HIGHER ORDER FUNCTIONS NECESSARY FOR SPATIAL THEORY DEVELOPMENT

Andrew U. Frank
Dept. of Geoinformation
Technical University Vienna
frank@geoinfo.tuwien.ac.at

Abstract

The tool we use influences the product. This paper demonstrates that higher order functions are a necessary tool for research in the GIS area, because higher order functions permit to separate the treatment of attribute data from the organisation of processing in data structures. Higher order functions are functions which have functions as arguments. A function to traverse a data structure can thus have as an argument a function to perform specific operations with the attribute data stored. This is crucial in the GIS arena, where complex spatial data structures are necessary. Higher order functions were tacitly assumed for Tomlin's Map Algebra.

The lack of higher order functions in the design stage of GIS and in the implementation is currently most felt for visualization, where the problems of the interaction between the generic computer graphics solutions and the particulars of the application area preclude advanced solutions, which combine the best results from both worlds. Similar problems are to be expected with the use of OpenGIS standardized functionality.

This paper demonstrates the concept of higher order functions in a modern functional programming language with a class based (object-oriented) type concept. It shows how the processing of data elements is completely separated from the processing of the data structure. Code for different implementations of data structures can be freely combined with code for different types of representation of spatial properties in cells. The code fragments in the paper are executable code in the Gofer/Haskell functional programming language.

1 Introduction

The tool used influences the product. This is true for industrial production as well as for research. Nevertheless, there is little discussion in the research community about the tools they use - e.g., formalization methods - and how they influence the research directions. Most research in Spatial Information Theory (Frank and Campari 1993)

is carried out using first order languages. I feel that this restriction is an unnecessary limitation, which makes researchers focus on static relations and precludes adequate treatment of processes. Higher order functions can take this hurdle, but are generally useful to help with the design of GIS architecture. This paper introduces the concept of higher order functions and demonstrates their use for the separation of treatment of data structure and object data. Other beneficial uses of higher order functions are left to be treated in other papers.

Higher order functions are functions which have other functions as arguments. Many programming languages (e.g., Pascal, C, C++) allow such constructions, but generally they are added *ad hoc* and not fully integrated. Higher order functions are well understood mathematically and functional programming languages cleanly implement these concepts.

Research in GIS has been hindered by a mixture of discussion levels: application specific issues, problems of efficient implementation and topics from spatial theory are all treated at once. The connection between these topics is clearly necessary to avoid developing theories for which there is no use in the real world, or the development of tricky solutions without the benefit of a theoretical understanding. But the resulting breadth of the arguments makes it often difficult to detect the essential features.

The formalization in GIS is most often carried out only at the implementation stage. The programmer must resolve all details, from application details to fundamental issues in mathematics. Different concerns are unseparably intertwined: GIS code is complex to understand, loaded with detail, and seldom useful to gain insight. If any formalization is attempted, GIS research has used most often first order languages for formalization (Egenhofer and Herring 1991; Frank 1996a; Pigot 1992; Worboys 1992). It is tacitly assumed that the described functions are integrated in a program system, but the overall architecture of this system cannot be described in a first order language.

Higher order functions are a powerful conceptual tool to separate concerns at different levels, for example data structure and processing of data elements embedded in the data structure. This is crucial for GIS, which are large data collections and require specific, highly optimized and, in consequence, complex data structures (Samet 1989a; Samet 1989b). The design of the operations treating the attributes of the features and the design of the processing of the data structure must be separated and solutions freely combined. For computer cartography, higher order functions open a door for the separation of the different concerns in the map rendering process: graphical issues, management of screen real estate, geometric map projection etc.

This suggests that higher order functions are a necessary tool for systematic research in geographic information systems and their theory. Tomlin in his Map

Algebra (Tomlin 1983a; Tomlin 1983b; Tomlin 1989) has tacitly used higher order functions and one can attribute some of the success of this concept to the clarity of the resulting framework. Map Algebra cannot be formalized without higher order functions (or some rectification of it). Higher order functions are important also to design visualization systems for cartography, where computer graphics tools and cartography demands must be linked. They will be important to combine the generic GIS operations in the forthcoming Open GIS modules.

This paper demonstrates the concept of higher order functions in a modern functional programming language. As an example for demonstration, the separation of operations on pixels and the traversal of the data structure (e.g., quadtree) are explained. It shows how the processing of data elements is completely separated from the processing of the data structure. Code for different implementations of quadtree structures can be freely combined with code for different types of representation of spatial properties in cells. Fragments of actual executable code are given in the Gofer/Haskell language(Hudak et al. 1992; Jones 1991). This is a modern functional programming language, which is class based to provide an object-oriented type concept.

The paper is structured as follows: the next section introduces the concept of higher order functions. Section 3 introduces functional programming languages and gives some examples for higher order functions. The next section introduces the fundamental operations `map` and `fold`, which apply a function to a data structure. Section 5 applies this to operations from the Map Algebra and Section 6 sketches an application to GIS query language and visualization, followed by conclusions.

2 Higher Order Functions

Higher order functions are functions which take functions as arguments. Higher order functions are the discriminating property of higher order languages, which means that first order languages do not permit to pass functions as arguments into functions.

As a metaphor, higher order functions can be seen as mechanical power tools, into which different drills or blades can be inserted to perform different operations. A simple example: a traversal operation is accessing every element of a list and applies an operation which is passed as an argument to every list. This operation can be used to double the value of every element in the list (when a function which multiplies by 2 is passed), can set all values to 0 (when the function returns always the value 0), reduce the values by 1, etc. This is similar to the instruction in everyday life to clean dishes ('take each dish in sequence and wash and dry it'). It is also the operation to draw a map: "Take these (selected) objects and draw each of them according to the scale and legend".

A formal language is called first order, if the symbols can range only over individuals (this precludes quantification over functions and functions which have functions as arguments). It is called second (or higher) order if symbols can range over relations and functions. The ordinary (first) order predicate calculus is a first order language; relational database and Prolog are using first order languages as bases.

The concept of higher order functions is so powerful that even standard programming languages (like C (Stroustrup 1986 p.127), C++ (Ellis and Stroustrup 1990) or Pascal (Jensen and Wirth 1975)) contain it. There are constructs provided which allow to pass a function (or procedure) into another procedure or function as an argument, but the integration of this element of higher order logic with the remainder of the programming language is typically restricted. In Pascal (Jensen and Wirth 1975), the use of this tool is limited: 1) functions cannot be assigned to variables, 2) passing functions circumvents some type checking and is therefore insecure. In C++ a special 'iterator' concept is provided (but tricky to use) to save the programmer the difficulties with passing functions as parameters. The programming languages used for implementation are based on variables and statements and functions remain second class citizens.

3 Functional Programming Languages

The function is the fundamental building block of a Functional Programming Language: everything is a function; functions with arguments, functions without arguments (which are constants) and functions which produce functions as result. Functional programming languages are as old as Fortran. The functional languages, which are strictly typed, use a type system in which functions have proper type and type checking includes the passing of functions as arguments (Milner 1978). Examples of functions are the function `add (+)`, which has two arguments and as a result computes the sum of the two arguments. Its type is written as `(+)::Int -> Int -> Int`. A function can also be user defined, e.g., a function `f (x)` which computes `3x + 2` and has type `f:: Int -> Int`. The constant π is a function with type pi::Float.

To introduce the concept of a higher order function, a function which applies a given other function twice is used. It is demonstrated with the functions to increment `inc`. The code is written in the language Gofer (Jones 1991), which is related to Haskell (Hudak et al. 1992)[1]

```
twice :: (Int -> Int) -> Int -> Int
twice x a = x (x (a))
inc :: Int -> Int
inc x = x + 1
twice inc 4 --->> 6
```

[1]Functional programming languages write functions and their arguments without parentheses: `f x`. Parentheses are only used for grouping expressions and do not indicate function application as in C++.

The fundamental operation is the evaluation of an expression, composed of some functions. An `if-then-else` expression and recursion are the fundamental control structures. Recursive data structures like sequence or tree fit best. The power of functional programming language is often attributed to built-in operations to treat lists; dynamic arrays were predefined with a very powerful set of operations in APL. As an example, a list and a tree are defined recursively (these data types are typically predefined). Code to sum the elements in the list and to count the leaves of a tree are given

```
data List a = Empty | Element a (List a)
data Tree b = Leaf b | Branch (Tree b) (Tree b)
sumList (Empty)= 0
sumList (Element x xs) = x + (sumList xs)
countTree (Leaf b) = 1
countTree (Branch a b) = (countTree a) + (countTree b)
```

4 Higher Order Functions to Map Operations to Data Structures

Complex objects are described by a collection of values, collected in a data structure. Much of the GIS literature is concerned with data structures for geographic data and the efficiency of particular operations on these data structures. The introductory examples here are a polygon as a list of coordinate pairs and a tree with the names and populations of towns in a county.

Operations on a data structure consist typically of code to traverse the data structure, i.e. code which decides which data element is considered next, and code which deals with a single data item. There are two variants, which are often used :

- map: e.g., each coordinate pair of a list is to be scaled.
- fold: e.g., the total population of all the towns in the tree of towns is summed.

Both these operations could be written easily in Pascal as loops over an array, but for realistic applications, more complex data structure will require more code. This code is essentially the same for any operation of this kind for a given data structure. Many lines of Pascal or C++ programs are filled with code controlling the traversal of data structures.

Higher order functions permit to separate the coding of the traversal of the data structure from the operation on the data element. In a functional programming language, the code for these two operations is:

```
scaledCoordList coordList scale = map (scaleTransformation scale)
coordList where
      scaleTransformation scale (Coord x y) = Coord (scale * x)
(scale * y)
totalPop statePop = fold ((+).pop) 0.0 statePop
```

Here, the `CoordList` contains the coordinate pairs to scale, `scaleTransformation` is the operation to change the scale; `statePop` contains the state population by county, `pop` gets the population figure from the record for a

county. The population values are then summed up (starting the count with 0). The two higher order functions `map` and `fold` are defined in the next subsections.

4.1 Map Function

An operation φ is applied to each element in a structure and yields a new data value. The result is a data structure of the same characteristics, but with different values possibly of different type. Examples: a list of reals can be rounded into a list of integers, coordinates in a list can be scaled (as above), or the record structure changed, e.g., to replace records with county name, population and area with records, which contain county name and population density.

The higher order function `map` applies function with signature phi::(a->b) to a data structure `f` with elements of type `a`. The result is a data structure `f` with elements of type `b`[2].

```
map :: (a -> b) -> f a -> f b
```

In many cases φ is a function which returns the same type as its input `(phi:: a->a)` and the result of mapping `a` on the data structure `f a` is again a data structure with type `f a`. This is, for example, the case, if coordinate pairs are scaled and the result is again a coordinate pair.

The `map` function defined above must be specialized for the data structure used. Assuming the definition for list and tree given above, mapping an operation `phi` to the list is simply applying it recursively to each element, mapping the operation `phi` on a tree is applying it to each leaf:

```
map phi (Empty) = Empty
map phi (Element x xs) = Element (phi x) (map phi xs)
map phi (Leaf b) = Leaf (phi b)
map phi (Branch x y) = Branch (map phi x) (map phi y)
```

With these definitions the code for scaling a list of coordinate pairs given above works. It can be used to update the population count of a list of cities with an operation `updatePop`, which is then mapped to the list of Cities[3]:

```
updatePop (City name area pop) = City name area (pop*1.1)
updatePop listOfCities = map updatePop listOfCities
```

4.2 Fold Function

To compute the total population of a county the population of its cities must be added. The input is a data structure, not a data structure as for `map` above. This

---

[2] In order to use higher order functions effectively, the type system must allow parametrized types, i.e. lists of integers, lists of reals etc. In the example f a means a data structure of f with elements of type a.

[3] `updatePop` is used here in a polymorphic fashion: `updatePop` applied to a data element of type city uses the operation definition of the first line, `updatePop` applied to a list of cities, uses the second definition, which calls the first for each city in the list. The type system of a polymorphic language controls this.

operation is called `fold`, because it reminds of folding a piece of paper over and over (Figure 1):

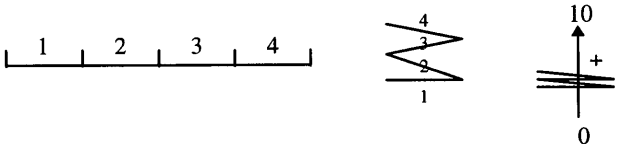

Figure 1: Folding operation

The operation applied in `fold` combines the current value with the result of the previous application. A start value must be given[4] (for the sum above, it must be 0). The standard example is the calculation of the values of the figures from 1 to 100:

```
fold :: (a -> b -> b)-> b -> f a -> b
fold (+) 0 [1..100]
```

The higher order function `fold` has as a first argument the function, as a second argument the start value and as a third argument the data structure. It may be necessary to have two `fold` operations, which operate from right to left and from left to right. The function which is used to fold the data structure must have the signature `psi :: a -> b -> b`, having one argument which is the same type as the data in the data structure and a second argument, which has the same type as the result (these two types can be the same, and most often are). This is necessary to make the result of one application the input for the next application of the function (with the next data value from the data structure).

The definitions for the list as defined above is:

```
fold :: (a -> b -> b) -> b -> f a -> b
fold f z (Empty) = z
fold f z (Element x xs) = fold f (f x z) xs
```

4.3 Combinations of Operations

To compute the total population from a tree of records with county name and population count one can proceed in two steps: 1) map from the record a single population count (with an operation `pop`) and 2) fold with `(+)`.

```
pt = map pop popTree
totPop = fold (+) 0.0 pt
```

Most functional programming languages are referentially transparent. Therefore equals can be substituted with equals. This makes reasoning about programs similar to reasoning in ordinary mathematics and avoids the complications of the temporal reasoning with pre- and post-conditions necessary for commercial programming languages like Pascal or C++. The rule given in the next line can be applied to the

[4] The start value is typically the *unit value* for the operation used, for (+) the value must be 0, for folding with (*), it should be 1 etc. Mathematicians call a group an algebraic structure, consisting of a set of values, an operation and a *unit value 0*, for which the axioms $a + 0 = a$ and $0 + a = a$ hold.

combination of the two functions above (in `totPop`) and yields the simplification `totPop'`.

```
fold f z (map g xs) = fold (f.g) z xs
totPop = fold (+) 0.0 (map pop popTree)
totPop' = fold ((+).pop) 0.0 popTree
```

There is a similar rule to combine multiple mappings:

```
map f (map g x) == map (f.g) x
```

In general, these simplifications are automatically done by the compilers for modern functional languages.

5 Application to GIS: Map Algebra

Map Algebra does not rely on a raster data structure, but is typically conceptualized as operating on a set of arrays of pixels with the same origin and orientation. The local operations in Map Algebra (Tomlin 1989) apply an unary operation (an operation with a single argument) to a single array, by applying it to each cell, or apply a binary operation to two arrays, by applying the operation to corresponding pixels from each array.

The operations in Tomlin's Map Algebra are independent of the data structure and the particulars of the implementation. They can be applied to rasters stored as a full array, run length encoding or as a quadtree. The different storage methods influence performance, but do not change the result. A rewriting of the Map Algebra using a functional language with higher order functions can bring two advantages:

- Potential for optimization: multiple operations can be executed together using the rules for combination given above. The combination of operations in a single pass over the data can greatly speed up performance as the time consuming access to the data is done only once and thus much disk access (or access to the data over the net) is saved.
- Generalization of the operations to work uniformly over raster and vector data in different data structures and to formally analyze the differences between vector and raster operations in the results and the error propagation.

5.1 Separation of Data Structure and Processing of Elements

To demonstrate the separation of treatment of data elements and traversal of data structure, the calculation of the area of a region stored in a quadtree is shown. An example using run length encoding works along the same lines, but cannot be shown due to the space limitations.

Quadtrees (Samet 1989b) are based on the principle of a 4-way branching tree data structure. It is customary to interpret a quadtree structure as a representation of space, in which the leaf nodes are pixels in a square array (Figure 2). Pixels of higher level represent four times the area of the pixel one level lower.

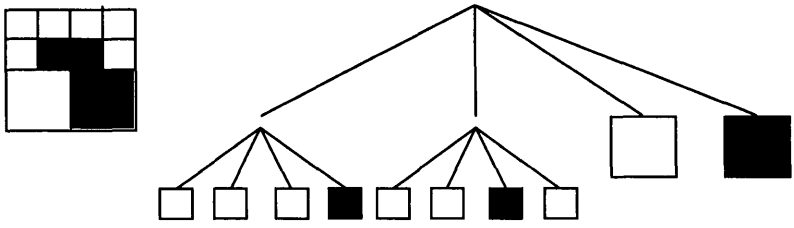

Figure 2 - Example of a quadtree

Recursively, a quadtree is either a leaf (`QL`) or it is a tree with four quadtrees. To keep the code simple, the size of the pixel is included with each leaf of the quadtree; this can be left out in optimized code.

```
data QuadLeaf p = QuadLeaf Int p -- the leaf size and the pixel
value
data Quad p = Q (Quad p)(Quad p)(Quad p)(Quad p) | QL p
```

Code to compute the area uses the corresponding operations for the pixels and corrects for the size of the area.

```
area (Pixel Inside) = 1
area (Pixel Outside) = 0
area (QuadLeaf j p) = j * (area p)

fold f s (QL p) = f p s
fold f s (Q2 q1 q2 q3 q4) = fold f (fold f (fold f (fold f s q1)
q2) q3) q4
area x = fold ((+).area) 0 x     -- x is of type Quad
```

5.2 Operations with two Arguments

The standard `map` function maps a function with a single argument. For the zonal operations of the Map Algebra, functions with two arguments must be mapped. An higher order function `map2` and `fold2` must be defined.

```
map2 :: (a -> b -> c) -> p a -> p b -> p c
map2 f (Empty) x = Empty
map2 f x Empty = Empty
map2 f (List x xs) (Element y ys) = Element (f x y) (map2 f xs ys)
```

With this the corresponding values from two lists can be added or subtracted. Assume you have a list with the population per census block and with the population under 20. The list of the population count (age > 20) is simply

```
map2 (-) totPopulation children
```

The combination of two quadtree or run length encodings is more involved: there are cases when one of the two structures must be expanded to meet the detail level of the other. In order to write this function only once to be used for `map2` and `fold2`, it is factored out in a function `zip`, which takes two data structures as arguments and makes a data structure with pairs of the values found in both structures. `Map2` and `fold2` then use the regular `map` and `fold`, and the function passed as an argument is applied to the two paired values.

```
zip (QL p) (QL q) = QL (p,q)
zip (Q p1 p2 p3 p4) (QL q) = Q (zip p1 q') (zip p2 q') (zip p3 q')
(zip p4 q') where q'= pushdown q
zip (QL p) (Q q1 q2 q3 q4) = Q (zip p' q1) (zip p' q2) (zip p' q3)
(zip p' q4) where p'= pushdown p
zip (Q p1 p2 p3 p4) (Q q1 q2 q3 q4) = Q (zip p1 q1) (zip p2 q2)
(zip p3 q3) (zip p4 q4)
map2 f p q = map f' (zip p q) where f' (a,b) = f a b
fold2 f s p q = fold f' s (zip p q) where f' (a,b) = f a b
```

With these operations, all the focal operations from Map Algebra can be written and applied to arbitrary data structures and feature data types. A choice of functions can be provided in a single program and polymorphism will select the appropriate operations to traverse the data structure and the operation to suit the data type of the feature.

6 Application to Computer Cartography

The examples so far were low level operations in a GIS, very close to the details of the implementation. In this section, I show that higher order functions are also very powerful tools to understand complex, large systems at the highest level of abstraction:

The classical computer graphics program consists of a list of data objects and a visualization loop, which applies the visualization transformation to each object (Foley and Dam 1982; Newman and Sproul 1979) and puts it on the screen. The visualization loop applies a series of transformations to the objects: perspective projection from 3D object coordinates to the 2D coordinates on the screen, clipping of parts which are outside of the viewing area etc. Other transformations apply to the lines or the symbols and produce the desired line style, according to the map legend selected.

These transformations can be written as functions applicable to objects. If the objects to display are in the `listOfFeatures`, the total operation is

```
map display.(scale 50).(clip xmin ymin xmax ymax).(symbolization
symboltable) listOfFeatures
```

This can be expanded to a query language for cartographic application. Simplifying the problem, one can start with a query language which has two query inputs - the selection criteria for the objects to display and the map legend to be used. This operation can be combined from a `filter` operation - a second order function - and `map` (as defined above):

```
filter criteria (Empty) = Empty
filter criteria (Element a as) = if criteria a then Element a
(filter as)                                  else filter as
query legend criteria database = (map (display legend)) . (filter
criteria) db
```

7 Conclusions

This paper introduces higher order functions, i.e. functions which take functions as arguments, in the GIS literature. Such functions are well known in mathematics, but

were not explicitly used for formalization of spatial information theory so far. First order languages are well suited for the analyses of static relations between objects, but they fail when dynamic behavior must be described. GIS must increasingly deal with dynamic objects and higher order functions are therefore necessary, but the same functionality is necessary when describing the behavior of complex, dynamic software systems, like GIS.

As a first example for the importance of higher order functions, this paper demonstrates how higher order functions allow to separate the part of operations specific to the data structure from the code of the operations which is specific to the data type stored. GIS are large data collections and must use complex spatial data structures. It is beneficial to separate the code which traverses the data structure from the code which operates on the feature data.

This is shown using a modern functional programming language and applied to Map Algebra (Tomlin 1989). The examples given are actual code as it can be executed. It shows convincingly the elegance and power of higher order functions. With this tool, the overall architecture of complex systems, e.g., Map Algebra or a cartographic query language, can be described in a single executable system without using *ad hoc* tricks. More examples can be found in (Frank 1996b; Frank 1996c)

The examples given here make clear that it is not sufficient to add a few higher order functions as fixed functions to a programming language, but that the full capability of writing new higher order functions is required. It is further necessary to allow data types with parameters, e.g., *List of Integer* must be differentiated from *List of Char* and a generic *List of x* or even *f of x* (*f* and *x* being type variables) must be possible.

References

Egenhofer, M. J., and J. R. Herring. 1991. High-level spatial data structures for GIS. In *Geographic Information Systems: Principles and Applications*, edited by D. Maguire, D. Rhind and M. Goodchild: Longman Publishing Co.

Ellis, M. A., and B. Stroustrup. 1990. *The Annotated C++ Reference Manual.* Reading, Mass.: Addison-Wesley.

Foley, J. D., and A. van Dam. 1982. *Fundamentals of Interactive Computer Graphics, Systems Programming Series.* Reading MA: Addison-Wesley Publ. Co.

Frank, A. U. 1996a. Qualitative Spatial Reasoning: Cardinal Directions as an Example. *International Journal for Geographic Information Systems* 10 (2).

Frank, A. U. 1996b. Hierarchical Spatial Reasoning: Internal Report. Dept. of Geoinformation, Technical University Vienna.

Frank, A. U. 1996c. Using Hierarchical Spatial Data Structures for Hierarchical Spatial Reasoning: Internal Report. Dept. of Geoinformation, Technical University Vienna.

Frank, A. U. , and I. Campari, eds. 1993. *Spatial Information Theory: Theoretical Basis for GIS.* Edited by G. Goos and J. Hartmanis. 1 vols. Vol. 716, *Lecture Notes in Computer Science.* Heidelberg-Berlin: Springer Verlag.

Hudak, P., et al. 1992. Report on the functional programming language Haskell, Version 1.2. *SIGPLAN Notices* 27.

Jensen, K., and N. Wirth. 1975. *PASCAL User Manual and Report.* Second Edition. Berlin-Heidelberg: Springer-Verlag.

Jones, M. P. 1991. An Introduction to Gofer: Yale University.

Milner, R. 1978. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences* 17:348-375.

Newman, W. M., and R. F. Sproul. 1979. *Principles of Interactive Computer Graphics.* New York: McGraw Hill.

Pigot, S. 1992. A Topological Model for a 3D Spatial Information System. In Proceedings of 5th International Symposium on Spatial Data Handling, at Charleston.

Samet, H. 1989a. *Applications of Spatial Data Structures. Computer Graphics, Image Processing and GIS.* Reading, MA: Addison-Wesley Publishing Co.

Samet, H. 1989b. *The Design and Analysis of Spatial Data Structures.* Reading, MA: Addison-Wesley Publishing Co.

Stroustrup, B. 1986. *The C++ Programming Language.* reprinted with corrections July 1987. Reading MA: Addison-Wesley Publishing Co.

Tomlin, C. D. 1983a. Digital Cartographic Modeling Techniques in Environmental Planning. Ph.D. thesis, Yale University.

Tomlin, C. D. 1983b. A Map Algebra. In Proceedings of Harvard Computer Graphics Conference, at Cambridge, Mass.

Tomlin, C. D. 1989. *Geographic Information System and Cartographic Modeling.* New York: Prentice Hall.

Worboys, M. 1992. A Model for Spatio-Temporal Information. In Proceedings of 5th International Symposium on Spatial Data Handling, at Charleston.