Speaking in Tongues:
The Language of Revelation

Scott D. Daniels

Oregon Graduate Institute
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 91-007

January 1992

Speaking in Tongues: The Language of Revelation

— Scott David Daniels
Oregon Graduate Institute
daniels@cse.ogi.edu

Abstract

The Revelation project addresses the problem of query optimization in object-oriented databases. The database community is using object-oriented technology to provide extensibility for database types. Unfortunately, this technology (when naively applied) reduces the ability of the database system to perform adequate query optimization. In Revelation we hope to reduce this impact by carefully orchestrating the techniques we use to pierce the veil of object encapsulation during optimization. This paper argues that the language used for implementing the objects in such a system should be strongly typed and polymorphic, and explores the constraints on such type systems. It then proposes a type system to match these constraints, and addresses some further aspects of the planned language environment and their relationship to the project goals of query optimization.

and out of long-term store is a non-trivial problem. Attempting to build a database system without persistence of the data is a bit like "playing tennis without a net."* In a database system, however, we may presume the existence of a persistent store. This store provides a convenient place to retain information from code analysis, something that often proves to be an issue in more traditional language environments. Since we have to be able to store objects for the database user, we may presume a mechanism to save any structured data that the language system needs — a sufficiently general system must already be in place.

2.2 Object-Oriented

Second, we speculate on what properties derive from the fact that this database is object-oriented. We will draw from the meaning of "object-oriented" in both the computer language and database communities.

While the relational model has worked extremely well for a large number of database applications, there are applications that are struggling with its limitations. These struggling applications include engineering databases, where the information stored has more structure than the simple tables available through the relational model. The big problem seems to be that in relational databases all of the types that may be stored in the tables must be provided by the implementors of the database, as well as the operations available on those types.

While the standard set of well-understood types typically defined by a database works well for many applications (storage and retrieval of textual information, accounting applications, ...), it begins to fall apart for applications where more complex elements are desired. Dates are a simple example, where one might want operations like *print, compare, subtract* (yielding days), and *add* (to days). If dates seem obvious, imagine areas in a plane, volumes in space, or hyper-volumes in space-time. The number of basic operations grows, and it is no longer clear what their definition might be: for defining *distance,* Euclidean distance might be most reasonable for general consumption, but Manhattan (or city block) distance is more appropriate for applications such as printed circuit board layout.

As these types and operations get more specialized, it seems less reasonable to include them in the database implementation. What we really want is to provide a way to extend the basic types of the database without changing the basic database code. Let a specialist in the application environment design his special types, and keep the database expertise focussed on database issues such as storage and buffer management.

Object-oriented languages have been quite successful in providing language users the ability to define new types and specify operations on those

* Robert Frost originally said this of free verse, but we feel sure if he knew the issues, ...

types. In fact, this may be one of the reasons that this programming style has become so popular. By shifting the focus from the structure of the data to the operations on that data, this technique has allowed a tighter encapsulation of implementation details. It is, we believe, this encapsulation that provides much of the power of the object-oriented paradigm by enhancing modularity of the code. Implementation details tend to be accessible in much narrower scopes, so the chances of building systems that can survive re-implementation of inefficient subsections are increased correspondingly.

Along with this focus on the operations on the data, the data structure itself is often hidden. When this happens, the data structure becomes a black box, and we can think of the operations as providing a "behavior" for the "object" (the encapsulated data structure). If all we examine is this "behavior," two radically different data structures can "behave" the same, and may in some sense be indistinguishable. Thus, this "behavioral identity" provides the basis for a kind of polymorphism that will allow different implementations (perhaps representing different engineering design trade-offs) of the same behavior.

Another major feature of these languages is called *inheritance*. This is a technique that allows implementation by providing organized ways of reusing an existing implementation. Roughly, the programmer specifies some implementation close to what he wants, and rewrites those parts that are not exactly what he wants. Since most programming tasks involve large amounts of similar code, this technique provides a big savings when properly used. Since inheritance involves a dynamic link between implementations, it provides a way to "fix a bug once," in that the code to perform some function is stored once, rather than copied.

Using inheritance to program systems is sometimes called "programming by differences." Implementation in an object-oriented system consists primarily of defining *classes*. Although almost all *classes* could be defined "from scratch," most of the object-oriented languages use inheritance. Roughly, inheritance allows an implementation to be defined by specifying an existing class as a *superclass* (or a set of superclasses for multiple inheritance) and writing a set of updates that define the new class in terms of the superclass(es). If no updates are specified, the new class behaves just like the old class(es), so objects will have the same behavior. There are basically three kinds of updates possible. First, the object's internal data structure may be extended (typically adding a field to a tuple). Second, new messages may be added for the object to respond to. Of course, in this case methods must be defined to perform the operations implied by the new messages. Finally, new methods may be defined to replace the response of the object to messages that had methods defined in the superclass(es).

When defining methods for any of these updates, one has available two special keywords for referring to the object that is executing the method:

self and *super*. Messages sent to this object using the term *super* use the method written in the superclass, i.e., the old behavior of the message. Messages sent to this object using the term *self* use the method provided by this object to the outside world. Note that there is no way to specify use of a parallel method (one from the same class): the choices are former behavior (a method from the superclass) or external behavior (the method associated with the leaf class). While this specification seems to be a missing feature, the lack simplifies modification. Since so many messages go out to the interface, a later inheritance simply redefines a method by superceding it. Now the implementation uses the new improved method for messages to *self* automatically.

Almost all forms of object-oriented programming allow at least some form of polymorphism. Polymorphism allows procedures to be written that can operate on a range of types of objects, without a case statement enumerating the possible types that can be used. Correct source code can be written and compiled that performs operations on objects whose structure has not yet been fully defined. It is just this sort of operation that is necessary in a database whose types are not fully determined when the system is already running.

2.3 Query Optimization

Finally, we come to query optimization. If we are going to do any meaningful query optimization, we will need to do more than proceed from message to message, interpreting the system language. We will need some form of partial information about the objects stored in the database. If we have no such information, it can cost us as much time to find out how to optimize a query as it would to execute it in the most naive way. Somehow, we need a way to obtain information that will remain true at least for the duration of the execution of the query, and not simply at the very moment we ask the question.

Computer languages of all kinds have had to deal with such issues from a slightly different point of view. The problem shows up in *Lisp* when we try an determine exactly what is being referred to by a name (the famous *funarg* confusion). In languages such as *Pascal,* the type of the variables provides a kind of partial information (for subrange types, that is exactly what is going on). This spectrum of ideas can be described as questions of *bindings [Lomet80] [Lomet85].* In translating and executing a program, the language system goes through a series of *bindings,* first binding names to variable identities (and hence some associated type), later variables are bound to storage, and even later values are bound to the variables (or the storage). Naturally, in the last case, rebindings will occur, and so we cannot rely on these bindings to be permanent. Even in functional languages, this rebinding occurs when procedure actuals are bound to formals. As long as the same function is used several times, some form of rebinding is needed.

We see a natural progression here with the lifetimes of bindings following a kind of nesting. The longer-lived bindings specifying less information, but remain in force while shorter-lived (inner) bindings come and go. The innermost bindings we will call "narrow" binding, as the value is most tightly constrained by these bindings. As we travel out along the nesting relationship, we refer to the bindings as being "broader". These bindings provide less constraint, but also remain in force longer. This sounds ideal for use in query optimization — you use the "narrowest binding" that will remain valid for your query in order to extract the most information about how to best perform the query. A query optimized with more "broadly bound" information may be stored and used later (as long as the binding remains in effect).

Finally (and probably most problematic), query optimization may use information about "tendencies," in which case obsolete information may work out perfectly well. This kind of information includes things such as set sizes and query selectivities. While such information is likely to change frequently, it is more often the order of magnitude than the actual value that is used in decision making. In fact, the value is used in comparisons between estimated costs of possible query plans; small changes in values that change the choice of plans are typically indicative of nearly equal-cost plans. A real problem with this kind of information is in identifying exactly when it has gone out of date, since we want to straddle the fence between regarding it as changing whenever the value changes and regarding it as a constant. Luckily, the result of executing queries with incorrect information of this sort typically slows down a query rather than rendering its results incorrect. For this kind of information, it may be safe to presume the future will be like the past, since that is often the best guess. We can use some form of "aging" to indicate that such plans must be re-optimized, or place the responsibility on the shoulders of the database administrator.

3 Choices for the Type System

The justification for these parameters really have more to do with large system organization than they have to do with this particular project. We hope to sacrifice as little practical functionality as possible from systems as general as Smalltalk–80 *[Goldberg83]*, *[Goldberg84]*, however we will make a few restrictions where the cost to query optimization of avoiding those restrictions seems prohibitive.

Chief among these restrictions in Smalltalk–80 is the elimination of the *become:* message. Its function is to swap identities between two objects. Unfortunately, this behavior can wreak untold havoc in a database. Not only may the object be bound to a variable that is typed incompatibly with the object, but it may even be stored as a member of a set or relation where the typing on the members provides structure. Once created, we will not

allow an object to change class, nor to change its instance variables except by invocation of its interface messages.

A similarly problematic feature in Smalltalk–80 is the ability to change the methods in a class while the system is using objects of that class (and possibly even executing the very method being changed). This mutability also presents intractable analysis problems for query optimization: "I know what this message does now, but what about after I've called this procedure that might rewrite all of the methods in the system?" Fortunately, such methods are nearly intractable for humans as well, and no self-respecting programmer will claim the loss of such operations is a tragedy (outside of a few perverse moments for the truly warped).

3.1 Polymorphic

We want to obtain the maximum amount of polymorphism from the system consistent with a goal of performing query optimization (clearly, with no polymorphism optimization would be easier, since we could then translate the query to a simple algebraic form and then optimize that). We believe (as many others do) that it makes little sense to allow messages to be sent to an object that does not implement them *[Black86] [Black87] [Hutchinson87]*. Thus we will disallow systems where a class re-implements the *doesNotRespond* method, and attempts some other technique to respond to the message. It is not that we think such systems are useless. Quite to the contrary, this technique has been used to provide delegation *[Stein87]* and communication with objects on remote systems. It is simply that such techniques make the analysis for query optimization nearly intractable. We do not, however, accept that the speed gains possible by relying solely on structural polymorphism* are worth the loss in generality of the source language. The reason we want a more general form of polymorphism is that we expect the user will need to change the implementations for some of his types, and we wish to allow the old and new implementations to co-exist in the system.

By supporting run-time polymorphism for all messages to all objects (actually, the primitives for integers, characters, and strings may not work this way), we hope to create a situation where a collection can contain several implementations of the same type (perhaps from a span of several customer class rewrites), and perform correctly with no explicit provision for that in the source code. If we can achieve this polymorphism at a reasonable cost, we will have provided a nice tight encapsulation of the implementation.

3.2 Strongly Typed

We believe in strongly typed language systems. They provide the kind of consistency check that helps a programmer catch the *thinkos* that plague

* Where the only polymorphism provided is via shared implementations.

us all: reversed parameters, wrong variable passed, etc. When these errors are fixed, programs stand a much greater chance of working correctly, rather than producing some meaningless result. In a simple program, such a failure (once detected) will require that the program be fixed and re-run. In a database, it could be weeks before the error is found, and re-executing all transactions over that period with the "new, improved, correct" code providing different results is typically not an option.

A second advantage we hope to gain from this strong typing is enforced encapsulation. We will only allow messages to be sent to a variable if the typing on that variable has that message in its repertoire. This encapsulation should increase the likelihood that re-implementations will work properly wherever they are needed.

Finally, and quite a bit more pragmatically, this typing provides us with an ideal spot to coalesce the partial information we accumulate about how this code might be optimized. The skeleton of typed variables and expressions provides a wonderful framework for accumulating facts for optimization. We finally have a good place to hang the partial information that we hope to obtain by code examination.

3.3 Classes as Types

In most object-oriented systems there is a natural candidate for a type: the class. A class provides the structure and code for all of its instances. Since all such objects (instances of a single class) have the same structure, and since they all use the same code to respond to messages, they all behave similarly. This similarity is much like the similarity of floating point numbers to each other in a piece of hardware, and there this match is viewed as a type. In most of the typed object-oriented languages classes and their descendedants constitute types *[Meyer86] [Stroustrup86a] [Stroustrup86b] [Stroustrup89] [Graver90]*.

As far as it goes, such a definition seems to work. However, when you begin to explore the relationship between different classes, problems crop up. It is tempting to take the ideas of subtypes and subclasses and identify them. "But that would be wrong!"* Certainly you lose much of the polymorphism that we feel is at the heart of the object-oriented paradigm with this identification. If we want to allow for re-implementations, such a type system would constrain us to using subclasses of the original class in order to get the types right. This means that we either lose freedom in how the new implementation works, or we waste storage by ignoring the instance variables (and even the methods) of the former implementation.

As early as 1986, Alan Snyder *[Snyder86]* pointed out that the advantages of encapsulation of abstract data types are at odds with the technique

* Richard Nixon in the infamous White House tapes, but we feel sure if he knew the issues, ...

of implementation by inheritance. Basically, too much behavior is inherited "accidentally" to provide a good type for the resulting object. There is another objection to types as classes: object-oriented languages in general, and Smalltalk–80 in particular, use an object's behavior as its meaning. This use of behavior rather than structure allows a rich variety of implementations to be used for the same purpose. To incorporate all of these implementations in a single class, one defines an "abstract class," and has every implementation inherit from it.

3.4 Message Sets as Types

Another approach to providing a typing for object-oriented systems is to treat the message name–signature pairs themselves as the atoms of a type system *[Black86] [Black87] [Hutchinson87] [Cardelli84a] [Cardelli85].* With this point of view, the type of an object is the set of message signatures that it implements. Some fairly impressive things can be done with such systems, including type inference (which frees the programmer from having to specify the types of all of the variables in his program). Unfortunately, these systems also have some of the problems that Snyder pointed out *[Snyder86].* Since inheritance provides all of the messages of the superclass as default implementations for the subclass, messages will show up in the interface of a subclass that are merely implementation details. This expansion of the interface, however, can interact poorly with a system that avoids the *doesNotRespond* problem. If one thinks of the *doesNotRespond* behavior as providing a warning of a type failure, one sees that by eliminating that style of type problem we have trimmed the type failures down to those where messages not in the interface are used. Thus, if we had built a *car,* for example, from a *bicycle* with some added messages, this "messages as the morphemes of a type" approach will allow messages through to our *car* that only affect the *bicycle* portion of the data structure, possibly in ways inconsistent with the *car* abstraction. So we see such an approach cannot detect the use of inappropriate messages that could scramble the internal state of an object.

4 Our Solution

4.1 Protocols as Types

We intend to separate the ideas of specification and implementation into two separate hierarchies. The specification hierarchy is a hierarchy of *protocols* and addresses the behavioral semantics of the objects. The implementation hierarchy is a hierarchy of *classes* and and addresses the internal workings of the objects. For our typing, we will use the term *protocols.*

What we propose is a system where the types are explicitly declared, along with the messages and type signatures that constitute the interface to objects of those types. We call the types thus declared "protocols" (by simple analogy with communications systems). We intend each protocol

to correspond to some abstract property, not simply to be a collection of messages. Thus an object could be said to obey the "stack protocol," if, for example, it not only had appropriate messages *(push:, pop, empty),* but it also behaved like a stack. These abstract properties mesh nicely with Goguen's ideas on type systems *[Goguen86]*. In his type system he associates types with functions and axioms (the axioms provide properties of the the type and its functions, allowing statements of properties such as transitivity, associativity, etc.). Initially we will simply rely on comments to explain these properties, but this does provide a good basis for an extension of this system.

Since we are appealing to the meaning of the code here, we must rely on explicitly provided (as opposed to automatically detected) links between classes and the protocols that they implement. We can, however, indicate that certain protocols are always satisfied whenever a particular protocol is satisfied (this satisfaction is clearly transitive). Any object which conforms to one protocol must obviously also conform to all other protocols which that protocol satisfies. This leads to a separate hierarchy of protocols from the transitive closure of this subtype relation. If adding a link in this relation creates an equivalence group of protocols, we may either reject such links, or warn the user since creating this equivalence class may not be intentional. Similarly, when the class–protocol link is made, the system may find that the class cannot implement the protocol, in which case it will reject the link and inform the user.

4.2 How Does This Help?

This use of a protocol as a type certainly allows us the polymorphism we feel is so important. Since the types describe behavior, not structure, we are free to allow many structures to satisfy a single type. In fact we may even allow one implementation to be used for several different types (a dequeue could be used as a dequeue, a queue, and a stack).

At any given moment in the database, there is some number of classes, and a number of protocols that they obey. We intend to be able to travel between the protocols and the classes that implement them, so we may discover facts about all current implementations of a protocol that are "coincidental" (properties of implementation rather than definition). This information can be used to optimize queries, provided we have a way of retracting this optimized code whenever changes in the protocols or classes invalidate some of these assumptions. To simplify this invalidation, we assume: "no protocols or classes will be changed or created during the execution of any code."

Exactly what "coincidences" are we looking for? Well, certainly one of the questions the query optimizer will ask the type system is: "How many distinct methods exist for this message sent to an object that obeys this protocol?" If the answer is exactly one method then the query optimizer might

expand the query in-line (*unfold* the message). Note that several distinct classes may use the same method simply by inheritance from a single class; determining this answer does not imply full-fledged function comparison. This unfolding can lead to similar requests on discovered messages to whatever level of expansion desired. This technique is similar to "specialization" or "customization" in Self *[Ungar87][Chambers89]*.

Other possibilities include discovering that there are precisely two such methods, and expanding the code to include a test based on the class of the object being sent the message. Yet another useful property to know about a method is whether it is a "leaf-level" method. Such methods are capable of computing their results with access only to a fixed set of the object's state variables (no further messages or recursion involved). Such methods may be compiled into machine code and executed at full machine speed given hooks to the necessary data, and execution of such methods may be delayed until all variables are accessible. Simply delaying evaluation till all data is available provides performance improvement by reducing the percentage of time spent locking and unlocking data in the buffer. In addition, this can greatly speed the processing of any unordered collection, since it is possible to execute the method as elements of the collection become available (all elements on the same buffer page, for example) rather than having to execute them in some arbitrary sequential order. Notice that this technique works out well in conjunction with the previous mentioned method unfolding, allowing even more methods to be executed in a single call to generated code.

Another useful property is to observe that some methods only read state, hence they can be arbitrarily interleaved with each other (another big savings for processing collections of all kinds). If it can be verified that all methods for a message in a particular protocol examine only globals and internal state, but only write internal state, then we may once again reorder the operations in the collection based simply on when the data arrives in the buffers, and avoid the excessive I/O normally implied by sequential processing of collection members.

4.3 Other Unusual Language Features

In order to avoid focusing on syntactic detail early in the project, we have decided to use a structural, rather than textual definition for the language. All referents (variables, protocols, classes) will be referred to by identity rather than by name. Thus we avoid name scope issues (the identity is what is specified by the code, not the name), and allow for later development of the "syntactic sugar" to create readable source code. Since the query optimizer may have to do expression unfolding, defining the language structurally rather than syntactically eliminates the parsing portion of that task.

The creation operation is performed with a protocol–class pair or just a protocol (in which case a default class is chosen), rather than a class. This

choice parallels our focus on behavior rather than implementations. We speculate that often code and queries care only about "any implementation of this abstraction," and, if so, the protocol-only form of creation will work out quite nicely.

Because of the way objects are created with protocols, we will store with each object not only its class, but its creation protocol as well. We intend to experiment with restricting message access to objects to those messages available in the protocol from which it was created (although messages to the special names *self* and *super* will remain unrestricted). What we hope to provide with this restriction is insurance that only the specified interface to an object is used, and therefore ease the task of an administrator who wishes to replace an implementation.

Since we are using a structural rather than textual representation for the code, a binding environment that maps names to objects such as protocols and classes will be needed to create code. This environment should contain the roots of the database as well, with protocol–object pairs for things such as the basic collections the queries will be run against. This environment will hold the dependence tracking mechanism that is used to retract code optimizations that rely on our "implementation coincidences," since it is in this environment that implementations are added to the protocols. Here is also a reasonable place to keep statistics about the root objects; this way the same retraction mechanism can be used to propagate new statistical information. Other information that might be kept here includes such things as common queries in both their raw and optimized forms. The query optimizations may then receive the same retractions as the code does, thus allowing frequent queries to remain optimized without the cost of regular query optimizer runs.

5 More Ideas on the Protocol Hierarchy

Modifications within the hierarchy of protocol must be done, if at all, with extreme care. When a new protocol is added at the leaves of the hierarchy, the operation is straightforward. However, when a message is added to an existing protocol, all classes that implement that protocol (or any of its subprotocols) must be checked to make sure they have an appropriate method. Similarly, adding a protocol–protocol link may add messages to the new subprotocols, and more checking must ensue in those cases as well. In order to perform some of these operations, it may be necessary to create a copy of the old protocol, and modify the copy, then put the new protocol in the binding environment under the old name. This copy may require a subgraph copy, but (since the protocol–subprotocol relation is a preorder), the copy should not be overly complicated.

Because the resulting messages in the protocols are not always obvious, there must be a means added to the system to query the protocol hierarchy

and determine exactly what each protocol actually entails. In fact, to enable reasonable development, we need to be able to dump the class and protocol hierarchies. This dump is not entirely straightforward, since the names for classes and protocols lie in the binding environment, and there may be no name for an old implementation (or protocol).

6 Examples of Query Optimization Information

Here are the kinds of questions that our system should be able to answer. We anticipate that a sophisticated a query optimizer (with some partial evaluation capabilities) can develop quite sophisticated plans by combining the answers to such questions with information about the top-level named structures (things such as sizes, statistics, and actual object bindings).

I. How many classes implement this protocol?
II. How many methods for this message/protocol?
III. What is the method for this message when sent to this object?
IV. Is this message/protocol always implemented by "ground" methods?
V. Is this message/protocol "functional?"
VI. Is this message/protocol a "local-read?"
VII. Is this message/protocol "read-only?"
VIII. Does this message/protocol write local state?
IX. Does this message/protocol write global state?
X. What are the methods for this message when sent to an object with this protocol?

7 Conclusion

The Revelation project addresses the problem of query optimization in object-oriented databases. We have described in detail the reasoning leading to our selection of the type system that we intend to use: *protocols*, and explained why we feel this typing best captures the concept of the behavior of an object. We have further described how such a type system would properly describe a series of implementations of the same classes, and hence serve as a basis for typing in a query-optimized object-oriented database. We have indicated how information collected these types can be used to provide information helpful to a query-optimizer. Finally, we have described briefly the name binding space at the root of the database, and shown how it is used to organize protocol and class replacements.

8 Further Research

Cook, Canning, Hill, and Olthoff *[CCHO88]* have a typing that nicely captures what (at first glance at least) we may want for parameterized types.

They call it "F-bounded lambda typing", and their types are of the following form:

$$\forall T \text{ such that } T \sqsubseteq F(T) : \sigma(T)$$

This typing works well for describing such things as Set(TotalOrderedType), an almost essential type for describing things such as binary relations.

A prototype will be built to find how these decisions will work in practice.

The dependence tracking mechanism needs substantial work. In the initial implementations, it will be done by hand; only later will we attempt to provide automatic support.

Existential queries in a database (queries which ask about the existence of objects with certain properties) seem in conflict with the whole idea of encapsulation. It does not seem to be anybodys business whether or not a car is implemented by using a bicycle as a sub-object and using the bicycles response to messages to guide the car's response to similar messages. However, if queries can be made against "all bicycles," this implementation detail will come to the fore. Perhaps an "implied universe" for existential queries is needed, with the universe corresponding to the NULL-named object in the name-binding space.

References

[Black86] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object structure in the Emerald system. *SIGPLAN Notices*, 21(11):78–86, November 1986.

[Black87] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and abstract types in Emerald. *Transactions on Software Engineering*, SE–13(1), January 1987.

[Blair89] G.S. Blair, J.J. Galagher, and J. Malik. Genericity vs. inheritance vs. delegation vs. conformance vs... *Journal of Object–Oriented Programming*, 2(3):11–17, 1989.

[Bloom87] T. Bloom and S.B. Zdonik. Issues in the design of object–oriented database programming languages. *SIGPLAN Notices*, 22(12):441–451, 1987.

[Borning82] A. Borning and D. Ingalls. A type declaration and inference system for Smalltalk. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 132–141. ACM, January 1982.

[Breazu89] V. Breazu-Tannen, P. Buneman, and A. Ohori. Can object–oriented databases be statically typed? In *Second International Workshop on Database Programming Languages*, pages 226–237. OCATE, Morgan Kaufmann, 1989.

[Burstall77] R.M. Burstall and J.A. Goguen. Putting theories together to make specifications. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence (IJCAI77)*, pages 1045–1058, 1987.

[Canning89a] P.S. Canning, W.R. Cook, W.L. Hill, and W.G. Olthoff. Inheritance is not subtyping. In *The Fourth International Conference on Functional Programming Languages and Computer Architecture (FPCA '89)*, pages 273–280. ACM, September 1989. Imperial College, London.

[Canning89b] P.S. Canning, W.R. Cook, W.L. Hill, and W.G. Olthoff. Interfaces for strongly typed object–oriented programming. *SIGPLAN Notices*, 24(10):457–467, 1989.

[Cardelli84a] L. Cardelli. Basic polymorphic typechecking. Technical Report 119, AT&T Bell Laboratories Computing Science, 1984.

[Cardelli84b] L. Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types*, number 173 in Lecture Notes in Computer Science, pages 51–67. Springer-Verlag, June 1984.

[Cardelli85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[Chambers89] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for SELF, a dynamically–typed object–oriented programming language. *SIGPLAN Notices*, 24(7):146–160, 1989.

[Cook90] W.R. Cook, W.L. Hill, and P.S. Canning. Inheritance is not subtyping. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 125–135. ACM, 1990.

[Cook89] W.R. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. *SIGPLAN Notices*, 24(10):433–443, 1989.

[Cox86] B.J. Cox. *Object Oriented Programming: An Evolutionary Approach.* Addison–Wesley, 1986.

[Goguen86] J.A. Goguen. Reusing and interconnecting software components. *Computer*, 19(2):16–28, Febuary 1986.

[Goldberg84] A. Goldberg. *Smalltalk–80: The Interactive Programming Environment.* Addison–Wesley, 1984.

[Goldberg83] A. Goldberg and D. Robson. *Smalltalk–80: The Language and Its Implementation.* Addison–Wesley, 1983.

[Graefe88] G. Graefe and D. Maier. Query optimization in object–oriented database systems: The REVELATION project. In *Advances in Object-Oriented Database Systems*, number 334 in Lecture Notes in Computer Science, pages 358–363. Springer-Verlag, September 1988.

[Graver90] J.O. Graver and R.E. Johnson. A type system for Smalltalk. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 136–150. ACM, January 1990.

[Hutchinson87] N. Hutchinson. *Emerald: An Object–Based Language for Distributed Programming.* PhD thesis, University of Washington, January 1987. Technical Report 87–01–01.

[Johnson86] R.E. Johnson. Type–checking Smalltalk. *SIGPLAN Notices*, 21(11):315–321, November 1986.

[Johnson88] R.E. Johnson and B. Foote. Designing reusable classes. *Journal of Object–Oriented Programming*, 1(2), June 1988.

[Kafura89] D.G. Kafura and K.H. Lee. Inheritance in actor–based concurrent object–oriented languages. *Computer Journal*, 32(4):297–304, 1989.

[Kamin88] S. Kamin. Inheritance in Smalltalk–80: A denotational definition. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 80–87. ACM, 1988.

[Lomet80] D.B. Lomet. A data definition facility based on a value–oriented storage model. *IBM Journal Res. Development*, 24(6), November 1980.

[Lomet85] D.B. Lomet. BIND. Private Communication, July 1985.

[Lunau89a] C.P. Lunau. Separation of hierarchies in Duo–Talk. *Journal of Object–Oriented Programming*, 2(2):20–25, 1989.

[Lunau89b] C.P. Lunau. *Separation of Type and Class Hierarchies in Object–Oriented Languages.* PhD thesis, University of Copenhagen, September 1989. Technical Report 89–118.

[Maier89] D. Maier. Why isn't there an object–oriented data model? Technical Report CS/E–89–002, Oregon Graduate Center, May 1989. Also summarized in IFIP 11 World Computer Conference, S.F., Ca. 1989.

[Meyer86] B. Meyer. Genericity versus inheritance. *SIGPLAN Notices*, 21(11):391–405, November 1986.

[Mitchell90] J.C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 109–124. ACM, 1990.

[Sciore89] E. Sciore. Object specialization. *Transactions on Information Systems*, 7(2):103–122, April 1989.

[Snyder86] A. Snyder. Encapsulation and inheritance in object–oriented programming languages. *SIGPLAN Notices*, 21(11):38–45, November 1986.

[Stein87] L.A. Stein. Delegation is inheritance. *SIGPLAN Notices*, 22(12):138–146, December 1987.

[Stroustrup86a] B. Stroustrup. *The C++ Programming Language.* Addison–Wesley, 1986.

[Stroustrup86b] B. Stroustrup. An overview of C++. *SIGPLAN Notices*, 21(10):7–18, October 1986.

[Stroustrup87] B. Stroustrup. What is object–oriented programming. In *European Conference on Object-Oriented Programming*, number 276 in Lecture Notes in Computer Science, pages 51–70. Springer–Verlag, 1987.

[Stroustrup89] B. Stroustrup. Multiple inheritance for C++. *Computing Systems*, 2(4), 1989.

[Ungar87] D. Ungar and R.B. Smith. Self: The power of simplicity. *SIGPLAN Notices*, 22(12):227–242, December 1987.

[Wegner88] P. Wegner and S.B. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In *European Conference on Object-Oriented Programming*, number 322 in Lecture Notes in Computer Science, pages 55–77. Springer–Verlag, 1988.

[Zdonik90] Stanley B. Zdonik and David Maier, editors. *Readings in Object–Oriented Database Systems.* Morgan Kaufmann, 1990.