Frank, Andrew U., and Werner Kuhn. "Specifying Open GIS with Functional Languages." In *Advances in Spatial Databases (4th International Symposium, Ssd'95 in Portland, Me)*, edited by Max J. Egenhofer and John R. Herring, 184-95: Springer-Verlag, 1995.

Andrew U. Frank
Werner Kuhn

Department of Geoinformation
Technical University Vienna
Gusshausstrasse 27-29/127
A-1040 Vienna (Austria)

Frank@geoinfo.tuwien.ac.at
Kuhn@geoinfo.tuwien.ac.at

ABSTRACT

The concept of Open GIS depends on precise definitions of data, operations and interfaces. This paper argues for the use of functional programming languages as specification and prototyping tools for Open GIS components. It shows how functional programming languages fulfill the key requirements for formal specification languages and allow for rapid prototyping in addition. So far, it has never been possible to integrate specification and prototyping in a single, easy to use environment. Most existing specification methods lack appropriate tools for checking and prototyping, while existing tools lack either sound semantics or usability or both.

The paper discusses the role of specifications in GIS, requirements for specification languages, and a survey of algebraic specifications as well as of functional languages. It then describes how functional languages can be used for writing and executing algebraic specifications. A brief example of a GIS data type specification in a functional language is presented, showing how specifications serve to describe differences in the semantics of GIS operations. We conclude that functional languages have the potential to achieve a breakthrough in the problem of specifying interfaces of interoperable components for Open GIS.

[1] Funding from Intergraph Corporation and from the Austrian Science Foundation is gratefully acknowledged.

[2] Frank, A.U. 1995. "Surveying Education for the Future". In *Geomatica*, **49 (3)**, pp: 273-282.

1 SPECIFICATIONS AND OPEN GIS

Specifications are essential for software quality and widely used in industry. For Geographic Information Systems (GIS), they are of special interest in the standardization of data models, transfer methods, Open GIS component interfaces, and database architectures. Practical specification methods are needed for the success of Open GIS architectures in which programs and data collections from different vendors need to cooperate at multiple levels of abstraction [Voisard, and Schweppe, to appear].

1.1 Specifications and Software Development

Specifications allow for a division of labor in software development. They serve as a contract between the software analyst, who understands the application problem, and the programmer, who is concerned with an optimal use of resources. They support producing the software and assessing its correctness.

CASE tools provide informal methods to design, manage, and communicate specifications for software. Positive results from largely informal software design methods in general have been reported [Head, 1994], but software practitioners criticize current tools for being too low level. A pointed observation is that CASE tools are nothing but glorified systems to draw diagrams and often do not scale up to large problems, where the amount of documentation becomes overwhelming.

Formal specifications, i.e., specifications written in a formal language with mathematically defined semantics, allow for formal checks and reasoning before programming starts. They can provide support for an automatic program verification [Guttag, Horning, and Wing, 1985]. However, such consistency checks are internal to the formal system and cannot ensure that the specifications capture the intentions of the designer or other real world requirements.

Rapid prototyping has been advocated to achieve programs that correspond not only to specifications but also to the actual user requirements. Prototyping reduces the danger that you get what you ordered, but not what you wanted. In current software design practice, specification and prototyping tools are often separated or only loosely coupled. If a language

allowed to write specifications and to execute what has been specified, it could serve as a combined specification and prototyping tool. This paper claims that functional programming languages, extended with recent research results, achieve this goal.

1.2 What is Special about Specification Needs for Spatial Data?

The need for formal specification languages is common to the whole software industry; problems with interoperability of tools from different vendors plague nearly every user of a computer system. However, the specification problem is more acute for GIS than for most other application areas:

- Economic use of spatial data is only possible if data can be used by many different users from different organizations. The need for a functioning market of spatial base data is larger and this market is growing faster than in most other areas of information technology. Consider as an indicator the offerings for data on CD-ROM: geographic data figure very prominently.
- The structure of geographic data is more complex than that of other data exchanged routinely. It is comparable to, but going beyond, that of CAD data, where similar problems are encountered.
- Sharing data transcends organization boundaries. The consumer usually pays for the data and expects them to be "usable", otherwise legal problems can ensue [Frank, 1992].

1.3 Specifications and the GIS Market

GIS are evolving into multi-vendor software environments, Open GIS, where heterogeneous components cooperate to solve complex spatial problems. Users buy software from different vendors to solve particular parts of their problem: they want to access the database system produced by one company from the mapping tool of another vendor or to assess the spatial dimension of marketing forecasts in a GIS. This movement toward interoperable, open environments will rapidly progress within and outside the GIS industry [O'Callaghan, 1993]. Specifications are the key to achieve this interoperability. They constitute a contract between providers and customers of services in an Open GIS.

Specifications of GIS services provide economical viability for the niche market vendor who can offer tools that are independent of the software environment of a client. If a vendor must provide a special version for any combination of client database system and GIS software, the business cannot operate profitably. GIS tools can only be provided for specific markets if they can be built on top of a general service layer.

Specifications are also essential to access the mass market with GIS software: Selling low cost software can only become profitable if GIS tools provide a standardized service (which can be used and understood independently of a specific vendor) and interact in a standardized way with databases, operating systems and other applications. Specifications are the enabling technique to ensure customer satisfaction with mass products, while also improving special applications that work within open environments.

The problem of data transfers can also be cast into the specification framework [Kuhn, 1994]. This helps to separate the issues of data representation (which are relatively easy to solve) from the critical issues of differences in semantics. The buyers in a "spatial data market" are interested in the information they can obtain from transferred data. They need tools to describe this information or to interpret existing descriptions. Defining this information content or semantics is precisely what a specification does.

The first SSD meeting pointed to the importance of meta data and formalisms to help potential users of spatial data find data in a distributed environment and to transfer them [Smith, and Frank, 1990]. Methods to describe meta data are missing today, current efforts for standardization still struggle with fundamentals of geometric data, and the standards are based on informal semantics. Future standards for Open GIS will need to use formal specifications for defining data semantics and for testing adherence of implementations to standards. Specifications as advocated here can formally define semantics at all levels of a data transfer.

2 REQUIREMENTS FOR SPECIFICATION LANGUAGES

Several properties of a specification language are desirable for its use in practice. This section summarizes only the most important ones: expressing semantics, ease of understanding, and rapid prototyping, i.e., a way to check whether the specified semantics are those intended.

2.1 Expressing Semantics

Specification languages must allow for expressing real situations in terms with exactly defined formal semantics. Defining semantics unambiguously and completely is so difficult that it presents one of the main motivations for formal specifications. Most specification methods produce documentation of which only a minimal amount can be checked automatically for well-formedness, completeness, and consistency. While the correctness of a specification with respect to what the designer had in mind can never be formally asserted, clean semantics of the underlying language and formal checking of the syntax make correctness much more likely.

Some data definition languages (e.g., EXPRESS [ISO, 1992]) allow to specify data types, but lack formal semantics. They describe static data types with attributes and relationships, omitting the specification of operations. However, a specification language based on types must have a method to associate data types with operations. Otherwise the concept of type remains vacuous. Section three describes algebraic specifications which achieve this association.

2.2 Ease of Understanding

Expressive power is a necessary condition for a specification language, but it does not guarantee its success. Specifications must be easy to write and read. Specification languages must help to master the complexity of real world applications and should offer abstractions which are easily understood. Too many specification approaches have stressed the formal aspects of the syntax, disregarding the interpretation of specifications by human readers (as opposed to syntax checkers).

Specifications often produce enormous amounts of documentation, and a common complaint for all specification methods is that they do not scale up. While they work well for

the small examples on which they have been developed and tried, they overwhelm the designers when applied to large, real world situations. Despite this complaint, any structured approach to specify a problem before programming is beneficial, nearly independently of the method and tools applied.

2.3 Rapid Prototyping

Organizing a large body of knowledge in a comprehensive fashion must bridge the gap between the cognitive and the formal. Due to the difficulty of writing and reading specifications, many errors are not detected before an implementation is completed. Rapid prototyping has been the industry's answer to this dilemma. It was motivated by the difficulty in the software design process to capture real world semantics. Prototyping is clearly preferable to testing of production code, as it can be done much earlier in the development process, where changes are relatively easy and cheap.

A specification language with a prototyping capability allows for executable specifications. By demonstrating program behavior in such a way, it becomes possible to observe deviations from the intended behavior immediately. Deviant behavior is easy to detect for humans, while the underlying violation of semantics cannot always be easily discovered in a formal description.

3 ALGEBRAIC SPECIFICATIONS

There is a growing consensus that formal specification methods are necessary for projects of some complexity; however, it is often unclear what method to choose. Algebraic specifications [Guttag, Horowitz, and Musser, 1978; Liskov and Guttag, 1986] combine the advantages of data abstraction (supporting object-oriented modeling) with an axiomatic method (abstracting from particular execution models) and a functional style (offering clean semantics). In addition, they are easier to learn, read, and write than most other styles.

An algebraic specification defines the behavior of an operation by axioms. These state the operation's effects in terms of other operations on the same data type. One differentiates

constructor operations which construct or change an object and observers which report its state. The behavior of the constructor operations is then expressed either in terms of other constructors or by observers. The well known example of a stack shows how the effect of a push operations is expressed with the observer top (to check the top element) and the additional constructor pop (to remove the top element):

```
top (push (item, stack)) = item
pop (push (item, stack)) = stack
```

This method of description is formally self-contained and complete. It says all about the behavior of push without relying on the semantics of another domain (e.g., of arrays or lists). Note that the variables used in the axioms are implicitly universally quantified, i.e., the above equations hold for all possible stacks and items. There are informal rules for generating axioms which are easy to follow and achieve sufficient completeness and consistency in the sense of [Liskov and Guttag, 1986].

4 FUNCTIONAL PROGRAMMING LANGUAGES

In functional programming languages, everything is a function returning a value. LISP is the oldest and probably most often used functional language, but APL and ML are other well known examples. One distinguishes pure functional languages, where functions are mathematically pure, produce only one result, and do not have side effects, from others which allow side effects to varying degrees. A comprehensive survey and pointers to the literature can be found in [Hudak, 1989].

4.1 Functions as Fundamental Building Blocks

Functional programs consist of the application of functions to some values. The basic control structure is recursion. For example, a typical implementation of factorials in a language like Gofer [Jones, 1994; Thiemann, 1994] looks like this:

```
fac n  =  if n==0 then 1 else n * fac (n-1)
```

In pure functional languages, the functions produce only one result value and do not have side effects. The input parameters are not changed, nor are there any other global variables that carry

state. This fulfills one of the major desires of programming language designers, to achieve locality of concerns: all effects of a function can be seen in its own code and no other effects must be considered.

4.2 Referential Transparency

By the same token, pure functional programming languages achieve "referential transparency", i.e., an expression always describes the same value. Destructive assignments are not possible, as a value can only be assigned once to a variable. This seems a very strong restriction, but guarantees that mathematical proofs of program behavior are straightforward and can use regular logic, without specific inference rules for each control structure [Dijkstra, 1977]. It allows mathematical reasoning about programs based on substitution. Understanding behavior and reasoning about it becomes much simpler and errors are easier to spot. Referential transparency is also found in (pure) Prolog and is probably one of the reasons why Prolog programming has been so successful.

4.3 Type Inference

Functional programming languages in the tradition of ML [Milner, 1978] are strongly typed. Every object has a particular type and the compiler checks that operations can only be applied to objects of the appropriate type (similar to Pascal and other strongly typed languages). Other functional languages like LISP are untyped or the types are checked at run time. A strong typing discipline is helpful for design as it helps discover problems early in the design.

The work on ML showed that the types need not always be provided by the programmer (as in Pascal or C++), but can be inferred by the compiler. Given the obvious types of constants (3 is an integer, 3.57 is a float) the types of variables and expressions can be logically deduced by a type inference mechanism (e.g., inferring from "a = 3.5 + 2.1" that a is of type float). This achieves the same as a strict type system with programmer-provided types, but eliminates the need to define each variable with its type and reduces program clutter.

4.4 Recent Extensions to the Theory of Functional Programming

Substantial progress has recently been made in the area of functional programming languages. New type theories allow for a conceptually clean treatment of inheritance. The application of ideas from category theory, in particular monad transformers, has the potential to overcome limitations of functional languages in dealing with state.

4.4.1 Class-Based Polymorphism

Polymorphism, i.e., the ability of an operation to be applied to arguments of varying types, is very important for modeling spatial data and operations. The intersection of two curves, for example, is the same operation, independent of the type of curves; intersection of two geometric objects has some common behavior for any combination of points, lines, and areas.

Separating the different aspects of polymorphism and combining them with a type hierarchy (inheritance) is conceptually difficult. Current programming languages often mix design issues with implementation concerns. Jones has designed and implemented (in Gofer) a class-based type system with polymorphism, which models multiple-inheritance in a clean way [Jones, 1994].

4.4.2 Dealing with State

The lack of side effects and "updates in place" in functional programming languages have hindered state-based concepts (like databases or I/O) and had to be overcome with loopholes in the past. Some functional languages (e.g., LISP, APL) have included extensions to the strict functional model to make them more usable.

Recently, a coherent extension of the theory has been developed which shows how to integrate database access and I/0 into a pure functional framework [Peyton Jones, and Wadler, 1993; Liang, Hudak, and Jones, 1995]. Imperative aspects (e.g., sequences of operations and side effects) are properly expressed without breaking the functional syntax or semantics. This makes functional languages usable and attractive for the specification and prototyping of database interoperability within Open GIS environments.

Implementations exist and have been used to build standard read and write operations for I/O. Efforts to provide graphical user interface tools in a functional language are underway. The tools to connect to database services are available and semantically more suitable methods, better integrated with the type system, are being studied. It is likely that ideas from the functional database studies [Shipman, 1981] can be reused.

5 USING FUNCTIONAL LANGUAGES FOR SPECIFICATIONS

Functional programming languages satisfy some key requirements for specification languages. On the other hand, they do not support some common features of specification environments. This section discusses the pros and cons of using functional languages for writing specifications.

5.1 Advantages

The use of functional programming languages for writing specifications offers three key benefits:

- *formality*: formal specifications allow for automatic checking of well-formedness, completeness, and consistency;
- *executability*: the specifications can be used as executable prototypes and deviations from intended behavior can be detected and corrected immediately;
- *extendibility*: new specifications can be integrated based on a sound type theory with polymorphism [Jones, 1994].

Furthermore, functional programming languages are readily available (often in the public domain) and possess some additional desired properties:

- functions can be combined to form algebras, allowing for an algebraic specification style;
- referential transparency permits mathematical reasoning by substitution;
- a lean syntax that is very close to standard mathematics affords ease of writing and reading.

Thus, functional programming languages satisfy the key requirements for specification languages. They share with specification languages a tendency to mathematical formalism and

brevity of expressions (APL provided an extreme example). The functional language Gofer, for example, is about three to five times more compact than equivalent, highly modularized Pascal code (even more for C++, if object-oriented features are being used). One of the major contributions to reduce clutter is the type inference mechanism of languages in the ML tradition.

5.2 Limitations

Functional programming languages do not have all properties of the most advanced formal specification environments: they are not designed for a formal verification of specifications, nor for version management, or for documentation and cooperation in teams.

The major issue is documentation of modular decomposition: how is a complex system subdivided into parts. Here, for both specification languages and functional programming languages, additional tools are necessary such as class browsers of the kind used in Smalltalk environments. CASE tools and visual programming environments are much more advanced in this respect, but there are no reasons why such tools could not be constructed for a functional language.

5.3 Constructive vs. Declarative Axioms

Axioms, when expressed in a functional language, are restricted to a constructive form: the left hand side of the axiom has to be a simple expression or contain only constructor operations. This excludes axioms stating general behavior without supplying a rewrite rule on the right hand side. For instance, it is not possible to say that an operation is transitive or reflexive, or to define a matrix inverse by stating that a matrix multiplied with its inverse produces the unit matrix.

This restriction is necessary to allow for execution and, by the same token, to simplify program proofs. According to our observations, it does not affect the use of functional programming languages for specifying GIS objects. In fact the theory of algebraic specifications [Guttag, Horowitz, and Musser, 1978] is based on the exact same restriction. Many operations are defined as simple expressions which can be copied from text books (e.g., coordinate

transformations, geometric constructions). Data types like lists and trees are easily specified by constructive axioms.

The type system can be used to ensure compatibility of definitions according to non-constructive axioms. This resembles the practice in physics to check the dimensions of formulae. Non-constructive axioms can also be used for testing that an implementation conforms to the intended behavior. Boolean functions, testing the intended behavior, are defined and called with appropriate arguments. For example, a function

```
commutative op a b = (a `op` b) == (b `op` a)
```

can be used to test if an operation op is commutative (for a particular pair of arguments). Such tests can be automated by supplying lists of appropriate test cases and applying the test functions to them.

5.4 From Specification to Implementation

Any specification language must be compared with the target implementation language to assure that the specifications can be implemented. There are two issues:

- potential mismatch in concepts
- speed of execution.

The influence that a specification language exerts on the design of the system must be assessed and the effect on the later implementation considered. If the specification and implementation languages differ widely, the implementation requires a substantial redesign. It is then typically very difficult and costly to show that the actual implementation corresponds to the system specified. This may be necessary if different companies are involved and for time- or safety-critical systems.

In contrast to special-purpose specification languages, functional programming languages raise the additional question of execution speed, i.e., whether translation to another language is necessary to achieve efficiency. From the point of view of specifying the behavior of complex spatial objects in GIS, possible efficiency shortcomings of current functional environments are not important. What matters, particularly in an Open GIS environment, is to have an

implementation-independent formalism that can be used to describe the semantics of data and operations and to execute these descriptions for test purposes.

Functional languages have been assumed to be inherently slow [Backus, 1978]. New implementation methods based on advanced mathematics seem to overcome this problem to a large extent. Compiled functional code may just be two to five times slower than a C program, i.e., fast enough for many applications. In the long run, functional programming languages may become efficient enough for implementations in most situations. In the short run, many GIS application programming languages are orders of magnitude slower than standard programming languages. Functional languages can therefore already now improve on the performance of application programming languages.

5.5. Experiences

In Gofer and related languages, we have found a practical tool allowing for a combination of the advantages of formal specifications with those of rapid prototyping. Our experiments have confirmed that such a tool (Gofer or a possible future merger of Gofer with Haskell [Hudak *et al.*, 1992]) provides the expressive power as well as the ease of use necessary for realistic applications in GIS. Gofer is very close to a standard algebraic notation and can be understood with hardly any explanations of the syntax. It is easy to learn and we had students (not from computer science) write non-trivial specifications for geometric data types within weeks.

We have also used Gofer in a "Surveying and Information Systems" course to specify data structures (e.g., a search tree) and to demonstrate their behavior. Gofer is being used in introductory programming courses at some universities [Thiemann, 1994]. We consider to use it as the first programming language for surveying engineering students, replacing an introductory course in Pascal which cannot cover data structures appropriately. In research, we have used Gofer for extensive experiments in specifying hierarchical graph models [Car, and Frank, 1995] and temporal reasoning [Frank, 1994].

6 A SPECIFICATION EXAMPLE

A recent discussion in the North-American Open GIS Consortium raised the issue of how to decide on point equality. Obviously, different systems cooperating in a heterogeneous environment can use different semantics in their equality operations. Interoperability is only possible if these semantics can be described independently of the implementations and succinctly presented to the user of an Open GIS service. The following simplified example achieves this using the functional language Gofer to write algebraic specifications for point data models.

Let us assume three different data models for the storage of point data. System A stores simple lists of coordinates and determines point equality based on coordinate values. System B represents points by coordinates and names and compares point names to decide equality. System C uses the same data type as system B, but decides equality like system A, comparing coordinate values.

The specifications focus on equality operations, leaving additional operations on points (such as a distance) unspecified. While Gofer would offer more compact ways to write these specifications (using classes), the code as written here has the advantage of being largely self-explanatory. The Gofer keyword "data" introduces the definition of a (abstract) data type, while the keyword "type" defines just a type synonym. The signatures of functions list the function name, followed by an argument list. Finally, the axioms describe the behavior of the functions by equations. The crucial difference among the three specifications lies in their last axioms, defining the different semantics of equality.

Data Model of System A

```
type Coord = Int
data Point = New (Coord, Coord)
x       ::      Point -> Coord
y       ::      Point -> Coord
equal   ::     (Point, Point) -> Bool
x (New (cx,cy)) = cx
y (New (cx,cy)) = cy
equal (p,q) = (x(p) == x(q)) && (y(p) == y(q))
```

Data Model of System B

```
type Coord = Int
type Name = String
data Point = New (Coord, Coord, Name)
x       ::      Point -> Coord
y       ::      Point -> Coord
name    ::      Point -> Name
equal   ::     (Point, Point) -> Bool
x (New (cx,cy,n)) = cx
y (New (cx,cy,n)) = cy
name (New (cx,cy,n)) = n
equal (p,q) = name (p) == name (q)
```

```
Data Model of System C
type Coord = Int
type Name = String
data Point = New (Coord, Coord, Name)
x       ::      Point -> Coord
y       ::      Point -> Coord
name    ::      Point -> Name
equal   ::     (Point, Point) -> Bool
x (New (cx, cy, n)) = cx
y (New (cx, cy, n)) = cy
name (New (cx, cy, n)) = n
equal (p,q) = (x(p) == x(q)) && (y(p) == y(q)).
```

The specifications show clearly that the decision of equality between two points depends on other operations, i.e., equality of strings or of coordinates. Assuming a transfer of point data between two systems: even if both systems used the same data model (e.g., both have data model of system B), two points may be determined to be the same in the first and different in the second system. This can happen if the full definition of equality is not the same, e.g., if string equality is case sensitive in one of the two systems. The algebraic specifications expressed in the functional language Gofer reveal these semantic differences in their axioms.

7 CONCLUSIONS

Specification methods are important for GIS design, because GIS are highly complex software systems. The currently available commercial systems with their limitations and known bugs are demonstrating that GIS is stretching current software design methods to the break point. The evolution toward Open GIS, where software components from different vendors cooperate, will only succeed if the interfaces between components can be formally defined. Open GIS depend crucially on formal, testable specifications. A component needs to be testable for compliance with the intended behavior so that components causing a problem can be identified and rectified. This is a commercial and legal necessity, which may decide on the viability of the whole idea of Open GIS.

This paper has argued that functional programming languages can and should be used to specify GIS software. Functional languages are suitable for specifications because they support the standard methods of mathematical proofs, in particular, substitution of equal terms. This effect of referential transparency is not found in imperative programming languages whose complex proof rules cannot be handled by most practitioners. Furthermore, functional programs are executable, allowing the specifiers to check whether they have specified what they wanted. Since it is impossible to formally prove that a program does what a designer wants, executable specifications are the best possible approximation.

Our experience has convinced us that functional programming languages like Gofer or Haskell offer an appropriate compromise solution to the various requirements for a specification language in practice. They are currently limited in dealing with I/0 and databases, but the theory to deal with these shortcomings is developing rapidly. The same theory seems to explain the interaction of modules in a much simpler way than previously possible. Performance of functional programming languages is rapidly improving and they naturally lead to parallel computations.

Current practice to help with interoperability is based on two practical tools: test suits and reference implementations. Test suits are sequences of test cases which are processed and compared with the "correct" results. Reference implementations can be used similarly. For any input, the result for the reference implementation and the system under consideration must be the same. Functional programming can be used in both cases to check implementations, with the key advantage that the specification represents the code of the reference implementation. It can be interpreted much easier than traditional programming code and it can be analyzed using standard mathematical logic.

ACKNOWLEDGMENTS

John Eisner (then working on the North-American datum readjustment) introduced the first author to the topic of data abstraction in Copenhagen, 1981. Benoît David (IGN, Paris) pointed us towards CAML, a functional programming language in the ML tradition, in Paris, 1992. John

Herring, Kevin Hammond, and Simon P. Jones provided helpful comments on drafts of this paper, as did four anonymous reviewers.

REFERENCES

Backus, J. "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs." *Communications of the ACM* 21 (1978): 613-641.

Car, A., and Frank, A.U. "Formalization of Conceptual Models for GIS using GOFER." In *GIS/LIS '95 Central Europe in Budapest*, 1995.

Dijkstra, E.W. "Guarded Commands, Nondeterminacy and Formal Derivation of Programs." In *Current Trends in Programming Methodology, Vol. 1: Software Specification and Design*, ed. Yeh, R.T., 1977: 233-242.

Frank, A.U. "Acquiring a digital base map - A theoretical investigation into a form of sharing data." *URISA Journal* 4 (1 1992): 10-23.

Frank, A.U. "Qualitative Temporal Reasoning in GIS - Ordered Time Scales." In *6th International Symposium on Spatial Data Handling in Edinburgh, UK*, IGU, 1994: 410-430.

Guttag, J.V., Horning, J.J., and Wing, J.M. *Larch in Five Easy Pieces*. Digital Equipment Corporation, Systems Research Center, 1985.

Guttag, J.V., Horowitz, E., and Musser, D.R. "The Design of Data Type Specifications." In *Current Trends in Programming Methodology*, ed. Yeh, R.T., Vol. 4: Data Structuring. Prentice Hall, 1978: 60-79.

Head, G.E. "Six-Sigma Software Using Cleanroom Software Engineering Techniques." *Hewlett-Packard Journal* 1994 (June 1994): 40-50.

Hudak, P. "Conception, Evolution, and Application of Functional Programming Languages." *ACM Computing Surveys* 21 (3 1989): 359-411.

Hudak, P. *et al.* "Report on the functional programming language Haskell, Version 1.2." *SIGPLAN Notices* 27 (5 1992).

ISO. *The EXPRESS language reference manual*. ISO TC 184, 1992. Draft International Standard ISO/DIS 10303-11.

Jones, M.P. *Qualified Types: Theory and Practice*. Ph.D. Thesis, Programming Research Group, Oxford University, Cambridge University Press, 1994.

Kuhn, W. "Defining Semantics for Spatial Data Transfers." In *6th International Symposium on Spatial Data Handling in Edinburgh, UK*, IGU, 1994: 973-987.

Liang, S., Hudak, A P., and Jones, A M. "Monad transformers and modular interpreters." In *ACM Symposium on Principles of Programming Languages in* ACM, 1995.

Liskov, B., and Guttag, J. *Abstraction and Specification in Program Development*. The MIT Electrical Engineering and Computer Science Series, Cambridge, MA: The MIT Press, 1986.

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

O'Callaghan, J.F. "The Impact of Applications and Information Technologies on Geographic Information Systems." In *GIS: Technology and Applications, Far East Workshop on GIS in Singapore*, edited by Hongjun, Lu, and Beng, Chin Ooi, World Scientific, 1993: 1-6.

Peyton Jones, S.L., and Wadler, P. "Imperative functional programming." In *ACM Symposium on Principles of Programming Languages (POPL) in Charleston*, ACM, 1993: 71-84.

Shipman, D.W. "The Functional Data Model and the Data Language DAPLEX." *ACM Transactions on Database Systems* 6 (March 1981).

Smith, T.R., and Frank, A.U. "Very Large Spatial Databases - Report from the Specialist Meeting." *Journal of Visual Languages and Computing* 1 (3 1990): 291-309.

Thiemann, P. *Grundlagen der funktionalen Programmierung*. Leitfaden der Informatik, Stuttgart: B. G. Teubner, 1994.

Voisard, A., and Schweppe, H. "A Multilayer Approach to the Open GIS Design Problem." In *2nd ACM GIS Workshop in Gaithersburg, MD*, edited by Pissinou, N., and Makki, K., ACM Press, New York, 1994: 23-29