One Step up the Abstraction Ladder: Combining Algebras - From Functional Pieces to a Whole

Andrew U. Frank

Department of Geoinformation
Technical University Vienna
Gusshausstr. 27-29, A-1040 Vienna, Austria
frank@geoinfo,tuwien.ac.at

Abstract. A fundamental scientific question today is how to construct complex systems from simple parts. Science today seems mostly to analyze limited pieces of the puzzle; the combination of these pieces to form a whole is left for later or others. The lack of efficient methods to deal with the combination problem is likely the main reason. How to combine individual results is a dominant question in cognitive science or geography, where phenomena are studied from individuals and at different scales, but the results cannot be brought together. This paper proposes to use parameterized algebras much the same way that we use functional abstraction (procedures in programming languages) to create abstract building blocks which can be combined later. Algebras group operations (which are functional abstractions) and can be combined to construct more complex algebras. Algebras operate therefore at a higher level of abstraction. A table shows the parallels between procedural abstraction and the abstraction by parameterized algebras. This paper shows how algebras can be combined to form more complex pieces and compares the steps to the combination of procedures in programming. The novel contribution is to parameterize algebras and make them thus ready for reuse. The method is first explained with the familiar construction of vector space and then applied to a larger example, namely the description of geometric operations for GIS, as proposed in the current draft standard document ISO 15046 Part 7: Spatial Schema. It is shown how operations can be grouped, reused, and combined, and useful larger systems built from the pieces. The paper compares the method to combine algebras – which are independent of an implementation – with the current use of object-orientation in programming languages (and in the UML notation often used for specification). The widely used 'structural' (or subset) polymorphism is justified by implementation considerations, but not appropriate for theory development and abstract specifications for standardization. Parametric polymorphism used for algebras avoids the contravariance of function types (which its semantically confusing consequences). Algebraic methods relate cleanly to the mathematical category theory and the method translates directly to modern functional programming or Java.

Keywords. Spatial Algebras, Spatial Data Models, Category Theory, Parameterization of Algebras

1 Introduction

A fundamental scientific question today is how to construct complex systems from simple parts. Science is – at least in the form typically reported in the Geographic Information Systems literature – very good at analyzing individual pieces of the puzzle. The combination of these pieces to form a whole is left as “a simple exercise for the reader” – and everybody knows from experience, that these simple exercises are not easy at all. The simple functions in a large system – and GIS are large systems – are all relatively easy to define (with some fudging at the seams) and formalization methods are often available. When the pieces are put together, unexpected interactions occur, questions become posed which do not have an answer within the previously studied limited context of the individual pieces and we see that the pieces do not fit together. This applies to software engineering and standardization as much as to other areas of science; it is a dominant question in cognitive science, where phenomena are studied individually and at different scales; we seem to know a lot of individual details, but we have difficulties to fit these pieces together. It is an important question in geography, where different disciplinary approaches are used but all the results must be brought together applied to the same geographic location.

Functional abstraction is widely used. Named functions are crucial in mathematics and procedures are the building blocks of all programs. Procedures draw their power from their parameterization, which allows to formulate general rules which can be used in different circumstances. A function *square x* can be used in various contexts with different values for *x*. The same concept can be applied at a higher level of abstraction. Algebras consist of several functions that can be named and have parameters. The parameters do not stand for concrete values as in procedures, but – a step more abstract – for types. They can be combined and the type parameters duly replaced by the actual parameters, much the same way as in the application of functions. Table 1 demonstrates the parallels between the two concepts.

Table 1. Comparison of procedural and algebraic abstraction.

	Procedural Abstraction	**Algebraic Abstraction**
object	operation (procedure, function, method)	Algebra (abstract data types)
components	a sequence of operations	several operations operating on the same sorts
applied to	values	sorts (types)
use	call	instantiation
formal parameters	formal parameters for values	formal parameters for sorts
actual parameters	values	representable data types
combination	call of procedure within another abstraction	use as a sub-algebra within another algebra

Category theory [2, 3, 14, 23] abstracts from individual values to sets of values (types, domains). Algebras group operations which are applied to the same data types. Axioms in the algebra define the properties (behavior) of these operations. Algebras are naturally parameterized in the types of the arguments the operations in the algebra take. An algebra can be compared to a procedural abstraction: it has a name, a set of parameters, which stand for types, and a set of operations on these types. Parameterized algebras can be reused and combined through instantiation. Algebras are instantiated, when for each carrier a concrete data type is provided and actual implementation given for the operations; this is comparable to the call of a procedure or function (Table 1).

Terminology here is extremely confusing, as the different disciplinary traditions of algebra, category theory, theoretical computer science, programming languages (functional, object-oriented) use conflicting terminology. Glossing over justified differences, I decided to continue to use the best-known terms (mostly with a programming languages background). The notion *operation* will be used for function or method. *Type* will be used for what is otherwise called sort, set of values, or carrier.

Functions can be used to build more complex functions, where the parameters are replaced by the parameters from the encompassing functions, which will be eventually replaced by actual values when the program is executed. The same applies to algebras, where several (sub)algebras are instantiated and combined to form a larger algebra, which is ultimately instantiated with representable data types. The semantics of an individual algebra is given by the axioms for the operations and carried forward to the combined algebra.

This technique is generally usable; it can be applied to cognitive science problems or geography, where small (descriptive) models can be constructed as algebras, which are later combined. It can be used in software engineering and specification writing. After an introduction of the concept of algebra in Section 2, the method will be described with the construction of vector space from two different algebras in Section 3. Section 4 will then apply the method to structure the current proposal for a standardization of the geometric operations used for GIS and CAD. Section 5 briefly sketches the application to a problem from cognitive science, namely the definitions of relative spatial reference systems in natural languages. Section 6 discusses generalization and polymorphism and Section 7 presents conclusions.

2 Algebra

Algebras capture the coordinated behavior of operations that are applied to the same object. Numbers are numbers not because one can add them, but because the operations of addition, subtraction, comparison, etc. work in a specific pattern (in algebra called ‘structure’). We have advocated the uses of multi-sorted algebras for the specification of GIS at least since 1986 [10, 11]. A recent paper has presented a family of geometric data models based on the theory of many sorted algebra [18]. Algebras should be built for maximal reuse, maximal cohesion, and minimal interdependence. These are the same requirements – now on a structurally higher level – demanded for procedures and object classes in programming [24].

An algebra consists of three parts: a type and a set of operations, the behavior of which are defined with axioms. The most familiar example of an algebra is numbers, for example, the algebra of integers (technically an Abelian Group). The operations are addition (+) and subtraction (-). There is a particular number zero (0), which has special properties. The notation is widely used in the literature [13, 25]; it follows in particular [8], small changes stress the similarity with the syntax of procedures in languages like Pascal. After the keyword *Algebra* follows the name of the algebra and the type parameters in parenthesis. The operations and constants are listed after the keyword *Operations*. For each operation the name of the operation followed by '::' and the list of argument types and the return type (the signature) is given. After the keyword *Axioms*, the axioms describing the behavior of the operations are listed ('—' indicates a comment, e.g., a name of an axiom). It is possible to abbreviate this format and give only the algebra name with the type parameters and then a list of operations included – this short format will be used in the examples given later in the paper. The following example for the familiar algebra of natural numbers should help to understand the syntax of the full format:

```
Algebra AbelianGroup (number)
Operations:   +, - :: number -> number -> number
               Negate::number -> number
               0 :: number
Axioms: a + b = b + a                               -- commutative law
              (a+b)+c= a+(b+c)= a+b+c               -- associative law
              0 + a = a + 0 = a                     -- existence of identity
              a + (negate a) = 0                    -- existence of inverse
              a – b = a + (negate b)                -- definition of subtraction
```

Algebras can be used to describe other behavior than numbers, for example, the properties of a stack. In such cases, more than one type is used and the algebra is called multi-sorted or heterogeneous [5].

```
Algebra Stack (stack of a, a)
Operations:   push :: a -> stack of a -> stack of a   -- constructor
              empty :: stack of a                     -- constructor
              pop :: stack of a -> stack of a         -- observer
              top :: stack of a -> a                  -- observer
Axioms:       top (push a s) = a
              pop (push a s) = s
              top (empty) = error
              pop (empty) = error
```

An algebra does not describe what the objects are, only how they behave. Algebras give specifications and do not determine the implementation. Many different objects can behave according to the same rules; within an algebra, one cannot differentiate between them. The current object-oriented debate, which is linked to the implementation of programming languages, equates objects with operations and data representation. The definition of operations (which forms an algebra) is merged with a description of a representation of the types. This forsakes the parameterization of the algebra (the free parameter is immediately bound to a particular representation) and destroys the potential for reuse of the algebra for other representations.

2.1 Type Parameters

The type names in an algebra should be read similar to the parameter names in a function: they are ‘formals’ standing for data types, the same way that a formal parameter stands for a later supplied concrete value. The definitions above do not state how the numbers in the algebra of Abelian Groups should be represented: binary numbers are as useful as numbers formed from Arabic or Roman numerals. The power of the algebraic abstraction is exactly this abstraction from a concrete realization (implementation).

The type names are only valid within the algebra and have no meaning outside the scope of the algebra description; this is similar to the use of parameter names in procedure descriptions in programming: the a in f (x, a, b) = … and in g (x, b, a) = … are not related. Parameter names are often selected to suggest an interpretation, but this is just a hint for the reader, not a formal property. When algebras are combined, the parameters are replaced; much the same way that parameters are replaced with the actual values when a function is called.

Types can be parameterized: The stack constructed before is a stack of integers, of plates, of books, depending on what is pushed on it. This dependency between carriers is expressed as a parameterization of the type. The example uses a parameterized type, *stack of a*, where the type *stack* relates to another type *a*. This is similar to the template notion in C++ [22], which is not fully integrated into the language and therefore difficult to use. Using the pushout construction from category theory, the combination of algebras is mathematically well defined [8]. The type inference rules for multiple parameters were presented by [16].

2.2 Operations

Without loss of generality but immense gain in notational clarity, operations are restricted to (pure) functions [1]. Functions have input parameters, which are not changed, and a single result (which can be a composition of several values); procedures that change the parameters can be rewritten to conform to this format. Special named constants (e.g., zero) are understood as functions without input and thus a constant result. In this categorical framework everything is a function.

The list of the types of the input parameters followed by the type of the result is called the *signature of the operation*. Operation names are assumed to be unique within the context of discussion (which avoids the notationally confusing problem of renaming operations).

Operations can be separated in constructors, observers, and derived operations. *Derived operations* serve as a convenience and are just abbreviations for combinations of other operations in the algebra. For example, the operation subtraction (-) can be defined in terms of negate: *a – b = a + (negate b). Constructors* are the operations that are used to construct all the values in the carrier; their result is always an object of the (primary) carrier of the algebra. *Observers* take objects from the primary carrier and relate them to other (probably already defined) carriers. The observers must be sufficient to differentiate between all values in the carrier.

2.3 Axioms

Axioms describe the properties of the operations (the *behavior* of the operations). Axioms written in a categorical (point-free) fashion describe operations independent of actual values (typically the function composition operation (.) and some constant functions (e.g., id, the function that does nothing: f (x) = x) from category theory are used.

negate . negate = id

Variables standing for variables in axioms are automatically prefixed with an 'for all' quantor and have the types described by the signatures. Axioms often state the existence of a value for which a property is asserted; such axioms are called non-constructive. If the axioms are written in a restricted equational logic [8], it is often possible to translate the axioms to program code and execute them.

If a formal language is used to describe the axioms, the existence of a definition for operations and the consistent use of types can be checked. If the language is executable, then the defined semantics may be compared with the intended one while observing the result of operations. It is difficult to determine if a set of axioms is sufficient; for practical purposes, an axiom system, which describes each constructor application in terms of observers, is usually sufficient. In the algebra of stack, two observers are applied to two constructors, for a total of four axioms.

3 Combination of Algebras to Form New Algebras

To demonstrate the concept of combination of algebras, it is applied twice: once to extend a given algebra with new operations and second, combining two algebras to construct the algebra of the vector space, as explained in most text books of a course in algebra. The underlying theory can be found in [8, 19].

3.1 Extension of an Algebra

The algebra for Abelian (commutative) groups is extended with multiplication:

```
Algebra Fields (num)
Use AbelianGroup (num)       -- the parameter 'number' in the definition is replaced
                             with 'num'
Operations: *, / reciproc, 1
Axioms:      a * b = b * a                          --commutativity for multiplication
             a * (b * c) = (a * b) * c = a * b * c  --associativity for multiplication
             a * 1 = 1 * a = a                      --unity for multiplication
             a * (reciproc a) = 1                   --existence of multiplicative inverse
             a * (b + c) = a * b + a * c            --distributive laws
             (b + c) * a = b * a + c * a
             a/b = a * (reciproc b)                 --derivation of division
```

3.2 Combination of Two Algebras to Form Vector Space

A vector space is constructed from an Abelian group of vectors, i.e., a set of objects, for which a commutative addition (the regular vector addition) and a zero element (the null vector) are defined. This Abelian group of vectors is combined with a field of numbers, called scalars, in a new operation scalar multiplication (*\$), which has a signature of *num->vec->vec* (and is therefore different from regular multiplication, where all types are the same: *a -> a -> a*). The following definitions come straight from a standard text on vectors [21].

```
Algebra VectorSpace (vec, scalar)
Use AbelianGroup (vec), Fields (num)
Operations:  *$ :: num -> vec -> vec
Axioms:      for all a, b, c... elem scalar, x, y, z… elem of vec
             a *$ (x + y) = (a *$ x)+ (a *$ y)
             (a + b) *$ x = (a *$ x)+ (b *$ x)
             a *$ (b*$ x) = (a * b) *$ x
             l *$ x = x
```

In this algebra, additional operations can be defined: e.g., the inner (dot) product of two vectors, vectorProduct. These have interesting geometric interpretations, which allow the derivation of other operations, e.g., a test for orthogonality: *orthogonal a b = (dotProd a b) ==0* (Comment on notation: ‘=’ stands for definitional equal, whereas ‘==’ is used to describe a logical test for equality).

3.3 ‘Use’ Interpreted as Substitution

The ‘use clause’ in the definition of an extended or combination algebra definition means that the original definition can be substituted, as in high school algebra (“substituting equals for equals”), including operation signatures and axioms, with the formal parameters replaced by the actual parameters given. This is correct because the functions do not have side effects. As we have demanded that operation names are unique, no renaming of operations is necessary.

4 Case Study: The Specification of Geometric Operations

Writing specifications for the geometric operations in GIS or CAD systems is notoriously difficult. Formalization of geometry is a mathematically very complex problem requiring contributions from several fields of mathematics (topology, algebra, etc.) that are difficult to combine. Therefore it presents a realistic, large problem to apply the method. Current efforts in the Open GIS Consortium [7] to write specifications for a wide variety of GIS operations and the pending proposal for an ISO standard to extend SQL with operations on geometric objects demonstrates the magnitude of the problem. The ISO draft document for geometric operations runs for nearly 100 pages. The effects of the operations are described in natural language, open to interpretation. Implementation will be difficult and differences in interpretation will hinder interoperability.

The tools currently available for standard design and writing are not sufficient: the formalization of the signatures of the operations using STEP/EXPRESS [15] or the comparable UML methods [6] cover only a small set of the normative content of a standard. An axiomatic method could be used to define behavior of operations [12], but it has not been demonstrated how algebras could be combined to construct large systems with multiple subsets, defined for different applications.

The standard should define useful subsets that can be implemented. From an application point of view, subsets must be possible in three different directions:

1. Embedding space dimension: application to geometry expressed with points in two-dimensional space and points in three-dimensional space should be covered.
2. Object dimension: applications which contain objects of 0, 1, 2 or 3 dimensions.
3. Interpolation type: applications that use only linear interpolation, or other, that use more complex methods for objects of one or two dimensions (choices not dealt with in standard document).

There are subtle dependencies between these choices.

- The object dimension must be smaller (or equal) to the dimension of the embedding space.
- Interpolation methods are available for objects of a dimension smaller than the dimension of the embedding space (objects with codimension 1: codimension = dimension of embedding space minus dimension of object)

For this purpose programming languages provide no support, and tools like UML use a diagrammatic language. The combination of algebras has the power to cover such cases within an easy formalism that has a clean mathematical interpretation.

4.1 Geometric Operations in GIS as Algebras

The "packages" from the standard document are quickly translated to algebras. Each package becomes an algebra, operations signatures are expressed using the sorts in the algebra (axioms given in the standard could be preserved). Unlike programming languages, all operations are written as functions with all parameters listed. The parameters are all 'read-only' and only the result parameter (the last in the list) is produced. Some of the major packages, which consist of several sets of "stereotypes", have been subdivided to gain flexibility for combinations (to save space, only operation names in accordance with the proposed ISO standard are given and the signatures are left out).

Algebra GeometricObjects (geomObj, directPos, length, int)
    Operations: *isSimple, isClosed, equal, intersect, union, intersection, difference, symmDifference, distance, dimension*
Algebra Boundaries (geomObjDimN, geomObjDim(N-1))
    Operation: *boundary*
Algebra Distances (geomObjM, geomObjN, lengthValue)
    Operation: *distance*
Algebra Points (point, directPos, vector)
    Operations: *position, point, bearing*
Algebra Curves (curve, directPos, vector, length)

Operations: startPoint, endPoint, tangent, parameterization, curveLength, curvePointToPointLength

Algebra Segments (segment, directPos)

Operation: segment

Algebra CurveConstructors (curve, segment)

Operation: curveMake

Algebra Surfaces (surface, lengthValue, areaValue, directPos)

Operations: perimeter, area

Algebra Patches (patch, directPos)

Operation: segment

Algebra SurfaceConstructors (surface, patch)

Operation: curveMake

Algebra Solids (solid, volumeValue, areaValue)

Operations: volume, area

Support:

Algebra VecSpace (Vector, length)

Operations: dotProd, orthogonal, …

Algebra Vec2 (Vec2, length)

Operations: vec2, unit1, unit2,x2, y2

Algebra Vec3 (Vec3, length)

Operations: vec3, unit1, unit2, unit3, x3, y3, z3

Algebra DirecPos (directPos, Vector)

Operations: Vector, directPos

4.2 Useful Combinations

The minimal implementations must either contain the vector algebra for two-dimensional or three-dimensional space. This can be expressed as two algebras, consisting of a number of the support classes:

Algebra Space2D (vec, length, directPos)
Subalgebras: Vec2 (vec, length), VecSpace (vec, directPos), DirectPos (directPos, vec)
Algebra Space3D (vec, length, DirectPos)
Subalgebras: Vec3 (vec, length), VecSpace (vec, directPos), DirectPos (directPos, vec)

Note: The statement that for the vector in these algebras the operations of the algebra Vec2s or Vec3s must be available dictates, for example, that an operation *vecProd:: vec -> vec -> vec* is possible in Space3D but not in Space2D.

Further algebras can be added, as their operations are required for the application area. A complete system to deal with Graphs, consisting of straight lines embedded in three-dimensional space is, for example:

Algebra GraphIn3D	*(geomObj1, geomObj0, segment, directPos, vec, lengthValue)*
Subalgebras:	*VecSpace3D (vec, length, directPos), Graph (geomObj1, geomObj0, segment, directPos, vec, lengthValue)*

Whereas a graph with curved lines between nodes in 2D requires segments and is described as:

Algebra Graph	*(geomObj1, geomObj0, segment, directPos, Vector, lengthValue)*
Subalgebras:	*VecSpace2D (vec, length, directPos), Graph (geomObj1, geomObj0, segment, directPos, vec, lengthValue), CurveConstructor (geomObj1, segment), Curves (segment, directPos, vec, length), Segments (segment, directPos)*

5 Case: Modeling Cognizant Agents

The example sketched here briefly considers modeling agents observing distance and directions in a simple world of point-like objects and translates their observations on a continuous scale into discrete expression of relative position in the 'world' around them (outputs like "from Peter's perspective, the ball is in front of the chair'). The focus of the study [9] was in the formalization of the relative spatial expressions in natural language and the corresponding reference frames [17]. In particular, formal definitions for deictic, absolute, relative, etc. reference frames were attempted, for which no unequivocal definitions can be found in the literature.

Coding such models in any language requires a few pages of code and the overview of the relations between the procedures is quickly lost. Insight into the interaction between the modules is important for the assessment of the cognitive adequacy of the proposed formalization. With combination of algebras, the interaction of the modules can be expressed in a few lines. (The following description is directly extracted from code which demonstrates the system).

In a first step, an algebra for rotation and translation in two-dimensional space and an algebra to discretize continuous values to a qualitative scale were defined. These were then combined to a model, which allowed comparing the expected error between different types of qualitative direction description.

Algebra Translation and Rotation (obj, angle, vec)
Use VecSpace (vec)
    Operations: *orientation, rotate, rotateTo, translate, translateTo, translateRotate, translateRotateTo*
Algebra Discretize (quantitativeVal, qualitativeVal)
    Operations: *discretize, undiscretize*

The world is modeled as a database where named objects with position in space can be placed. The agent is a special kind of object, which can observe its own orientation, see the other objects in the world to build its own set of knowledge, and describe this knowledge from different perspectives:

Algebra World (world of obj, obj, id, value)
    Operations: *putObj, getObj, thatHas, observer*
Algebra Named (obj, name)
    Operations: *putName, getName*
Algebra Positioned (obj, vec) Uses VecSpace
    Operations: *putAt, isAt*
Algebra Agent (agent, world, name, text)
    Operations: *putOwnOrientation, see, describeWorld, egocentric, absoluteAllocentric, intrinsic*

These algebras can be combined to a complex world in which multiple agents exist and can observe:

Algebra WorldwithAgents (world of things, things, id, name, vec, text)
    Union things = furniture | agent   *a thing is either a piece of furniture or an agent*
Use world (world of things, things, id, name), Agent (agent, world of things, name, text), Position (things, vec), Named (things, name)

6 Generalization and Polymorphism

The cleanliness of the combination of algebras is due to the concentration on the intention and complete abstraction from implementation. In particular, generalization of operations which can be applied to many types of objects is achieved through parameterization. This is different from the current object-oriented debate, which is mostly based on properties of current object-oriented languages and has introduced the concept of sub-type and inheritance. These concepts are more linked to implementation and are not appropriate for the specification and the design.

The UML notation uses 'structural' generalization, which defines types and subtypes for these types. For example, Point is a subtype of GeometricObject. The subtype is said to inherit the operations from the supertype, i.e., an operation 'distance' which is applicable to GeometricObject is, by inheritance, applicable to Point. The problem with this subtype relation are the contravariant relations between the parameters and the result type. An operation op1 :: a -> b can be used for a type a' (a' being a subtype of a) yielding a result of b' (b' being a supertype, not a subtype, of b). For details of this formal consequence of subtyping see [1]. Different methods to contain this 'contravariance' are used in current object-oriented languages; UML requires repeated definitions. They are necessary whenever implementation (data representation) dominates. Java separates representation oriented classes (with single inheritance, which is structural) from behavior oriented 'interfaces'.

Algebraic combination uses parametric polymorphism. The use of an operation for different types is defined by the type parameters which are instantiated for the type parameters listed in their definition. Parametric polymorphism in a functional programming language are natural transformations [4] and avoid the contravariance problem.

7 Conclusions

Procedural abstraction, the naming of methods to perform some action, is a powerful abstraction method, not only used in programming but throughout science. The same concept of encapsulation can be applied to conceptually related sets of methods to be performed on some objects, yielding the mathematical concept of algebra. The semantics of the operations in an algebra are given with axioms. Algebras can be parameterized similarly to the parameters of procedures, and algebras can be reused and combined to form more complex systems. Combination of parameterized algebras uses a simple substitution semantics: the combined algebra contains all the operations and axioms of the constituting algebras, with the formal parameters duly replaced with the actual ones.

Algebras capture the structure of the behavior of a set of related operations. Often results of scientific research are expressed in 'laws' which relate operations. Parameterized algebras permit to combine such results. It can be applied in science to combine the individual results of separate research efforts. It is a crucial method for multi-disciplinary sciences, like cognitive science, where results of individual disciplinary contributions must be integrated. It is useful for sciences like geography, where phenomena are studied at various scales, as it provides a framework in which different effects can be brought together. As the concept is simple, it can be used in informal arguments as well as in a formal description.

This notion captures very closely the notion of object-orientation seen from a specification point of view. The current confusion with object-orientation and in particular inheritance of operations is based on the implementation (data structure centered) viewpoint present in commercial programming languages. The algebraic position has become possible through research in parameterization of algebras [8, 19] and research in parameterized type inference systems [16]. Only lately the programming languages have become available, which demonstrates the practical usability of the concept for software engineering [20]. Parameterization is a much less confusing concept than the often-discussed inheritance based on subtyping.

Acknowledgements

Werner Kuhn’s comments contributed to focus the presentation. Funding from the European Commission under the ESPRIT program for the projects GIPSIE and COMMUTER supported some of the work.

References

1. Abadi, M. and L. Cardelli, *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, New York (1996).
2. Asperti, A. and G. Longo, *Categories, Types and Structures - An Introduction to Category Theory for the Working Computer Scientist*. The MIT Press, Cambridge, MA (1991).
3. Barr, M. and C. Wells, *Category Theory for Computing Science*. Prentice Hall, London (1990).
4. Bird, R. and O. de Moore, *Algebra of Programming*. Prentice Hall, London (1997)
5. Birkhoff, G. and J.D. Lipson, *Heterogeneous Algebras.* Journal of Combinatorial Theory (1970) 8: 115-133.
6. Booch, G., J. Rumbaugh, and I. Jacobson, *Unified Modeling Language Semantics and Notation Guide 1.0*. Rational Software Corporation, San Jose, CA (1997).
7. Buehler, K. and L. McKee (eds), *The OpenGIS Guide - An Introduction to Interoperable Geoprocessing*. The OGIS Project Technical Committee of the Open GIS Consortium, Wayland, MA (1996).
8. Ehrich, H.-D., M. Gogolla, and U.W. Lipeck, *Algebraische Spezifikation abstrakter Datentypen*. Leitfäden und Monographien der Informatik, H.-J. Appelrath, *et al.* (eds). B.G. Teubner, Stuttgart (1989).
9. Frank, A.U., *Formal Models for Cognition - Taxonomy of Spatial Location Description and Frames of Reference*. In *Spatial Cognition - An Interdisciplinary Approach to Representing and Processing Spatial Knowledge*, C. Freksa, C. Habel, and K.F. Wender (eds). Springer-Verlag, Berlin (1998) 293-312.
10. Frank, A.U. and W. Kuhn. *Cell Graph: A Provable Correct Method for the Storage of Geometry*. In *Second International Symposium on Spatial Data Handling*, Seattle, WA, 1986, 411-436.
11. Frank, A.U. and W. Kuhn, *A Specification Language for Interoperable GIS*. In *Interoperating Geographic Information Systems*, M.F. Goodchild, *et al.* (eds). Kluwer, Norwell, MA (1998).

12. Frank, A.U. and W. Kuhn, *Specifying Open GIS with Functional Languages*. In *Advances in Spatial Databases (4th Int. Symposium on Large Spatial Databases, SSD'95, in Portland, USA)*, M.J. Egenhofer and J.R. Herring, (eds). 1995, Springer-Verlag, 184-195.
13. Guttag, J.V. and J.J. Horning, *Larch: Languages and Tools for Formal Specification*. Springer-Verlag (1993).
14. Herring, J., M.J. Egenhofer, and A.U. Frank. *Using Category Theory to Model GIS Applications*. In *4th International Symposium on Spatial Data Handling, SDH'90*, Zurich, Switzerland, 1990, 820-829.
15. ISO, *The EXPRESS Language Reference Manual*, ISO TC 184, Technical Report ISO/DIS 10303-11 (1992).
16. Jones, M.P., *Qualified Types: Theory and Practice*. Ph.D. Thesis, Programming Research Group, Oxford University. Cambridge University Press (1994).
17. Levinson, S.C., *Frames of Reference and Molyneux's Question: Crosslinguistic Evidence*. In *Language and Space*, P. Bloom, *et al.* (eds), MIT Press, Cambridge, MA. (1996) 109-170.
18. Lin, F.-T., *Many Sorted Algebraic Data Models for GIS*. IJGIS (1998) 12(8) 765-788.
19. Loeckx, J., H.-D. Ehrich, and M. Wolf, *Specification of Abstract Data Types*. Wiley, Teubner (1996).
20. Peterson, J., *et al.*, *Report on the functional programming language Haskell, Version 1.3*. In *http://haskell.cs.yale.edu/haskell-report/haskell-report.html - Research Report YALEU/-DCS/RR-1106*. Yale University (1996).
21. Reinhardt, F. and H. Soeder, *dtv-Atlas zur Mathematik: Grundlagen, Algebra und Geometrie (Band 1)*. dtv, Muenchen (1991).
22. Stroustrup, B., *The C++ Programming Language*. 2nd edn. Addison-Wesley, Reading, MA (1991).
23. Walters, R.F.C., *Categories and Computer Science*. Cambridge Computer Science Texts, Vol. 1. Carslaw Publications, Cambridge, UK (1991).
24. Wirth, N., *Algorithms + Data Structures = Programs*. Prentice Hall, Englewood Cliffs, NJ (1976).
25. Yeh, R.T.-Y. and P.A.B. Ng, *Modern Software Engineering: Foundations and Current Perspectives*. Van Nostrand Reinhold, New York (1990).