution of the specification. All of these aspects of programming can have a visual representation but visual programming primarily covers the language of specification.

5 THE DESIGN FEATURES OF FVPL

The objectives of our user interface are as follows:

- data availability and type are clearly defined
- operations available and syntax are clearly defined

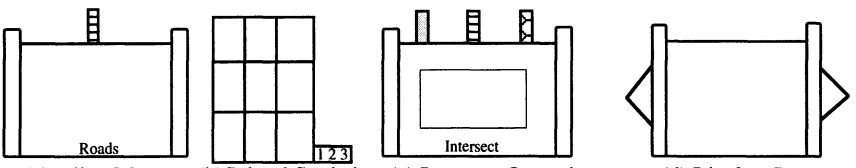
- a query is specified in a non-procedural form
- queries are developed in such a way that are guaranteed to obey the constraints of the system both in the use of the command and the suitability of the application of the command to the type of data

The system that has been designed as part of our research called FVPL (Functional, Visual Programming Language) considers these objectives and creates a visual programming interface that incorporates certain rules and knowledge that allows the validation of the query to be an inherent part of query formulation. An essential design feature of the interface needed to meet our objectives is that it must be easy to work with and incorporate the relevant knowledge and rules.

A number of graphical interfaces to GIS have been developed but many just concentrate on the presentation of data, for example ArcView, a product of Environmental Systems Research Institute (ESRI). CIGALES (Mainguenaud and Portier, 1990) is a graphical, query language for GIS that uses icons to represent data and operations but it uses little information about the data or operations to provide validation during the query building process. Graphic Map Algebra (Kirby and Pazner, 1990) uses a flowcharting structure with icons that represent map layers and operations to build up a picture of the query being developed. Once specified the graphic query is executed and errors reported. This approach has some similar attributes to our own but differs in two main ways: the validation of the query is not an inherent feature of the process of building a query; and because of its reliance on a flowcharting style the query is built up in a procedural manner rather than in a functional form as we propose.

What we are proposing are that icons and options on menus can be used to represent functions and data or maps so that their availability is not an issue. An operation icon can be opened to present a schema that requires certain inputs in order that constraints can be satisfied and data types matched. Schemas, representing the detailed functions, can be combined to create a visual, functional specification. The query can be constructed in a number of ways so users are relieved of the procedural burden of typical GIS command languages. As the query is built the constraints on the operations and data are embedded in the process and have to be met since each schema would have place holders for arguments (coverages, parameters, etc) that would have to be of the correct type. If a coverage of the wrong type, or a schema that resulted in a coverage of the wrong type was placed in a place holder then the inserted object would be highlighted to denote a mismatch and rejected. An example icon, representing a line coverage, is given below (figure 2 (a)). The rectangle with the lines inside is a peg that denotes the type of the icon or data. The symbol of a scrolled map denotes a map coverage. By this method, the data availability and its type becomes clear.

The icon shown in figure 2 (b) represents a table of data that is not directly part of a coverage. An icon can also represent an operation, such as an Intersect operation (figure 2 (c)). This is a polymorphic map type, as depending on the arguments the result of the function can be a polygon, line or point type.



(a)A line Map (b)School Statistics (c) Intersect Operation (d) Listdata Icon

Figure 2 Examples of icons used within the system.

This system relies on a well defined set of types used in geographical information systems such as maps, a sub-type being points, lines, polygons; character data and numeric data. Numeric data could be ranked, nominal or ratio and would be metadata on the attributes of a file. While the types that have been specified in the system are suitable to illustrate the objectives, other types could be introduced if necessary to improve the validation of the query. The ListData Icon (figure 2(d)) shows the output (to the screen) type. This function takes a map or table value as an argument and converts it to a printable representation, rather like the show function that appears in most functional programming languages.

A schema represents an exploded operation or function. The function has one or more arguments, of certain types. These arguments can be data, operations or attributes. The schema can be collapsed back into an icon, and will have a completed or incomplete state, which signals whether all the arguments or sub-arguments have been completed. The schema can be seen as a form which is usually completed by dropping icons into placeholders. The schema in figure 3 is for the *Intersection* operation, which takes a line, point, or polygon coverage and overlays this with a polygon coverage. The resulting type is the same as the type of the first argument.

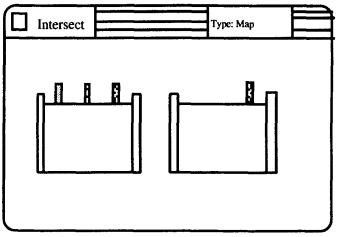


Figure 3 An exploded icon displays a schema that shows the arguments needed.

A different type of operation could take a table, operation or coverage as an argument and therefore would be a polymorphic function. An example of this type of placeholder would be *ListData* shown in figure 4 (a).

By making the data types a visual part of the system they are unambiguously defined. The user can see and therefore decide which type of argument a function requires and whether this is available. For example the roads icon (figure 4 (b)) is clearly the wrong type of argument for the polygon placeholder (figure 4 (c)).To attach an icon to a schema, the icon has to be dragged and dropped in the placeholder in the appropriate schema or similarly a schema is dragged and dropped in the placeholder. If there is a type mismatch the icon or schema will not be accepted by the placeholder.

In addition to selecting coverages as arguments some operations take attributes. Take the *mean* function: this requires a coverage or map type but also an attribute of that coverage. The attribute has an associated data type that is represented by an icon that appears in a pop-up window. Calculating the mean only has meaning when applied to certain types of data, for example it may be meaningless to find the mean of some nominal data or even an internal identification attribute for a feature. Often this function would lack meaning when applied to ranked data. For this example the mean can only be applied to ratio data.

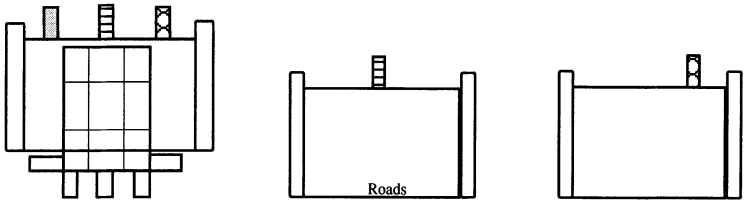


Figure 4 Type matching. (a) is a polymorphic type placeholder. (b) is a line icon and (c) is a polygon type placeholder

Figure 5 shows a pop-up window on the right with the attributes of the map type placed in the placeholder opened up. In this window the attributes are represented by names and icons which again visually signify the type. The icon has to be matched with the attribute placeholder in the schema. In this case the operator and a value have to be included which must comply with the attribute type.

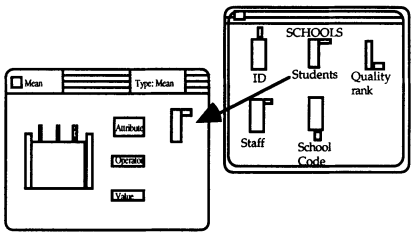


Figure 5 The mean schema and pop-up attribute window from which attribute icons can be dragged and dropped.

Any empty (no arguments completed) function is shown in figure 6(a). If it is partially completed the schema has an outline 6(b) and if all the arguments have been input then it is highlighted by the thick, dark border around the icon 6(c).

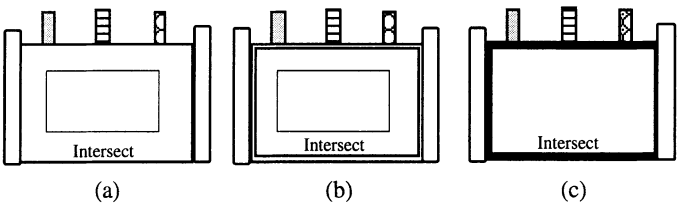


Figure 6 Icons shown in various states. (a) An icon with no arguments completed. (b) An icon with some arguments completed. (c) An icon with all arguments completed.

One of the objectives of the interface was for the process to be non-procedural. The following examples show the process of building a query and how the order of query construction

becomes less important. The query in figure 7 in natural language form could be expressed as *List the schools within a three kilometre radius of Beldon ANZ Bank.* Although this may seem a simple request it does involve a number of steps:

a) Taking out the Beldon ANZ Bank from the Banks coverage;
b) Buffering that bank to three kilometres;
c) Intersecting the buffered bank and the schools coverages;
d) Reselecting those schools within the buffered zone;
e) Listing the school names.

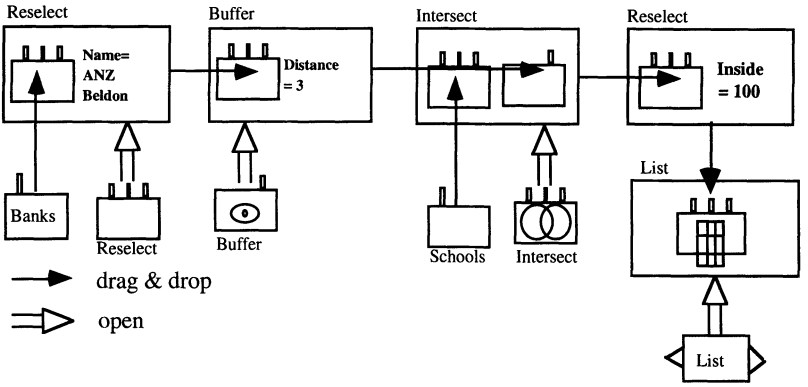


Figure 7 An example of a query being built up illustrating the drag and drop method.

The graphical, functional specification would be built up in this example by opening the *reselect* icon to present the *reselect* schema. The *banks* coverage would be dropped in the *reselect* place holder and the selection criteria added (by keyboard entry). This schema would form an object that could be placed in the *buffer* schema once opened along with the *distance* argument , which again would form an object to be dropped in the *intersect* schema as well as the *schools* coverage. The *intersect* schema would be placed in the *reselect* schema with added text input which as a whole could be placed into the *list* schema. The order in which the query is constructed is not important; the user could actually work backwards from the *list* operation and explode the icons that represent operations, for example *intersect*, and then complete the schema when required. Figures 8 and 9 show the first stage of the query being completed but this time showing the schemas closed to icons which are then dragged and dropped in the placeholders instead of schemas being dragged and dropped.

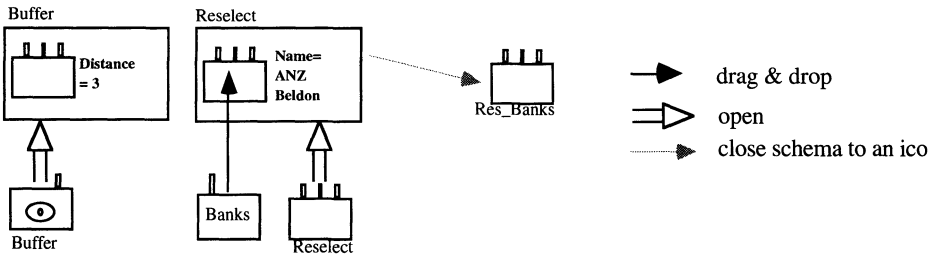


Figure 8 Open buffer icon to schema and complete reselect.

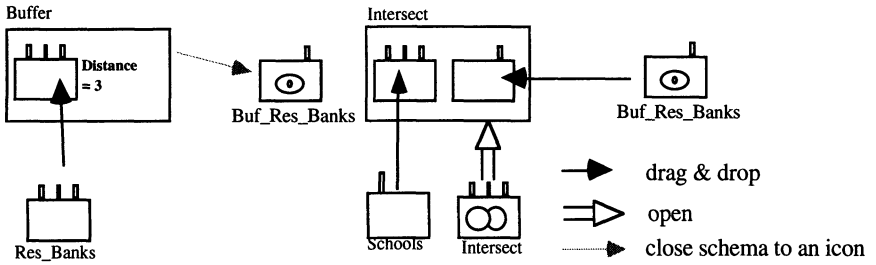


Figure 9 Complete buffer schema and close to an icon and complete intersect schema.

The completed, exploded functional specification is shown in figure 10. A query can be exploded by double clicking on the icon within a schema. The query can still be edited in this mode by dragging and dropping icons and schemas.

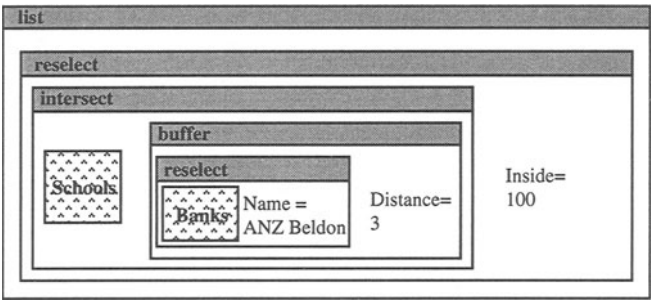


Figure 10 An explosion of a query showing the higher order nature of the approach.

The graphical interface involves validating functions as the overall query is being constructed. It is clear whether the constraints are met as each schema is completed. Once the query has been fully specified the user knows that it is a valid query. The graphical specification can be translated to a text based functional specification which can then be reduced to create the procedural sequence of commands to be passed onto the GIS. This functional, graphical user interface would appear to satisfy all of the objectives (Standing and Roy, 1992b).

6 SUMMARY

The paper started with outlining the objectives of the interface in terms of the knowledge content. Through the tight integration of the visual and functional strategies described the data availability and type are clearly defined by the use of icons with distinguishing features. If data is unavailable then no icon is shown. Similarly, the operations available in the system have an iconic representation. The set of operations is extensible since the user can create reusable

higher-order functions. The syntax of the language is clearly defined by using placeholders into which only certain types may be dropped. The user has the freedom to construct queries in any order. The screen can be seen as a work area where the query can be pieced together. The query constructed at the interface level is guaranteed not to fail as all the testing of the query is carried out within the interface so the user is provided with feedback before any command is passed to the GIS. This means we do not have to rely on the GIS to report errors or bad commands, and more importantly we have a good chance of detecting operations which might be considered invalid on certain types of data before an invalid (and possibly undetected) result is produced.

7 REFERENCES

Bird, R. and Wadler, P. (1988) *Introduction to Functional Programming.* Prentice Hall.
Field, A.J. and Harrison, P.G. (1988) Functional Programming. Addison-Wesley.
Henderson, P. (1980) *Functional Programming: Application and Implementation.* Prentice-Hall.
Kirby, K. and Pazner, M. (1990) *Graphic Map Algebra.* Proceedings of the Fourth International Symposium on Spatial Data handling, Zurich, Switzerland.
MacLennan, B. J. (1990) *Functional Programming: Practice and Theory.* Addison-Wesley.
Mainguenaud, M. and Portier, M. (1990) *CIGALES: A graphical query language for geographical information systems.* Proceedings of the Fourth International Symposium on Spatial Data handling, Zurich, Switzerland.
Reade, C. (1989) Elements of Functional Programming. Addison-Wesley.
Shu, N. C. (1988) *Visual Programming.* Van Nostrand Reinhold Company, New York.
Standing, C. and Roy, G. G. (1992) *The requirements of a high-level interface to Geographical Information Systems.* Proceedings of the Fourth Annual Colloquium of the Spatial Information Research Centre, University of Otago, Dunedin, New Zealand.
Standing, C. and Roy, G. G. (1992b) *User interface design for Geographical Information Systems.* Proceedings of the Third Annual Research Conference of the Computer Science Department, University of Western Australia, Perth, W.A.

8 BIOGRAPHIES

Geoffrey Roy

Geoffrey Roy is Professor of Computer Science at Murdoch Univeristy, Western Australia. He worked for a number of years in the areas of expert systems and the design of user interfaces in CAD environments. Areas of application have included regional planning, building design, menu system design and use of codes of practice in design. A current research project involves the development of visual, functional programming languages.

Craig Standing

Craig Standing is a lecturer at the Department of Information Systems at Edith Cowan University and is currently working towards a Ph.D. at The University of Western Australia. He obtained his B.A.(Hons) in Geography at the University of Lancaster, UK., and his M.Sc. in Computation at the University of Manchester Institute of Science and Technology. His research interests include: visual programming languages, and systems development methodologies.