Classification in Object-Oriented Systems

Peter Wegner, Box 1910, Brown University, Providence, RI 02912

Abstract: We characterize object-oriented systems by their classification mechanisms because they are prescriptive in their classification paradigm but permissive in their communication and message passing mechanisms. They extend the flat classification mechanism of traditional type systems to tree-structured collections of types with common operations and attributes. We review the relation between biological and object-oriented classification, relate the classification paradigm to communication and state transition paradigms, and explore its realization by calculi and algebras. Object-oriented type systems determine a calculus of classes weaker than calculi for computing with values that can be modelled either algebraically [BW] or by the second order, polymorphic, lambda calculus [CW]. A fuller discussion with more technical details can be found in [We].

Contents:

1. Classification in Biology and Mathematics

Classification arises from the universal need, in any domain of discourse, to describe uniformities of collections of instances. It is a basic activity of infants in organizing sense impressions, of scientists in organizing knowledge within scientific disciplines, and of application programmers in organizing domain knowledge and behavior. We illustrate the scientific importance of classification with examples from biology, mathematics, and computer science.

Biological classification originated with Aristotle and was rediscovered and refined by Linnaeus and Darwin [Ma]. Aristotle introduced both a top-down logical technique for classification by a sequence of dichotomies (not unlike the game Twenty Questions) and a bottom-up empirical technique of examining variability of individuals to determine significant attributes for distinguishing among species. Linnaeus in the 18th century developed a widely accepted classification system in terms of the concepts "class, order, genus, and species" that provided an effective method for identification of species. The Linnaean system was successful in identifying species by differences in characteristic attributes but made no attempt to explain similarities among species. It illustrates the conflict between the goals of identification of differences between classes in an already organized domain of discourse and understanding of relationships among classes in an uncharted domain of discourse, a conflict which arises in many different disciplines. The view appropriate for identification of classes is generally top down, while that for understanding of relationships is generally bottom-up. In applications programming the domain of discourse is generally uncharted at the time of design of a new application but is organized at the time of operation and maintenance. Thus creation of an applications program generally is more bottom-up in its approach than subsequent description and presentation of the program.

Darwin's approach to classification is based on evolutionary divergence of species from common ancestors. It is explanatory as well as descriptive, explaining the diversity of organic life by evolutionary mutation and survival of the fittest. Evolutionary classification was largely developed by Darwin himself in Chapter 9 of his "Origin of Species". But Darwin was careful to distinguish between pure genetic hierarchies resulting from evolution and organizational classification hierarchies designed for explanation and identification. The adoption of evolutionary metaphors in object-oriented systems is likewise motivated in part by evolutionary and in part by organizational considerations.

Both the design process and the operation and maintenance phase of large application programs are evolutionary processes. In designing large programs it is natural to start with high-level incomplete behavior descriptions, to fill in details that specialize and complete high-level descriptions, and to realize changes and enhancements by subtypes that augment and modify existing types. However, in addition to external evolution-driven changes there is an internal (endogenous) reason for the usefulness of inheritance in organizing knowledge domains. Inheritance hierarchies serve to systematically capture similarities among collections of types in a supertype and encourage us to factor out common attributes among a collection of similar phenomena. Such factoring out of similarities lies at the heart of understanding and organizing any application domain. The expressive power of inheritance is due to the fact that it simultaneously captures evolutionary change and organizational similarity in a single framework.

How do the mechanisms of type inheritance and biological inheritance differ? Type changes are introduced in a systematic, goal-directed manner while biological changes occur by random mutation. Goal directed change is much faster and more efficient than change by random mutations. The greater control over system evolution by the creator and the discrete nature of systems allows us to be much more precise about behavioral properties and mechanisms for change of object-oriented classes than of biological classes. In addition the ability to use the same mechanism for structured description of knowledge domains as well as for the management of change adds greatly to the expressive power of object-oriented methodology.

What is the relation between classification of natural phenomena and classification in the world of concepts and mathematical entities? Classification of concepts originated with Plato, whose theory of forms postulated ideal types (the ideal table) which are approximated by imperfect instances in the real world. Such ideal forms do not exist in the empirical world of natural phenomena but do populate the artificial world with which we deal in mathematics and computer science. Real numbers are unrepresentable ideals which may be approximated by finite binary approximations within a computer.

Sets in set theory and object types in object-oriented programming are the intellectual descendants of Platonic ideals. Set theory is concerned with classification of a universe of discourse into sets with uniform properties and with the description of sets by predicates that constitute necessary and sufficient conditions for set membership. Object types similarly describe application domains by classes with uniform properties and additionally support an inheritance mechanism for classifying classes.

Major attempts to provide a unifying framework for mathematics, such as Russell's theory of logical types and axiomatic set theory, may be viewed as attempts to characterize mathematics in terms of classification. The paradoxes of set theory are essentially paradoxes of classification which arise when methods developed for the classification of instances are extended to the classification of classes. The extension of classification methods from individuals to classes arises also in programming languages, where types that classify classes are called polymorphic types.

Polymorphic types extend the classification mechanisms of a programming language from values to types. They arise from a need to classify types as well as values. They reflect the fact that classification does not respect the boundaries between types and values and is applicable to entities of any kind whatsoever. When types are viewed as a mechanism for classifying values, then polymorphic types may naturally be interpreted as a mechanism for classifying classes. Inheritance is a special kind of polymorphism that classifies sets of "similar" types by factoring out their common attributes and providing a supertype name for the resulting polymorphic type.

2. State Transition, Communication, and Classification Paradigms

Models of computability developed in the 1930s and 1940s include the lambda calculus, combinatory logic, Turing machines, recursive function theory, and Post production systems. Of these the Turing machine is the closest to physical computers and is the basis of a paradigm for computing that we here call the state transition paradigm.

In order to understand classification as a paradigm of computing let's compare the role of statements, modules, and types in problem specification. These three linguistic mechanisms are respectively associated with the activities of state transformation, communication, and classification, and give rise to

paradigms for computing which may respectively be characterized as follows:

(1) state transition paradigm: execution of sequences of statements
examples: Turing machines, assembly languages, Fortran

(2) communication paradigm: passing messages among modules or objects
examples: actors, CSP, CCS

(3) classification paradigm: progressively constraining the class of the solution
examples: object-oriented, logic, constraint-based, relational specification

Each of these paradigms can, in its pure form, compute all computable functions. Programs could in principle be written entirely as statement sequences, as actors whose primary action is message passing, or as relations whose range is progressively constrained to a result value. But programs written in a pure paradigm such as Turing machines, the lambda calculus, or set theory, are generally difficult to understand, and programming languages generally provide some combination of pure paradigms as a basis for problem-solving.

How can we crisply compare and contrast these paradigms? One approach is to compare the interpretation of nodes and edges of typical computation graphs in each paradigm. In the state transition paradigm, programs are represented by "control flow graphs" whose nodes are actions (statements) and whose edges are control paths. In the communication paradigm nodes are modules and edges are communication paths. In the classification paradigm nodes are specifications of knowledge or behavior and edges represent relations among types such as inheritance. The difference of node interpretations (actions, modules, and behavior specifications) brings out very clearly the difference in focus of each paradigm. Since edges represent relations among nodes, this comparison also brings out the fact that the relations emphasized in each paradigm are respectively control-flow, message-passing, and behavior relations.

Object-oriented programming strikes a balance among state transition, communication, and classification paradigms. Early languages like Fortran emphasized statement-oriented programming. Ada provided a rich module specification and communication facility but a relatively weak typing facility that excluded procedures and packages from being first-class (typed) objects. Object-oriented languages strengthen the ability to use types for purposes of classification, thereby providing a better balance between classification and other mechanisms of problem solving.

In principle it is possible to view all computation as classification by viewing function evaluation as a process of progressively narrowing the domain in which a value must lie until a domain with a singleton element is obtained. In practice, programs are a balance between the descriptive specification of classes and the imperative evaluation of expressions and statements constrained to belong to particular, programmer-defined classes.

A pure calculus of classes requires primitives of very fine granularity, such as a class "bit" with logical operations, as a basis for building up classes that arise in practical programming. The development of such a calculus is a worthwhile research exercise but is beyond the scope of this paper. Any pure calculus starts with primitives of low granularity and requires intricate constructions to develop higher-level constructs. In developing practical programming languages it is usually better to start from a redundant set of primitives which in general includes expressions and statements, various forms of message passing, and a variety of classification mechanisms.

The term "calculus of classes" draws attention to the complementarity (duality) between the "calculus of communicating systems" [Mi] and the "calculus of classes" as twin requirements for programming in the large. Providing a computational framework for classification is comparable in its practical importance and research interest to providing a computational framework for communication.

3. Type Expression Sublanguages

The set of types of a programming language may be defined by a type expression sublanguage with a simple syntax.

Type → Basic-type | Constructed-type
Basic-type → Integer | Real | Boolean | String | ...
Constructed-type → Arraytype | Recordtype | Functiontype | Typevariable | ...
Type-definition → "Typedef" Type-variable "=" Type ";"

How should semantics for types be defined? What do we mean by the semantics of types? This question can be answered at a variety of different levels. A complete semantics requires mapping type expressions into denotations in a domain such as algebras or the lambda calculus. A partial semantics can be defined by specifying relations such as type equivalence or subtype relations to be satisfied by type expressions. Equivalence of the two record types "<x:Int,y:Real>" and "<y:Real,x:Int>", and relatedness of the two types "<x:Int>" and "<x:Int,y:Int>" are examples of such semantic relations among types.

Type equivalence in languages like Algol 68 gives rise to equivalence relations over type expressions that may be quite difficult to compute. Overloading gives rise to the dual situation in which a given token represents a syntactically defined equivalence class of semantically different types. Certain kinds of semantic relations among types, such as that between integers and reals, can be represented by coercion. These examples indicate that the expressiveness of a type system is determined not only by the syntactic richness of its type expressions but also by the relations among type expressions that it can express.

Semantic relations among type expressions are non-trivial even in traditional strongly typed languages but become much richer in object-oriented languages. Object-oriented programming extends the notion of type in the following respects:

(1) The class of things that can be typed is extended to data abstractions: That is, to collections of operations (and possibly other attributes) that may be related to each other by sharing a common data structure that persists between the invocation of operations.
Constructed-type → ... | Abstract-type

(2) Certain interesting relations among types (in particular inheritance) can be expressed in the language. Inheritance may be viewed as a form of type composition that creates a composite type from a subtype and its parent type.
Constructed-type → ... | Inherits(Parent-type, Subtype)

What is the significance of these two extensions of traditional type systems, both in general and in the context of classification? Abstract types provide additional raw power in defining important new kinds of types. Inheritance augments the calculus of classes by enriching the "flat" class structure of traditional type systems so that it becomes hierarchical. Moreover, inheritance determines relations between parent types and subtypes that greatly enriches the semantic relations among types of the type system.

The introduction of inheritance as a type classification mechanism may be compared to the transition in biology from Linnaean to Darwinian classification mechanisms. Like Darwinian inheritance, type inheritance enriches the expressive power of classification in the following ways:

(1) It extends the flat classification categories of traditional languages to tree-structured hierarchies.

(2) It determines relations among collections of subtypes, based on the fact that they share common attributes of the supertype. The relation may, as in the Darwinian case, be the result of evolution, but may also be the result of deliberate design.

An important similarity between inheritance in biological and programming systems is that both are designed to handle evolution in time. Inheritance is especially concerned with the management of change and it is expected that object-oriented systems will be particularly effective for languages, environments, and databases of applications with a very long lifetime. As a first approximation we can think of modularity as a mechanism for handling extension in space and inheritance as a mechanism for handling extension in time. Since object-oriented systems nicely integrate modularity with inheritance, they appear to be suited for handling applications with extension in both space and time.

Let's compare inherited with parameterized modules as a mechanism for describing a range of behavior. Parameterized modules (such as stacks parameterized by their element type) are closed specifications that parametrically determine their domain of variation at the time they are defined. In contrast, inheritable supertypes determine incomplete open systems that can be a behavioral component of a wide variety of as yet unspecified behaviors. An inheritable supertype is an incomplete specification that can be viewed as an approximation to a set of associated complete behaviors. The ways in which the incomplete behavior of a supertype may be completed are quite unconstrained compared to the predetermined set of behaviors of a parameterized module.

4. What We Mean by Object-Oriented

We define "object-oriented" so that it includes prototypical object-oriented languages like Smalltalk and Loops, but excludes languages like Ada, Modula, and Clu which have data abstractions but are generally not considered object-oriented. Support of data abstractions is a necessary but not sufficient condition for a language to be object-oriented. Object-oriented languages must additionally support both the management of collections of data abstractions by requiring data abstractions to have a type, and the composition of abstract data types through an inheritance mechanism:

object-oriented = data abstractions + abstract data types + type inheritance

This definition accords with the traditional use of the adjective "object-oriented". Ada, which supports untyped data abstractions (packages) and CLU, which supports typed data abstractions (clusters) but not inheritance, are not object-oriented according to this definition. It is the calculus of classes centered on typed data abstractions with inheritance rather than other attributes such as message passing that makes languages object-oriented.

Are we splitting hairs in failing to identify object-oriented programming with data abstraction and insisting that objects have types and inheritance, or is there some real benefit in imposing these conditions? We believe that the benefits are substantial but that, just as in the case of biological systems, inheritance comes into its own particularly for very large, long-lived systems. The state transition paradigm is suited to programming in the small, the communication paradigm to programming in the large, and the classification paradigm to programming in the very large.

Let's first examine the benefits of requiring data abstractions to be typed and then the benefits of the type inheritance mechanism. Untyped data abstractions, such as the packages of Ada, cannot be passed as parameters, appear as components of structures, or be assigned as values of variables. Once defined they have a useful functionality, but they cannot be managed as data within the language and must instead be managed by special-purpose language facilities like Ada generics or library facilities. The importance of managing collections of modules may be negligible in small or medium-size systems with less than 100 modules, but becomes vital as the number of modules and module types increases into the hundreds and thousands.

Inheritance in programming systems has an impact similar to that of Darwinian classification in biology. Just as Darwinian classification provided a systematic framework for describing relations among a variety of biological species, so inheritance provides a basis for describing relations among types. The reasons for relatedness among types and the mechanism for evolution may differ from those for biological species. But the type inheritance structures for management of diversity and adaptation are similar to inheritance structures of biological systems. The analogy between object-oriented and

biological systems is quite deep.

5. Classification Versus Communication in Object-Oriented Systems

Is message-passing or type inheritance the characterizing feature of object-oriented languages? It is tempting to characterize object-oriented programming in terms of message passing, since object-oriented specifications model computing at the level of passing messages among collections of objects rather than at the level of execution of expressions and statements. Object-oriented systems emphasize communication among objects rather than sequential statement execution, and messages are the basic mechanism for communication.

However, the term "message-passing" has several different meanings. The first object-oriented language, Simula, had coroutines, an asynchronous (quasi-parallel) form of message-passing in which the sender saves its state and goes to sleep and must be explicitly reawakened by a resume call rather than by an automatic reply from the receiver. Smalltalk and Loops equate message-passing with remote procedure calls - a synchronous form of message-passing in which the sender must wait for a reply from the receiver before continuing. Modules in distributed systems may communicate by rendezvous, which combines remote procedure call with synchronization between the calling and called processes, by asynchronous message-passing, or both.

Objects certainly need to communicate, but any form of message-passing appears to be compatible with object-oriented programming. The precise nature of the mechanism for communication is not central to the definition of object-oriented programming. The assertion that message-passing is basic to object-oriented programming appears to be nothing more than an assertion that objects must be able to communicate, a very weak and obvious requirement. In contrast, the nature of the type mechanism does appear to be a distinguishing feature of object-oriented systems. The requirement of typed data abstractions with inheritance is explicit and definitive, suggesting that object-oriented programming should be characterized by the nature of its type mechanisms rather than by the nature of its communication mechanisms. Object-oriented programming is prescriptive in its methodology for organizing knowledge domains but is permissive in its methodology for communication.

Mechanisms for message-passing in object-oriented and distributed systems are concerned with communication rather than classification. One of the themes of this paper is that concerns of classification are complementary (orthogonal) to those of communication. In particular, calculi of classes for programming languages are largely independent of calculi of communication. There clearly needs to be some coordination between the two (an interesting research area) but the interconnection appears relatively weak. The object-oriented programming paradigm is distinguished from other paradigms by its highly developed calculus of classes and is compatible with a variety of calculi of communication.

6. Should Types be Modelled by Calculi or Algebras?

Should types be modelled by a calculus as in the lambda calculus models of [CW] or by an algebra as in the models of [FGJM]? In order to address this question we examine the meaning of the terms "calculus" and "algebra" and the relation between these terms?

Newton and Leibniz used the term "calculus" in describing the differential and integral calculus. More recently the term has come to mean a set of rules for reasoning as in the predicate calculus or a set of rules for computing as in the lambda calculus. The term algebra derives from a 9th century book by Al Khowarizmi whose contributions to both algebra and calculus were acknowledged by naming the concept of algorithm after him. Al Khowarizmi's concept of an algebra as a collection of reduction rules is close to the modern notion of a calculus. The term "algebra" is nowadays used by mathematicians to denote an abstract characterization of behavior and is therefore closely related to the notion of a type. Algebras consider the properties of operations on integers such as addition and multiplication and of algebraic structures such as semigroups, groups, rings, and fields. Universal algebras generalize this notion of algebra to arbitrary collections of operations on a set S. Multisorted algebras are a further generalization to operations whose domain and range are chosen from a collection of sorts S1,S2,...,SN.

Data abstractions provide a mechanism for the programmer to define algebras with programmer-defined sets of operations and may be modelled by multisorted algebras.

A "calculus" may be a formal system such as the predicate calculus where computation consists of applying rules of inference to prove theorems. It may be a reduction system such as the lambda calculus where computation consists of reducing expressions to a normal form in which no further reductions are applicable. The notion of calculus that emerges from these examples is that of a syntactically defined set of computation or inference rules that are unidirectional and terminate when a goal is reached (in theorem proving) or no further rules are applicable (in reduction systems). Thus 3+4*5 is reduced to 3+20 and then to 23 which cannot be further reduced. Similarly the differential calculus reduces expressions to eliminate differentiation operators and the lambda calculus reduces expressions to eliminate operator-operand combinations (redexes).

Whereas a calculus is a syntactic system of transformation rules an algebra is a semantic system whose behavior can be realized by a variety of syntactic calculi. For example the algebra of integers can be realized by decimal or binary number systems or even by lambda calculus representations. An algebra may be thought of as an equivalence class of calculi with common behavior or as an abstraction from a specific syntactic realization of a calculus to a semantic specification of an underlying behavior. For example the algebra of integer domains captures most of the important properties of the integers but also has non-standard models that behave differently from the integers.

In some contexts we have a choice between specifying a computation as an algebra or a calculus. For example, in the database world relational query languages may be specified either by a relational algebra or a relational calculus. Relational algebras express queries by applying operations such as union, product, or projection to a relation while relational calculi specify queries by predicates that determine a condition to be satisfied by tuples that answer the query.

This example illustrates that concrete algebras with syntactically specified concrete operations are effectively calculi. This is true for binary or decimal representations of integers. Conversely calculi such as the lambda calculus may be specified by concrete operations of an algebra. The specification of calculi by algebras has led to algebraic characterization of mathematical logic by lattice-structured sets of formulae as exemplified by "The mathematics of metamathematics" [RS].

The relation between calculi and algebras may be summarized by saying that calculi are concrete (syntactic) algebras while algebras are abstract (semantic) calculi. This statement is at first confusing, paticularly since the algebra of Al Khowarizmi and grade school algebra are essentially calculi. However, viewed as a mathematical statement we are simply saying that an algebra is a model of a calculus in the model theoretic sense, and conversely that a calculus is a concrete realization of some behavior it is trying to model. Moreover, a given calculus generally has many semantic models and conversely a given algebra can generally be realized by many calculi.

How do these general notions of calculus and algebra apply to the modelling of types? In object-oriented languages classes are computational objects with computation rules that determine a calculus of classes. But from the user point of view types specify abstract forms of behavior that may be specified by algebraic structures. The domains and ranges of operations are the sorts of the algebra and equations (axioms) constrain operations to have the desired behavior. The integers are the quintessential algebra that provided mathematicians with a paradigm for the development of abstract algebras and computer science with a motivation for calculi for computing.

The above discussion contributes to our understanding of how types should be modelled by clearly establishing that algebras specify behavior while calculi specify rules for computing. Since individual types are forms of behavior they are appropriately specified by algebras. However, relationships among types and computational properties of a calculus of classes are appropriately specified by calculi. Multisorted algebras provide a basis for modelling properties of individual types while the second-order lambda calculus provides a basis for modelling properties of the type system as a whole, such as polymorphism. Relations among types such as inheritance may be modelled by features of the second-order lambda calculus such as bounded quantification [CW]. Further details on the relation between algebraic and lambda calculus models of type may be found in [BW].

7. What is a Type?

This is a difficult question to answer because a complete answer must characterize aggregate properties of the type system as a whole as well as properties of individual types. We were tempted to replace the question by "What is a type system?". Instead of such rephrasing we choose to interpret the original question as an abbreviation for the longer question: "What properties must types have both individually and collectively to constitute an acceptable type system for an object-oriented programming language?".

The following collection of partial answers illustrates the variety of levels at which an answer may be formulated.

(1) Application programmer's view:
Types partition values into equivalence classes with common attributes and operations. Polymorphism allows types to have overlapping value sets so the partition (equivalence relation) becomes a covering (compatibility relation) of the universal value set.

(2) System evolution (object-oriented) view:
Types are behavior specifications (predicates) that may be composed and incrementally modified to form new behavior specifications. Inheritance is an important mechanism for incrementally modifying behavior that supports system evolution.

(3) Type-checking view:
Types impose syntactic constraints on expressions so that operators and operands of composite expressions are compatible. A type system is a set of rules for associating a type with every semantically meaningful subexpression of a programming language.

(4) Verification view:
Types determine behavioral invariants that instances of the type are required to satisfy.

(5) System programming and security view:
Types are a suit of clothes (armor) that protects raw information (bit strings) from unintended interpretations.

(6) Implementer's view:
Types specify a storage mapping for values.

These definitions provide some insights but none of them attempts a complete definition. This is due in part to the fact that the notion of type, just as the Platonic notion of "tablehood" or the notion of "computable function", is a primitive notion not completely definable in terms of other primitive notions. Plato, in his theory of ideals, discarded the idea of defining tablehood in terms of a necessary and sufficient set of defining attributes and sensibly settled instead for a definition in terms of an "ideal" table in heaven. Plato's difficulty becomes evident by considering the simpler problem of finding even a single necessary attribute of the set of all tables. Tables need not be flat, or have four legs, or even provide a surface on which objects can be placed.

Philosophers call primitive notions not expressible in terms of other notions "natural kinds". The existence of natural kinds places a limit on the expressive power of object-oriented definition since natural kinds are not completely definable by finite sets of operations or attributes. These limitations are exemplified by the inability to define the class of computable functions by an explicit set of properties and by the fact that Church's thesis that Turing machines capture the intuitive concept of computability is inherently unprovable. It would be nice to have a paradigmatic mechanism such as Turing machines to capture the notion of "typehood". This is not possible because the properties and computational power that should be possessed by type systems has not been agreed upon.

Another example of definitional limitation arises in trying to define the discipline of computer science for the benefit of funding agencies or university administrators. Definitional difficulties stem not from the fact that computer science is an ill-defined discipline but from the inherent impossibility of

such definitions (mathematics is no easier to define). Explicit recognition of the theoretical impossibility of such definitions is long overdue.

Whereas it is reasonable to define technical classes such as integers or even Toyotas in terms of a finite set of applicable operations, it is unreasonable to define non-technical concepts in this way. However, our attempt to define the concept of type illustrates that, even when a precise and complete definition is unattainable, the attempt to approximate a concept by multiple partial views provides invaluable insights into the concept.

8. God and Plato

In trying to understand notions of classification we are constantly led back to fundamental issues in mathematics and philosophy. It was no accident that we encountered Plato's theory of ideals in exploring the limitations of object-oriented specification. In the present section we examine some additional foundational issues and again succumb to the temptation of philosophical analogy. We relate the duality between classification and communication to God and Plato, and compare realist and intuitionist views of types to the philosophical positions that "observability depends on existence" and "existence depends on observability".

The "classification" view of systems can be likened to that of the creator of a domain of discourse who must know all interfaces, be omniscient, and have a global perspective. In contrast, the "communication" view can be likened to that of Plato's cave dwellers who can interact with the universe in which they live only in terms of observable communications, represented by reflections on the walls of their cave.

The classification perspective is that of God the creator while the communication perspective is that of Plato the cave dweller. A system for programming in the large must support both the viewpoint of the creator/designer and the viewpoint of individuals who will populate the system after its creation. It must support the viewpoint of both God and Plato. In addition, Platonic cave dwellers must be allowed to play God with respect to the subsystems they create and the God of any given domain of discourse may be a Platonic cave dweller in some larger universe.

The need to provide an ability for a given person to switch flexibly between the roles of God and Plato is a persuasive argument for providing integrated design and implementation facilities within a single system rather than segregating the design and implementation sublanguages in different subsystems. The ability to provide an integrated approach to design and implementation is in fact one of the strengths of object-oriented programming.

Should multiple religions be allowed to flourish (multiple views are equally legitimate) or should a single true religion be imposed (the creator's view encompasses all other views)? This question relates to practical questions such as whether design and implementation languages should be separate or part of a single integrated system. Reality is probably closer to a single true religion since there is in fact a global reality to which the local view of the Platonic cave dweller is subsidiary. However, Platonic cave dwellers have the illusion that their cave is the whole universe and are, by hypothesis, unaware of the global reality outside their cave. There is thus a need for local religions for each clan of cave dwellers, and software methodology should ensure that such religions be cooperative rather than antagonistic.

In conclusion we examine some philosophical implications of the notion of type in constructive (intuitionistic) type theory [Ml, Co]. Constructive types may be identified with propositions and constructive values with proofs that provide evidence for the truth of propositions (witnesses for the existence of elements of the type). This view is particularly fruitful because, somewhat unexpectedly, type constructors for products, variants, and functions, can be mapped into the propositional connectives and, or, implies, allowing reasoning about composite types to be conducted in terms of corresponding propositional formulae.

Let's consider the current philosophical debate among realists and intuitionists concerning the ontological status of types and values. The realist view is that we live in a world populated by values and that types are an explanatory mechanism introduced for the purpose of classifying and managing

values. The intuitionist view is that types are the basic conceptual entities of the domain of discourse and that values exist only in so far as they are constructible from some type. Thus the realist postulates the existence of a global universe created by God while the intuitionist requires existence to be demonstrated by observability or constructibility. The realist views God as an explanatory framework introduced by inhabitants for describing and managing preexisting phenomena. Namely, that man made God in his/her own image. The intuitionist does not really believe in God but believes that natural phenomena as well as artificial mathematical and computational entities exist only when they are observed, constructed, or interpreted.

This debate has its counterpart in the natural sciences, where realists assert that observability depends on existence while operationalists (constructive scientists) assert that existence depends on observability. This dichotomy is illustrated by Eddington's fish-net experiment; the observation that all fish are at least two inches in diameter can be explained either as a law of the universe or as a consequence of the fact that nets used for fishing have a mesh that is two inches wide.

Newton, whose notions of space and time presuppose the existence of a universe whose laws we are trying to explain, was a realist. Theories of relativity and quantum mechanics are formulated in terms of the ability of the observer to observe, and are positivist (operational, behaviorist) in their orientation.

Do philosophical questions at the level of realism versus intuitionism make a practical difference? In physics the answer is an unqualified yes, as exemplified by experiments that demonstrate the greater explanatory power of relativity over Newtonian mechanics. Intuitionists also claim that the answer is yes in the domain of computer science, but the difference has not yet been as clearly pinpointed.

The intuitionist view of propositions as types and the converse view of types as propositions provides a basis for a propositional calculus of classes whose relation to traditional classification has yet to be fully understood. Exploration of this relation is one of many challenging research problems.

9. References

[BW] K. Bruce and P. Wegner, An algebraic model of subtypes in object-oriented languages, Brown Universty and Williams Reports, May 1986.

[Co] R. L. Constable et al, Implementing mathematics with the Nuprl proof development system, Prentice Hall 1986.

[CW] L. Cardelli and P. Wegner, On understanding types, data abstraction, and polymorphism, Computing Surveys, December 1985 (actual publication in August 1986).

[FGJM] K. Futatsugi, J. Goguen J. Jouannaud, and J. Meseguer, Principles of OBJ2, Proc POPL 85.

[Mi] R. Milner, Calculus of communicating systems, Lecture Notes in Computer Science 92, Springer Verlag, 1980.

[Ml] P. Martin-Lof, Constructive mathematics and computer programming, in Mathematical logic and programming languages, eds. Hoare and Shepherdson, Prentice-Hall International, 1985.

[Ma] E. Mayr, The development of biological thought, Harvard University Press, 1982.

[RS] H. Rasiowa and R. Sikorski, The mathematics of metamathematics, Mathematical Monograph 41, Polish Academy of Sciences, 1963.

[We] P. Wegner, Classification as a paradigm for computing, Technical Report CS-86-11, Brown University, May 1986.