Type Theories and Object-Oriented Programming

SCOTT DANFORTH and CHRIS TOMLINSON

Systems Technology Lab, Advanced Computer Architecture Program, MCC, 3500 West Balcones Center, Austin, Texas 78759

Object-oriented programming is becoming a popular approach to the construction of complex software systems. Benefits of object orientation include support for modular design, code sharing, and extensibility. In order to make the most of these advantages, a type theory for objects and their interactions should be developed to aid checking and controlled derivation of programs and to support early binding of code bodies for efficiency. As a step in this direction, this paper surveys a number of existing type theories and examines the manner and extent to which these theories are able to represent the ideas found in object-oriented programming. Of primary interest are the models provided by type theories for abstract data types and inheritance, and the major portion of this paper is devoted to these topics. Code fragments illustrative of the various approaches are provided and discussed. The introduction provides an overview of object-oriented programming and types in programming languages; the summary provides a comparative evaluation of the reviewed typing systems, along with suggestions for future work.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs—*abstract data types*; *data types and structures*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*type structure*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*lambda calculus and related systems*

General Terms: Languages, Theory

Additional Key Words and Phrases: Data abstraction, inheritance, object-oriented programming, polymorphism, type checking, type inference

INTRODUCTION

When implementing a system, an important initial decision concerns how the system should be perceived—What are its parts and how do they interact? One point of view shown to be successful in designing and implementing complex software systems is suggested by object-oriented programming (OOP) in which systems are constructed from self-contained objects that interact via messages.

Proponents of OOP suggest that it aids design, implementation, and maintenance of complex systems by supporting modularity and that it aids code reuse and the construction of easily extensible systems by supporting inheritance. In addition to these benefits, object orientation may allow designs in which objects reflect opportunities for variable-grain parallelism and in which decisions related to whether objects are implemented in hardware or software may be postponed or flexibly changed.

CONTENTS

These are all powerful advantages, and they may account for a recent upsurge of interest in OOP as well as a number of new OOP language implementations [Black et al. 1986; Cox 1984; Lang and Perlmutter 1986; Moon 1986; Schaffert et al. 1986; Stroustrup 1986]. In order to make the most of these advantages, however, it is necessary to develop a type theory for OOP that is able to support checking and controlled derivation of systems, while aiding efficient implementations.

As a step in this direction, this paper surveys a number of existing type theories, examining the manner and extent to which these theories are able to represent the objects and object interactions that arise in OOP. To introduce the characteristics of OOP important to this objective, a cursory review of OOP follows. Readers wishing a more detailed introduction to OOP may refer to Stefik and Bobrow [1986]. Also, a collection of papers edited by Shriver and Wegner [1987] provides a good overview of current research directions in OOP and includes papers describing various OOP languages. Additionally, the proceedings of the first two annual conferences on OOP systems, languages, and applications [Meyrowitz 1986, 1987] provide a wealth of information concerning the currently developing theory and practice of OOP.

What Is OOP

Many of the ideas associated with OOP originated with the Simula language of Dahl and Nygaard [1966]. These ideas were refined and extensively developed during construction and standardization of Smalltalk [Goldberg and Robson 1983; Ingalls 1978; Kay 1972], the first substantial, interactive, display-based OOP implementation. Of special interest in the context of systems involving parallel or distributed execution are the actor languages of Hewitt [1985] and the various models of cooperation among objects suggested by this work. There are a number of recent efforts in the area of parallel OOP [America 1986, 1987; Dally 1986; Ishikawa and Tokoro 1986; Yonezawa and Tokoro 1987; Yonezawa et al. 1987].

Object-oriented programming, like functional programming or logic programming, incorporates a metaphor in which computation is viewed in terms divorced from the details of actual computation. In the case of OOP, this metaphor is rarely introduced with the mathematical precision available to the functional or logic programming models. Rather, OOP is generally expressed in philosophical terms, resulting in a natural proliferation of opinions concerning exactly what OOP really is. The following introduction also presents a view of OOP that, by leaning to the philosophical side, permits considerable latitude of interpretation. Different interpretations represent the potential for different OOP languages.

The heart of the OOP metaphor is an anthropomorphic view of the objects of computation. Simply stated, the term "object oriented" is used to describe programming languages in which the objects of computation are (in a sense) like people. In OOP, objects generally have an identity, called the *self*, that persists over time independently of changes in the state of the object. Objects are intelligent and respond

to requests addressed to them (generally called *messages*), thus effecting computation. In terms of the model, an object's response to a message might be to change its internal state, send messages to other objects, reply with an answer, create new objects, or all of these. A prescription for handling a message is generally called a *method.*

In order to compute with objects, they must be brought into existence. There seem to be essentially two ways of going about this. One is to create new objects by using objects already in existence as *prototypes.* If this mechanism is used, it must also be possible to create objects "from whole cloth" by specifying a set of methods and the variables that are used to hold the object's state. The more usual approach is to specify the *class* of the new object. A class may be thought of as a template that identifies the methods and *instance variables* to be used by the new object for handling messages and storing object state, respectively. Initialization of an object's state can be supported by providing a method for this, but many class-based OOP languages consider a class itself to be a kind of object that can respond to a message requesting a new object of its class. This message is then used both to request creation of a new object and to specify the desired initial state; the class's response is a reply containing or referencing the new (initialized) object. In this view, classes are similar to objects in that they respond to messages. Thus they may also have a class, called a *metaclass.* The desire to treat *everything* as an object in OOP appears to be the primary justification for such an approach. Parsimony of mechanism is often a desirable goal in languages, but many approaches to OOP avoid metaclasses and some, based on prototypes, avoid classes as well [Lieberman 1986].

The two OOP concepts on which we focus with respect to type theories are abstract data types (ADTs) and inheritance. Objects encapsulate a state, along with methods for dealing with this state, and thus provide data abstraction through their message interface. An object may therefore be viewed and used in a way entirely analogous to an ADT in traditional programming languages. In attempting to provide a less philosophical statement of the OOP metaphor, many researchers have found this similarity between objects and ADTs an obvious and important area for careful attention. Indeed, ADTs as well as OOP emerged from the Simula language.

If ADTs are so similar to objects, is it not possible that OOP is simply a programming model in which all data are abstract and all data manipulation is implemented via ADT operations? In fact, this is not an unreasonable conjecture, since few languages outside OOP provide such comprehensive support for data abstraction. Even in Ada, an advanced language incorporating extensive support for various forms of abstraction, packages are not first-class citizens that may be passed as parameters [U.S. Department of Defense 1983]. However, there is more to OOP than support of ADTs as first-class citizens—in particular, inheritance.

Inheritance in OOP may be based either on the concept of *subclass* (in those OOP languages with classes) or on default delegation of responsibility (in those OOP languages based on prototypes). The essential idea is enhancement of descriptive power with respect to object creation, during which the methods to be used by the new object for handling various messages must be somehow indicated. In class-based languages, inheritance allows the methods used by objects of a given class to be specified in a modular and extensible fashion by placing logically related methods in individual classes and relating these classes within a subclass hierarchy. Then the methods potentially available to an object of a particular class are not only those methods defined in the object's class but also those of all its class's ancestors within the class hierarchy. For instance, the class of automobiles could be defined as a subclass of the class of vehicles, and an automobile object might then answer "yes" to the message, "Are you a vehicle?" because the vehicle class provides this method. The implications for software maintainability become clear when one considers adding a new method for all vehicles—only the vehicle class need be changed; the automobile class (and any other classes inheriting

directly or indirectly from vehicle) need not be modified. Similar benefits may be arranged in prototype-based OOP languages through use of delegation [Stein 1987].

Elaborations on the above occur in most OOP languages. In some OOP models the class of an object is allowed to change over time. This appears to be useful in object-oriented database systems [Skarra and Stein 1987]. In some OOP languages, multiple inheritance is allowed [Moon 1986; Schaffert et al. 1986]. In class-based OOP, this takes the form of allowing a given class to have more than one immediate ancestor (called a *superclass*) within the class hierarchy, which allows methods from a number of otherwise unrelated classes to be combined and used within a single object. This facility can be useful in achieving modularity and extendability. In prototype-based OOP, multiple delegation paths can achieve the same effect. Class-based OOP languages generally allow inheritance to be "defeated" by either forgetting or overriding methods that would otherwise be inherited from a superclass. A **sendsuper** facility is often provided to allow invoking a superclass method explicitly when inheritance has been defeated. This is useful when the behavior provided by a superclass method is to be enhanced incrementally by addition of special behavior supplied by a subclass.

Clearly, a wide variety of OOP languages is possible within the general framework described above. Although attempts at providing a formal type system for OOP may currently fail to address all the requirements of any particular OOP language, focusing on integrating type models for ADTs and inheritance addresses an important intersection of capabilities.

What Are Types

Types in programming are generally used and thought of as a means of characterizing values that arise dynamically in the course of a computation. For instance, a value that is to be computed by a program may be represented by a name or an expression, and although the particular value to which this expression refers may not be known in advance, other information concerning the value might be available. This information could be an indication of the *meaning* of the expression in an "approximation" semantics, or it might be considered a *constraint* describing a property that the value must have . Given the latter view, it is a natural step to conceptualize the set of all values satisfying the constraint and then to think of the type constraint as simply requiring membership in this set.

Viewing types as constraints leads naturally to the idea of types characterized by logical formulas. This is the approach taken in work directed toward supporting polymorphism with universal quantification [MacQueen et al. 1984] and supporting abstract data types with existential quantification [Mitchell and Plotkin 1985]. Recently, Cardelli and Wegner [1985] used these ideas as the base for a polymorphically typed lambda calculus language with support for both abstract data types and inheritance [Cardelli and Wegner 1985]. Also recently, a great deal of interest has been shown in dependent types, which use this "formulas-as-types" philosophy [Constable and Zlatin 1984; Hook 1984; MacQueen 1986]—the formulas in this case being borrowed from constructive logic [Martin-Löf 1982; Turner 1984].

The algebraic approach to types is set oriented; in an algebra, a type (or sort) is a set of individual elements upon which operations of the algebra are defined. Other set-oriented approaches are possible, though. Matthews [1983] considers types to be represented by a set of operations rather than a set of values. Thus, in Matthews' language, Poly, the type **boolean** is not characterized by a set of possible values, such as {T, F}, nor by constraints that define this set but rather by the fact that the operations in the set {**and**, **or**} are guaranteed to be available for operating on values of the type. Donahue and Demers pioneered this approach in their language, Russell [Demers and Donahue 1983; Donahue and Demers 1985]. In Russell, a data type is a collection of named operations that provides a consistent interpretation of a single, universal value space. In spite of the similarity of this approach with

OOP, Poly and Russell do not currently support inheritance.

The typing systems of interest in this paper may be divided into two general categories: those based on an algebraic model of computation, in which there is a strong distinction maintained between values and operations, and typing systems based on higher order models within which functions and even types themselves may be viewed as values (which have types). The algebraic approach has been traditionally used to represent ADTs [Guttag 1980], but it is only within the last few years that higher order theories have been used for this purpose.

Potential Benefits of Typed OOP

There are many potential advantages to be derived from successfully embedding the essence of OOP into a well-grounded type theory. Both types and objects provide a uniform framework within which to understand the entities of a programming language, but a typing theory can be useful in other ways. Especially within declarative languages, where denotations are the means of guiding computation, it is advantageous to have a powerful theory capable of explaining and representing the meanings of program expressions. Such a theory can guide equivalence-preserving program transformations and assist in the program development process.

In general, the objective of high-level programming languages should be to provide programmers with as much descriptive power as possible in order to aid construction of useful and understandable software, while allowing the efficient utilization of underlying hardware. Regarding efficiency, information concerning which objects communicate with other objects can be invaluable in achieving locality of access in parallel systems and can also be useful in load balancing. Optimizations made possible by typing include folding multiple levels of data hierarchy into a single structure [Scherlis 1986] and integration of operation invocations in order to avoid procedure calls [Johnson 1986]. A typing system can help support efficiency objectives, provide a language framework capable of guiding a system designer's conceptualizations, and verify (often statically, before execution) the consistency of the descriptive information provided explicitly and implicitly by a program.

Also possible is a natural connection between the formal, extensional objects represented by a program and their concrete, intensional representations within memory or a file system. This connection provides opportunities for type-secure separate compilation, an important aid to the software engineering of large systems [MacQueen 1986], and opens the way for persistent objects capable of independent existence between program executions [Cardelli and MacQueen 1985].

Perhaps all these things could be accomplished in an ad hoc fashion through the use of special annotation and supporting mechanisms. However, such an approach might result in unexpected interactions between solutions to separate problems and conceptual complications for system designers and language designers alike. Certainly, the simplicity of a single typing theory that provides a consistent and flexible framework for system descriptions at the outset is preferable—assuming that the theory is able to provide straightforward representations for the objects, behaviors, and computations of interest. Such a theory should aid the construction of an integrated environment especially tailored for the description and construction of systems.

Our Review Approach

This paper reviews a number of typing systems that have been recently proposed by researchers in the area of type theory. These researchers have been concerned with formal rigor and with capturing, within accepted logical/mathematical systems, their basic intuitions concerning types in programming languages. Unfortunately, there seems to be almost no limit to the variety and mathematical sophistication of these theories—ideas from universal algebra, second-order lambda calculus, and constructive mathematics have all been used to represent and formalize these

intuitions. We do not attempt to provide deep insight into these formal systems, but restrict ourselves in our review to the essential nature and practical implications of these typing systems, with the primary goal of determining how well they support ADTs and inheritance, two capabilities that are central to OOP.

It is interesting to note the tension between the two foci of interest identified above. Data abstraction attempts to provide an opaque barrier behind which methods and state are hidden; inheritance requires opening this interface to some extent and may allow state as well as methods to be accessed without abstraction [Snyder 1987]. OOP languages (and the type theories we review) take various stands on this issue, both with respect to its importance and its resolution.

1. ABSTRACT DATA TYPES

In the Introduction types were presented as characterizing values that arise in the course of a computation. Types in this informal sense have been used since the first FORTRAN compilers, in which type information supported decisions concerning whether to emit floating-point or integer operation codes for arithmetic computations. Types thus originated because the information they provided was useful to the compiler. The development of abstract data types, on the other hand, was due to different pressures.

Concern with programming methodology and the need for reliable and maintainable software resulted in awareness of the importance of abstraction in program design and construction. Two results of this awareness were the introduction of various control abstractions[1] and the introduction of data abstractions, or abstract data types. ADTs were thus an important step in the development of programming methodology.

An ADT encapsulates information that provides a representation of a complex data object, such as a stack, and provides operations that implement the manipulations of which the object is capable. The internal representation of the object represented by an ADT is completely hidden from its users, and only the operations that implement manipulations of this representation are made available to the object's user. An ADT's user need not know how the object it represents is implemented, just as the user of a control abstraction such as a square-root routine need not know how the square root is calculated. In addition to the intellectual leverage for programmers, who can take bigger strides in their thoughts, it provides flexibility in modifying the ADT implementation. As long as the external interface remains the same, any code that uses the ADT should continue to work unchanged.

The kind of abstraction provided by ADTs can be supported by any language with a procedure call mechanism, given that appropriate protocols are developed and observed by programmers. Because of the importance placed on maintainable and modifiable software, however, many languages have attempted to enforce such protocols by making them part of the language definition. Violations of ADT security are then made impossible, and the use of ADTs is made easier because the language directly supports them. Such languages include Concurrent Pascal [Brinch-Hansen 1977], CLU [Liskov et al. 1977], and Ada [Department of Defense 1983].

Type checking in these languages has not been based on any foundational theory giving meaning to ADTs—thus the challenge to researchers in programming language semantics, who seek to put programming on a firm foundation with useful semantic theories. To these researchers, a program and the objects represented therein should have unambiguous values in some semantic domain. Types provide a way of representing such meanings. If we characterize the traditional objects of computation (numbers, characters, etc.) as elements of some semantic domain (or type), what is the corresponding explanation of ADTs? There are a number of possibilities.

[1] Control abstractions initially included subroutines and procedures and, later, with the introduction of structured programming, constructs such as **if-then-else** and **while**.

1.1 Algebraic (First-Order) Approaches

In the last 10 years, various algebraic ideas have been useful in the areas of compiler correctness, specification, and abstract data types [Burstall and Goguen 1982]. Among others, Goguen has helped develop this connection between theory and practice; he has put forth both substantive language suggestions and a variety of results concerning the theoretical support that algebras can provide for various aspects of programming. A recent paper describing OBJ2, a language showcase for these ideas, cites many references in which this work appears [Futatsugi et al. 1985].

An algebra is a formal, mathematical entity, essentially composed of sets of values (called the *carriers* of the algebra) and operations for manipulating these values. As can be seen, this formal entity corresponds closely to the concept of an ADT. In defining an algebra, one begins with a *signature*, which is a declaration of the types, constants, and operations of interest. In terms of programming languages, a signature can be viewed as an interface specification.

Figure 1 is an example of the kind of information provided by a signature. As shown, constants are represented as constant functions (i.e., they have no arguments and evaluate to the desired value).

A signature, therefore, introduces the names that will be used to refer to types (also called sorts), values, and operations of the algebra. These are only names, however, and the remaining task is to give the names meanings. This is done in the following way: Types are associated with carrier sets (i.e., value domains are defined), and operators are associated with functions (i.e., the functions that implement the operators are defined). Defining these sets and functions can be an interesting theoretical exercise when the sets involved are infinite, but this need not concern us here; we merely wish to introduce the algebraic approach to ADTs. Figure 2 therefore presents a simple algebra of numbers modulo 2.

Note that although MOD2 successfully represents an abstraction of numbers modulo 2 and operations on these numbers, it cannot directly encapsulate a mod2 number; that is, there is no concept of state. The style of programming that results is one in which the implementation of an ADT does not store data but is used to manipulate data. Thus, the user of a stack ADT might ask an ADT interface for a new stack, and the representation of an empty stack would be returned to the user as an abstract, atomic value, instead of being held behind the ADT interface as in OOP. When the user wants to put something on the stack, the data item and the stack representation are passed to the ADT interface, which will return the appropriate resulting stack state, again to be held by the user. Thus, the user holds ADT objects but makes calls on the ADT implementation in order to use them.

```
number, bool: type;

zero, one: function() result number;
plus, times: function(number, number) result number;

true, false: function() result bool;
and, or: function(bool, bool): result bool;
```

Figure 1. A signature for an algebra.

One can take the viewpoint that the essential concept supported by ADTs is not support of state itself but abstraction of data representation and operations. If this is so, then algebras directly support ADTs. If, on the other hand, one takes the view that ADTs are closer in concept to abstract machines that embody a state as well [Parnas 1972], then the above algebraic approach to ADTs is only partially satisfactory in supporting ADTs. The practical advantages of data abstraction are obtained in either case, as long as the state held by the user of the ADT interface is always treated as being abstract by the ADT user.

Goguen has developed an interesting approach to supporting state within ADTs [Goguen and Meseguer 1987; Meseguer and Goguen 1983]. He suggests the use of *reflective* programming, in which OBJ2 is used to define itself and a hidden type is used to represent the necessary internal

MOD2 Signature:
number: type;
zero, one: function() result number;
plus, times: function(number, number) result number;

MOD2 Semantics:
number == {**0**, **1**}
zero() == **0**
one() == **1**
plus(x,y) == if (x=**0**) then y; else if (y=**0**) then x; else **0**.
times(x,y) == if (x=**0** or y=**0**) then **0**; else **1**.

Figure 2. MOD2: A simple algebra.

state space. The basic idea he develops is that the underlying algebraic theory being used can be modified as a result of expression evaluation. In the above example of MOD2 numbers, for instance, a constant called *my_number* might be used to hold a number within an ADT. When this number needs to be changed, the underlying algebra is modified so *my_number* represents the new value. This may seem overly elaborate, but perhaps it is a reasonable approach if one wishes to stay within an entirely algebraic framework. The approach is analogous to the use of behavioral replacement to model state in Actor languages [Agha 1986].

The main restriction of the algebraic approach is that it is first order; that is, functions (and also ADTs) are not first-class values—they cannot be passed as input to other functions, returned from functions, or stored within data structures such as records. Goguen does not consider this a problem and claims that the absence of higher order capabilities in OBJ2 enhances opportunities for efficient implementation. We do not necessarily agree with this and feel that higher order capabilities can be efficiently supported while providing a greatly enhanced degree of expressive power within the language. The ability to create and manipulate ADTs as data at run time appears to us to be an extremely important ability within a program development environment, and within systems in general.

1.2 Higher Order Approaches

MacQueen [1986] gives a good review of the origins of type theories germane to this section. Historically, they are based on the "formulas-as-types" notion that evolved through the work of Curry and Feys [1958] in which a close correspondence between axioms of propositional logic and basic combinators was observed. The notion was used by Girard [1971], who introduced a form of second-order typed lambda calculus as a tool in his proof-theoretic work, and by Reynolds [1974], who independently invented a programming language that has come to be called the second-order lambda calculus.

1.2.1 Mitchell and Plotkin's Existential Types

Mitchell and Plotkin [1985] used an extended version of the second-order lambda calculus, called SOL, to represent ADTs. The primary focus of this work was expressing the *type* of ADTs. This can be contrasted with a purely algebraic characterization, in which types (i.e., sets of value domains) remain on a completely different semantic plane from ADTs (i.e., algebras).

Mitchell and Plotkin [1985] begin by acknowledging the algebraic model of ADTs and consider the concrete representation (i.e., implementation) of an ADT to be a *data algebra* of the kind described in Section 1.1. Their next step, however, is to show how data algebras can be considered typed values within SOL. The result is that data algebras may then be considered first-class values that can be passed as parameters to functions or returned as results. Practical implications of this include the ability to examine data at run time in order to choose an appropriate ADT representation (e.g., a sparse matrix) and then call a data processing function with the desired

```
abstype point with
        create: function(real, real) result point;
        plus: function(point, point) result point;
        x_val: function(point) result real;
        y_val: function(point) result real;
is
representation real × real
        create(x,y) == cons(x,y);
        plus(c1,c2) == cons(car(c1)+car(c2), cdr(c1)+cdr(c2));
        x_val(c) == car(c);
        y_val(c) == cdr(c);
in
        point.x_val(point.plus(point.create(1,4),point.create(2,3)));
```

Figure 3. A SOL ADT for points.

ADT representation as a parameter. A conditional statement may be used to return one of two different ADT representations as its result.

The central idea of SOL that supports this ability is that of existential types. Existential types provide an appropriate mechanism for expressing the types of data algebra expressions; they tell about the available operations and how they may be used, without describing the implementations of the operations or the type used as the carrier of the algebra. Figure 3 shows a SOL expression in which an ADT implementing geometric points is used. The value of the overall expression is 3.

The expression in Figure 3 was built from an interface specification, a data algebra expression, and a body within whose scope the abstype *point* is visible. Basic data algebra expressions in SOL have the form

$$\textbf{representation}\ \tau M_1 \cdots M_n,$$

where τ is a type expression (representing the carrier of the algebra) and $M_1 \cdots M_n$ are function definitions (representing the operations of the algebra). The interface specification can be viewed as an expression of the fact that there is a type named *point* with operations *create*, *plus*, *x_val*, and *y_val* (and the types of these functions depend on *point*). This informal reading of the interface specification corresponds exactly to the type of the data algebra. The types of data algebras are therefore expressed using the form

$$\exists \tau.\sigma_1(\tau) \text{ and } \sigma_2(\tau) \text{ and } \cdots \sigma_n(\tau),$$

where τ is the name of the ADT, and the $\sigma_i(\tau)$ are the types of the operations provided for dealing with τ.[2]

As indicated above, one feature of SOL is that a program may select from among several different ADT implementations at run time. A parser that uses a symbol table, for instance, may be parameterized by the representation of the type *sym-tab* and can then be passed different implementations, such as a hash table or a binary tree. Even though multiple ADT implementations may be manipulated directly by a program and passed around at run time, SOL programs are statically type checked at compile time.

1.2.2 Cardelli and Wegner's Existential Types

Cardelli and Wegner [1985] have designed a language similar to SOL, in which existential types are used to support ADTs.[3] Although many of the ideas are unchanged, the syntax is considerably more flexible, and records provide a powerful and useful structuring tool. This language was used as a tool for tutorial exposition by Cardelli and Wegner and was given the name FUN. As in SOL, the base language for FUN is lambda calculus, and the introduction of support for ADTs appears as a natural consequence of allowing existential type quantification.

---

[2] The notation $\sigma(\tau)$ is used to emphasize that τ may appear free in the type expression σ.

[3] FUN supports inheritance as well, which is discussed in Section 2.

In FUN, type specifications for variables of existentially quantified type have the form

$$\exists \tau.texp(\tau),$$

where *texp* is a type expression that may contain free occurrences of the type variable τ. If some variable p has an existential type of this form, then we interpret this as meaning that for some type τ, the type of p is $texp(\tau)$. Because FUN includes records, no special syntax is needed for representing signatures; $texp(\tau)$ can simply be the type of a record whose components will be the named operations made available by an ADT.

Within this typing system, one can work up to the idea of ADTs slowly by first examining simple uses of the ideas behind existential types. Thus, for instance, the type of the pair $\langle 3, 4 \rangle$ can be represented by either of the following expressions:

$$\exists \tau.\tau \times \tau \quad \text{or} \quad \exists \tau.\tau.$$

In the first case, τ = integer; in the second τ = integer × integer. This highlights the important fact that the same object can have different existential types. Because of this, and as an aid to type checking, FUN requires that special syntax be used to create objects whose types are to be existential and that the particular existential type desired by the user be explicitly indicated.

The syntax provided within FUN for creating objects of existential type is based on the idea of *packaging* so that internal structure is hidden. Thus, for instance, we might want to represent the pair ⟨3, add1⟩ as an object having the existential type

$$\exists \tau.\langle \text{val: } \tau, \text{ op: } \tau \rightarrow \text{integer} \rangle,$$

which uses a record type as the type expression within which τ appears free. To denote the desired object, we use the syntax

```
obj
= pack[τ = integer in⟨val: τ, op: τ → integer⟩]
  ⟨3, add1⟩.
```

This syntax achieves the above-mentioned goal of explicit indication of the desired existential type and suggests (because of the keyword *pack*) a hiding of information. Indeed, to use *obj*, its components may be referred to by name. The expression, obj.op(obj.val) thus evaluates to 4.

A special (optional) syntax is also provided for the use of existential objects. Staying with the packaging analogy, the keyword available for using a (packaged) object is *open*. To apply the operation packaged within *obj* to the packaged value, we could use the syntax

open obj *as* id[t] *in* id.op(id.val),

which evaluates to 4, as desired. Use of this syntax allows the introduction of a local name (in this example, *id*) for *obj* within the expression to be evaluated and another local name (in this example, *t*) for the otherwise hidden *representation* type.

Representation type is another name for what was called the carrier in connection with algebras; it is also sometimes called the *witness* type. The local name for this type (within the opening of a package) is treated as a new atomic type within the scope of its use, and the type of the result of the open expression (i.e., the type of the expression following the keyword *in*) is not allowed to depend on this type. This prevents the representation type from escaping its scope, so that it remains hidden within the package. It is this restriction that is at the heart of the fact that existential types are just ordinary types in FUN (and SOL) and that packages are ordinary values that can be manipulated in all the usual ways.

For all practical purposes, the above value *obj* is an ADT representation packaged with its set of operations. Its user knows (from *obj*'s type specification) that *obj.op* may be applied to *obj.val* and that the result will be an integer. Another expression with the same existential type as *obj* is ⟨list(1 2 3), length⟩. The fact that the similarity between these very different objects can be captured by existential types is interesting: Since they are of the same type, they could be put on the same typed list. The availability of existential types considerably increases the flexibility of a strongly typed language.

A representation of points similar to that displayed for SOL is now given. Figure 4

```
Definitions:
type point_ADT = ∃ p_rep . point_wrt[p_rep]

type point_wrt[p_rep] =
                    <create: function(real,real) result p_rep;
                     plus: function(p_rep,p_rep) result p_rep;
                     x_val: function(p_rep) result real;
                     y_val: function(p_rep) result real;>,

value cart_point:point_ADT =
          pack [p_rep = (real × real) in point_wrt[p_rep]]
                    <create = fun(x,y) cons(x,y);
                     plus = fun(p1,p2) cons(car(p1)+car(p2), cdr(p1)+cdr(p2));
                     x_val = fun (p) car(p);
                     y_val = fun (p) cdr(p);>
```

```
Legal Use:
value result = open cart_point as p in
          p.x_val(p.plus(p.create(1,4),p.create(2,3)));
```

```
Illegal Use:
value result = open cart_point as p1 in open cart_point as p2 in
          p1.x_val(p1.plus(p1.create(1,4),p2.create(2,3)));
```

Figure 4. FUN ADT for points.

contains an appropriate FUN definition of the types and an expression involving their use that, as in the SOL example, evaluates to 3. The keywords *value* and *type* are used to emphasize the nature of the entities being defined. Types do not exist at run time in FUN; values do.

As indicated, the type of the value *cart_point* (in Figure 4) is *point_ADT*, an existential type. The package prefix "cart" was chosen to indicate that a Cartesian representation is provided by the package. A polar representation is also possible, and the type system will prevent the manipulation functions in two different packages from interacting. In fact, as the example of illegal use indicates, the type system will even prevent interaction between manipulation functions obtained by separate openings of the *same* package. Because $p1$ and $p2$ are both *cart_points*, the representation types returned by $p1.create$ and $p2.create$ must be the same, but the typing system can neither recognize nor make use of this fact. $P1.plus$ can only be applied to points that originate from the $p1$ package.

The reason for this strange state of affairs is that in SOL and FUN the representation type incorporated within a package is *hypothetical*,[4] which means that there is essentially no access to the representation type (it is not even recognized to exist!) outside of its package. Even within an *open* statement, the most that can be done in FUN is to provide this type with a local name, which is treated as a new atomic type (even though the witness type may actually be a constructed type, such as a pair or a record). This means that the representation type has no meaningful permanent identity and can never be related to any other type except within the scope of an *open* expression (where it can only

[4] This terminology was introduced by Cardelli and MacQueen [1985]. They speak of three alternative ways of treating the representation type component of a package and characterize these with the terms *transparent*, *hypothetical*, and *abstract*.

match itself). This is an even stronger kind of information hiding than that normally implied by an ADT, where the representation type normally retains its identity even though its structure is hidden. This is a serious restriction because it makes it very difficult to provide implementation efficiency within a collection of interacting ADTs by making use of knowledge concerning representation types.

Since there is really nothing wrong with the second example of Figure 4, one's first impulse is simply to allow such behavior (by making the necessary changes to the type checker). Doing this is tantamount to electing to use what Cardelli and MacQueen call the *abstract* witness model. Their comments concerning this model are as follows:

> If we want to continue to view packages as values and existential types as ordinary types in this model, the distinction between types and values becomes blurred and we have to impose some rather ad hoc constraints to preserve static type checking. For instance, if A and B are of type $\exists\tau.\sigma(\tau)$, and we define
>
> $$C = \text{if } p \text{ then } A \text{ else } B$$
>
> then we will probably require that the witness type of C does not match either the witness of A or B. [Cardelli and MacQueen 1985, p. 234]

These comments seem reasonable if we want static typing because the witness types of A and B can be different, and p will in general depend on data available only at run time. Such an approach does not seem to be a serious restriction given the benefits. Within the abstract witness model, then, both examples in Figure 4 are acceptable. Of course, there will be other meaningful programs that are unacceptable within the abstract witness model owing to the desire to preserve static type checking in the presence of conditionals.

If dynamic type checking were to be used, propagating the abstract witness types of packages through conditionals at run time would provide sufficient information on which to base run-time checks, as long as type comparison is identity based rather than structure based (i.e., witness types must remain abstract when compared). This is because the use of representation or witness types in FUN is simply a way of getting a handle on the fact that two objects with the same existential type can have different implementations. The actual structure of the representation type may or may not be part of this difference—a fact that is highlighted by our examples, in which the representation types of the polar and Cartesian point packages are structurally identical.

1.2.2.1 *Adding in Universal Quantification.* In addition to supporting inheritance (discussed in Section 3), FUN also provides universal type quantification, which supports generic function types. When the two notions of existential and universal quantification are combined, it is possible to represent parametric data abstractions. These could be quite useful in the modular construction of software and can be thought of as providing functions over existential types (although no run-time computations are involved). It is interesting to note the similarity between inheritance and parametric data abstractions—both approaches provide a mechanism for incremental construction of software based on previously defined modules.

Suppose, for example, that we want to build on top of *point_ADT* by providing an ADT for lines. For every representation of points, there could be a corresponding representation of lines. Therefore, we might want to parameterize a line ADT with respect to the point ADT upon which it is based. Figure 5 shows how this can be done.

Because of the hypothetical witness of FUN ADTs, we must explicitly provide a point package within the line package to provide access to point operations that operate on the same representation as the line operations. *Generic_line* is therefore defined to be the type of a polymorphic function that (once specialized to a particular point representation) takes a point package representation (e.g., *cart_point*) and returns an object of type *line_ADT* with respect to that point representation. As shown in Figure 5, once the polymorphic function named *line* is defined, we can make a Cartesian line package, *cart_line*,

```
Definitions:
type generic_line = ∀ p_rep . function(point_wrt[p_rep]) result line_ADT;

type line_ADT = ∃ p_rep . line_ADT_wrt[p_rep];

type line_ADT_wrt[p_rep] = ∃ l_rep . line_ADT_wrt2[l_rep,p_rep];

type line_ADT_wrt2[l_rep,p_rep] =
                    <points: point_wrt[p_rep];
                     create: function(p_rep,p_rep) result l_rep;
                     length: function(l_rep) result real;>

value line[p_rep]:generic_line = fun(p:point_wrt[p_rep])
          pack[l_rep = (p_rep × p_rep) in line_ADT_wrt2[l_rep,p_rep]]
                    <points = p;
                     create = fun(p1,p2) cons(p1,p2);
                     length = fun(p1,p2) ((p.x_val(p1)-p.x_val(p2))^2 +
                                          (p.y_val(p1)-p.y_val(p2)^2)^1/2;>

value cart_line:line_ADT =
          open cart_point as p[p_rep] in
                    pack [rep = p_rep in line_ADT_wrt[rep]] line[p_rep](p)
```

```
Use:
result = open cart_line as l in
          l.length(l.create(l.points.create(0,0),l.points.create(0,1)));
```

Figure 5. Combined universal and existential quantification.

by first opening the Cartesian point package and then packaging up the desired line and point operations using the representation type and manipulation functions of *cart_point*. Note that *line*[*p_rep*] is actually applied to the point package *p* in order to produce the *cart_line* package; and especially note that in creating a line in the use example, the point package that was placed within the line package must be used to create the endpoints of the line.

As with the SOL example, the packages in Figures 4 and 5 do not hold a state for the abstractions they support but simply provide the operations for creating and manipulating them. A technique that Cardelli developed for Amber [Cardelli 1984a], however, allows maintaining a hidden state within an ADT as well. This technique, which makes use of recursive type and structure expressions, also provides a way of getting around the problem of interaction between packages of the same existential type, as Figure 6 shows.

In the example ADTs in Figure 6 (*cart_point* and *polar_point*), the tuple field labeled *value* holds the state of a point, either in Cartesian or polar representation, while the other tuple fields hold functions for manipulating this value. The overall approach in this example is still essentially functional; the plus operation returns a new ADT encapsulating a new state, rather than modifying an existing state via assignment. Given an assignment statement, side-effecting state modification would also be possible. This example helps highlight an important distinction between a packaged function returning an instance of the package's representation type (a violation of information hiding and, therefore, illegal) and returning an instance of the package's existential type (legal and done in this example by *plus*). Note how both Cartesian

```
Definitions:
type point_ADT = ∃ p_rep . point_wrt(p_rep);

type point_wrt(p_rep) =
                    <value: p_rep;
                     x_val: function() result real;
                     y_val: function() result real;
                     plus: function(point_ADT) result point_ADT;>

rec value cart_point(x,y):point_ADT =
          rec self . pack [p_rep = (real × real) in point_wrt(p_rep)]
                    <value = cons(x,y);
                     x_val() = car(self.value);
                     y_val() = cdr(self.value);
                     plus(p) = cart_point(
                               self.x_val + p.x_val,
                               self.y_val + p.y_val);>

rec value polar_point(x,y):point_ADT =
          rec self  pack [p_rep = (real × real) in point_wrt(p_rep)]
                    <value = cons((x^2+y^2)^1/2,...);
                     x_val() = car(self.value)*cos(cdr(self.value));
                     y_val() = car(self.value)*sin(cdr(self.value));
                     plus(p) = polar_point(
                               self.x_val + p.x_val,
                               self.y_val + p.y_val);>

-------------------------------------------------------------------

Use:
value result = (cart_point(1,4).plus(polar_point(2,4))).x_val()
```

Figure 6. ADTs with state.

and polar representations are allowed to interact. Smalltalk programmers should notice a very real and intriguing similarity between their style of programming and the style displayed in Figure 6. Assuming that appropriate Smalltalk classes for Cartesian and polar points have been defined, their use in Smalltalk code corresponding to the above example would look like the following:

result ← ((*cart_point* new: (1, 4))
plus: (*polar_point* new: (2, 4))) *x_val*,

which is exactly the form appearing in Figure 6, aside from the syntactic details of function invocation and argument passing.

1.2.3 MacQueen's Use of Dependent Types

The Introduction mentioned that the "formulas-as-types" notion has been a useful one for language theoreticians. SOL and FUN borrow their type formulas from second-order lambda calculus. Recently, there has been a great deal of interest in the formulas of constructive logic [Martin-Löf 1982], and MacQueen [1986] has shown how these can be applied to the description of ADTs.

Since the formulas of constructive logic are more expressive than the formulas of second-order lambda calculus, there is reason to hope that the resulting type theory will be more expressive. The main problem to be addressed concerns the drawbacks of

the formally based hypothetical witness model of existential types versus the informally based but more expressive abstract witness model. In particular, MacQueen takes issue with the requirement that ADTs based on Cartesian points (lines, for instance, as in Figure 5) must be defined within the scope of a single opening of the *cart_point* package:

> In general, we must anticipate all abstractions that use the point structure and might possibly interact in terms of points and define them within a single expression. It appears that when building a collection of interrelated abstractions, the lower the level of the abstraction, the wider the scope in which it must be opened. We thus have the traditional disadvantages of block structured languages where low-level facilities must be given the widest visibility. [MacQueen 1986, p. 279]

1.2.3.1 *Dependent Types.* To continue our discussion, we must introduce dependent types. The concepts behind these are not difficult to grasp, but our presentation will of necessity only scratch the surface. As an aid to further study of the ideas behind dependent types, we can recommend a number of references [Bates and Constable 1985; Constable and Zlatin 1984; MacQueen 1986; Turner 1984].

Type expressions in programming languages generally have a syntax that is borrowed (or adapted) from a syntax used in logic. From early experience with mathematics, this syntax has become almost second nature, so that the notation $f: A \rightarrow B$, meaning f is a function that maps elements of domain A to elements of the range B, seems natural to us. The type constructor in this case, $\rightarrow$, is used with the two type identifiers, A and B, to build a new, constructed type that we automatically identify as a "function" type. But we should be careful to separate the idea represented by $A \rightarrow B$ from the idea of a function; there are functions this representation does not adequately describe. An example will be given presently.

Another idea we pick up from our early experience is that of a Cartesian product between two domains. The usual notation is $z: A \times B$, meaning z is a pair whose components are, respectively, composed of elements from domain A and elements of B. In this case, the type constructor, $\times$, is used with two type identifiers to build a new, constructed type that we automatically identify as the type of a pair. But again, there are pairs that this type representation does not adequately describe (the type of the second element might depend on the value of the first).

It is possible to generalize our notions concerning functions and Cartesian products, but our syntax must also change. Constructive logic provides generalizations for our intuitions concerning functions and Cartesian products and also provides a syntax.

$\Pi x: A.B(x)$ is the notation used to represent a new idea of function: It is the type of a function that will map an element x from a domain A to a range $B(x)$; that is, the range depends on x (the element of A to which we apply the function) with B specifying the dependence.[5] B itself can be viewed as a type-valued function. Objects of this new function type, *general product*, are still created with lambda abstraction, for example. It is just that our new way of describing function types is more flexible; we can be more expressive and provide more information about the function. A good example is the division function. We can now express the fact that if we divide x/y (x, y reals), the range of the result will depend on y: If y is nonzero, the range is the set of real numbers; if y is zero, the range is the set containing the single value *undefined*. Of course, the old notation, $f: A \rightarrow B$ is still useful when the range of f does not depend on the value to which f is applied.

$\Sigma x: A.B(x)$ is the notation used to represent a new idea of Cartesian product; it is the type of a pair in which the value of the first component x (which is of type A) determines the type of the second component—again, with B specifying the dependence.[6] Objects of this new type, *general*

[5] Another syntax that has been used is $x: A \rightarrow B(x)$ [Constable and Zlatin 1984].

[6] An alternative syntax that has been used is $x: A \times\times B(x)$ [Burstall and Lampson 1984].

sum, are created by a pairing function (similar to cons, for instance) called *inject*, or *inj*, and are inspected by two projection functions, called *witness* and *out* (similar to *car* and *cdr*, for instance).

The function *witness* takes an object whose type is a general sum $\Sigma x{:}A.B(x)$ and returns x, the witness for the object (i.e., the first component of the pair). The type of *witness* is $(\Sigma x{:}A.B(x)) \rightarrow A$, using the syntax for expressing the type of a function whose range does not depend on the value of its argument. The function *out* takes a pair p, whose type is a general sum $\Sigma x{:}A.B(x)$, and returns the second component of the pair. The type of *out* is $\Pi p{:}(\Sigma x{:}A.B(x)).\ B(\text{witness}(p))$, a general product type, because the range of the result depends on the value of its argument.

One can use general products to represent generic or polymorphic functions, and general sums to represent existential types. The availability of the above projection functions on general sums, however, make the resulting objects (which MacQueen calls *structures*) "open," in contrast to the packages of FUN. Such an approach corresponds to the *transparent* witness model for existential types described by Cardelli and MacQueen [1985].

1.2.3.2 *Are Structures Values?* In FUN, existential types (the types of packages) are represented by starting with a type expression of the "usual" sort (i.e., composed from the available type constructors, primitive types, and variables representing types) and then abstracting with respect to the desired representation type variable. Thus, when a package is created by supplying a value for the representation type (bound by existential quantification) and values for the components of the object described by the overall type expression, the result is a value of the "usual" sort. Thus, packages are first-class citizens in SOL and FUN—they are simply data values. Unfortunately, this is not the case for MacQueen's structures.

The types of structures (structures being the dependent-type analog of packages) are not created with existential abstraction of a type variable in a type expression. Rather, in the typing theory investigated by MacQueen [1986], types are restricted to those that can be constructed with the Π and Σ type constructors.[7] We therefore need to explain how the types of structures are represented in this theory. This is done in two steps.

First, we look at the set of all the types that can be constructed, starting from primitive types (such as int and bool) and our type constructors Σ and Π. This set is closed with respect to Σ and Π constructions and is called the set of *small* types. Since we have agreed that we can think of a set as representing a type, we can call this set $\mathbf{Type}_1$. Now $\mathbf{Type}_1$ is, itself, not a small type—it cannot be constructed from Σ and Π and any of the small types. We therefore let $\mathbf{Type}_1$ be a member of (i.e., a value of type) $\mathbf{Type}_2$, which we define by starting with $\mathbf{Type}_1$ and again closing with respect to Σ and Π.

This process of defining types and the type of types (and the type of types of types ...) could go on forever. Luckily, though, we do not need to go any further; the existential types of SOL and FUN correspond to values in $\mathbf{Type}_2$. In particular,

$$\exists \tau.\sigma(\tau) \simeq \Sigma \tau : \mathbf{Type}_1.\sigma(\tau).$$

Although the type of the left expression above is of the same class as, for example, *integer* in FUN or $\mathbf{Type}_1$ in the DL system (carrying on the analogy), the type of the expression on the right is $\mathbf{Type}_2$. Since the types of structures in this new system are of $\mathbf{Type}_2$, the values of structures are found in $\mathbf{Type}_1$. In this typing system structures are therefore types (i.e., at the level of "integer" as opposed to the level of "1"). MacQueen [1986] suggests a special syntax for building structures, which may be thought of as a module definition and interconnection language. Within this language, called DL, a *signature* is used to indicate the type of a structure, and functions that produce structures (such as line, the generic function in Figure 5) are called *functors*.

[7] Other constructors useful for representing the types of records, for instance, are made available, but $\exists$ is not available.

```
Definitions:
signature point_ADT = Σ p_rep:Type1 . <
                    create: function(real,real) result p_rep;
                    plus: function(p_rep,p_rep) result p_rep;
                    x_val: function(p_rep) result real;
                    y_val: function(p_rep) result real;>

structure cart_point:point_ADT = inj(real×real, <
                    create = fun(x,y) cons(x,y);
                    plus = fun(p1,p2) cons(car(p1)+car(p2),cdr(p1)+cdr(p2));
                    x_val = fun(p) car(p);
                    y_val = fun(p) cdr(p);>)

structure polar_point:point_ADT = inj(real×real, <
                    create = fun(x,y) cons( ..., ...)
                      ...>)

signature line_ADT_wrt(p:point_ADT) = Σ l_rep:Type1 . <
                    create: function(|p|,|p|) result l_rep;
                    length: function(l_rep) result real;>

structure cart_line:line_ADT_wrt(cart_point) = inj(|cart_point|×|cart_point|, <
                    create = fun(p1,p2) cons(p1,p2);
                    length = fun(l) let p1 = car(l), p2=cdr(l) in
                             ((p1.x_val-p2.x_val)^2+(p1.y_val-p1.y_val)^2)^1/2;>)
```

```
Use:
value result = open cart_point as p1 in
               open cart_point as p2 in
                 open cart_line as l in
                   l.length(l.create(p1.create(0,0),p2.create(0,1)));
```

Figure 7. DL ADTs for point and line.

In DL, if we agree that we will only compute with values (i.e., not types) during run time, as MacQueen suggests, then we can no longer select structures at run time through the use of conditionals or create structures dynamically on an as-needed basis, as is possible in SOL and FUN. Nor can we use the technique illustrated in Figure 6 for Smalltalk-like state-based interaction between ADTs of the same existential type; structures cannot be passed or returned at run time. Instead, structures must be statically defined and connected before execution. It is interesting that this is essentially the same restriction seen in OBJ2, a first-order language [Futatsugi et al. 1985].

What may not be immediately clear is what has been gained. Let us therefore build up point and line abstract data types in DL, as done in Figures 4 and 5 for FUN, and compare the way in which this is accomplished. We should see a difference in the way the line ADT is able to access the point ADT and its operations. Figure 7 shows this. We use the notation | structure | to represent the expression "*witness*(structure)."

The benefit of the above approach is to allow points created through any use of *cart_point* to be used in construction of a *cart_line*. As can be seen in this example, it is no longer necessary to explicitly open a package (e.g., *cart_point*) in order to obtain

a scope within which a higher level abstraction (e.g., *cart_line*) may be defined. This ease of expression of structures based on other structures and their respective types is an improvement over the approach required by existential types under the hypothetical witness model. Nevertheless, the *line_ADT* above requires a point structure as its argument, preventing lines from being constructed from points with different representation structures. This restriction was also seen in FUN (Figure 5) owing to the need to explicitly provide a point package required for creation of end points within the line package. As seen in the next section, this restriction is not found in Russell.

MacQueen [1986] provides numerous examples of the increased flexibility of DL and shows how various approaches to defining structures are possible within this language. He believes that the module system developed for ML [MacQueen 1985] has this dependent-type model as it most natural generalization (as opposed to that provided by existential types).

The DL model of types is a *stratified* model, in which there are different levels of types, each level being constructed from those below it, but there are also *unstratified* models, in which there is a "type of all types." Cardelli [1986] has recently developed a type system, in which framework the dependent types we have discussed in this section may be placed. This system has the promise of providing packages as values, with transparent witness types. It appears that static type checking within this system is in general undecidable, but whether this is actually a problem for realistic programs remains a question for further research.

1.2.4 Russell and Poly

The FUN and DL typing systems have as their overall aim an understanding of programming in terms of the type domains from which denotable values are drawn. One result of this has been expressing the type domains from which ADTs (i.e., objects whose behavior corresponds to our intuitions concerning ADTs) are drawn. Rules for type checking ADTs arise naturally from the coupling of abstract syntax to semantics that such a denotational approach provides.

The heart of this approach seems to be the view that the types from which values are drawn are somehow fundamental and that values must be explained in terms of these types. Type checking is then the act of verifying that denoted values are indeed members of the type from which they are claimed to be drawn and that these values are used in ways consistent with the properties common to values of this type.

But one can also take an opposing view—that values are the fundamental reality and that a type is just a set of operations (i.e., procedures and functions) that provide a consistent interpretation of values. Under this regime, type checking is the act of verifying that values will not be *misinterpreted* by operations in which they are used. Values in such a language do not have types, themselves, but are interpreted by the operations of types. We hope the connection with ADTs here is clear: In this view, all types (even such primitive types as integer and Boolean, for instance) support data abstraction in much the same way as data algebras, packages, and structures do—by providing the necessary operations for manipulating the representation of data for which they were designed.

This approach should make sense to programmers, who are generally aware that the same value stored in a memory location can be thought of as an integer, a boolean, a machine instruction, a pointer, a floating-point number, or any one of a number of different possibilities, depending on the operations in which it takes part. In the course of an integer add operation, for instance, it is not the data values that hold or define "integerness"—it is the add operation itself (or, indeed, the entire collection of integer operations, all of which agree on a consistent interpretation of a universal untyped value space). The first programming language to make uniform use of this point of view toward types was Russell [Demers and Donahue 1979]. Although this may seem a major change of view, Hook has defined a language based on dependent types called Kernel Russell into which

Russell programs may be naturally translated [Hook 1984]. Thus, it is possible to relate these two viewpoints.

Russell supports the principle of declaration correspondence [Landin 1966]—anything that can be declared can be passed as a parameter, and vice versa. In Russell, therefore, one can parameterize any construction within the language with respect to any free identifier appearing in it. Since types are constructible values in Russell (they are, after all, just sets of operations), this allows parameterization with respect to types. Thus, parametric polymorphism (of the kind used in the *generic_line* example in Figure 5) is available from first principles.

Another principle of Russell is that all type checking be done statically, at compile time. As might be expected, static type checking in a language in which types may be created and passed as values is a problematic objective. Although it is difficult to take issue with any of Russell's principles or with its view of types (which is a useful approach, as is shown presently), the particular syntactic restrictions placed upon the language to guarantee static type checking may be questioned.

In order to support static type checking in Russell, no function or type expression may use a free identifier bound to a variable in the surrounding scope.[8] This requirement is a result of two facts: First, functions can return types as their results; second, types match in Russell if and only if they are syntactically equal or can be converted through renaming, reordering, or forgetting so that they are syntactically equal. As a result of this restriction, called the *import rule*, syntactically equal type expressions in Russell always denote the same type.

Another language based on the same principles as Russell, but with different restrictions to guarantee static typing, is called Poly [Matthews 1983]. Poly does type matching by name, which is a simpler approach than that taken by Russell. Matthews, the designer of Poly, has suggested that Russell might recognize two different kinds of functions: those that do not use free identifiers bound to variables in a containing scope and those that do. The first class could be safely allowed in type expressions, whereas the second could not. This would be less restrictive than the current import rule, which prohibits all functions from importing variables. Matthews believes that this would be the most desirable approach for future languages similar to Russell or Poly.

An example of Russell code is now given in Figure 8. In it, the capabilities demonstrated in Figure 6 (multiple ADT representations) and in Figure 7 (transparent witness) are combined. As mentioned earlier, it is not clear how to do this in either FUN or DL.[9]

In Russell, the keyword *with* corresponds to the keyword *rec* in the FUN example of Figure 6—it introduces a local name for the structure being defined (in this case, a Cartesian product structure defined using the Russell *prod* constructor) that can be referred to the in the definitions for the operations on this structure. The Mk operation referred to in the operation definitions is provided by *prod* and is analogous to an *n*-ary cons.

In Russell, the problem with the hypothetical witnesses of existential types is not present; the problem with dependent types, that structures are not first-class values, is also not present. Of all the example typing systems reviewed so far, the Russell system seems the most satisfactory in terms of its expressive power and flexibility. As can be seen by the above example (in which a new line is created without explicitly mentioning the types of the point arguments), a certain amount of type inferencing is performed by the Russell compiler. This has the benefit of making the language less verbose and easier to use.

Unfortunately, there is currently no formal base for type inferencing in Russell, so

[8] The term *variable* has a special meaning in Russell, indicating that a side-effecting assignment may be used to change its contents. Variables are distinguished from constants, which are called *values*.

[9] We are indebted to Professor Hans Boehm of Rice University for his assistance in developing this example. A Russell compiler developed by Professor Boehm, A. Demers, P. Matthews, and J. Hook is available to the general public.

Definitions:

```
#define Point_type type P {
    new: func [x,y:val Float] val P;
    xval,yval: func[val P] val Float;
    + : func[p1,p2: val P] val P }

cart_point === prod { x, y: val Float } with cp {
  new == func [x,y: val Float] {cp$Mk};
  xval == func [val P] { cp$x };
  yval == func [val P] { cp$y };
  + === func [p1,p2:val cp] {new[xval[p1]+xval[p2], yval[p1]+yval[p2]]}
  } export {new; xval; yval; +};

polar_point == prod { r, theta: val Float } with pp {
  new == func [x, y: val Float] { pp$Mk(sqrt(x*x + y*y), arctan(y/x)) };
  xval == func [val P] { pp$r * cos(pp$theta) };
  yval == func [val P] { pp$r * sin(pp$theta) };
  + === func [p1,p2:val pp] {new[xval[p1]+xval[p2], yval[p1]+yval[p2]]}
  } export {new; xval; yval; +};

line == prod {p1: val t1; p2: val t2; t1,t2: Point_type} with l {
  new == l$Mk;
  length == func [line: val l] {
    let
      dx == xval[p1[line]] - xval[p2[line]];
      dy == yval[p1[line]] - yval[p2[line]]
    in sqrt[dx*dx + dy*dy] ni
    }
  } export {new; length}
```

Use:

```
let
  p1 == cart_point$new[1.0,2.0];
  p2 == polar_point$new[2.0,3.0];
  l == line$new[p1,p2]
in
    line$length[l];
ni
```

Figure 8. Point and line ADTs in Russell.

the programmer often has to guess whether an explicit signature will be required. In many cases, types need to be given explicitly even though it seems reasonable for the compiler to be able to infer them. In the *cart_point* example in Figure 8, for instance, it seems strange to have to state explicitly *xval == func[val P]{cp$x}*, rather than simply state *xval == func[val P]{x}* when *x* clearly belongs to the *cart_point* structure.

The kind of type inferencing done by ML language compilers [Cardelli 1987; Milner 1978], for which no types at all need be explicitly mentioned by the programmer, is not possible in Russell as it now stands. Nor, for that matter, does it seem likely for FUN. The ML typing system is considerably less expressive than that of Russell or FUN (and in any case it does not deal with modules); this seems to be a necessary trade-off. The question as to just how much explicit typing is necessary in these languages remains for future research. There is current interest in addressing these foundational issues for Russell, and we hope

that in addition to elucidating a formal base for type inference in Russell, future work will result in an extension of Russell types to handle inheritance, the other necessary component of OOP.

2. INHERITANCE

We now turn to inheritance, the other half of the OOP ≃ ADTs + Inheritance equation. To inherit is to receive properties or characteristics of another, normally as a result of some special relationship between the giver and the receiver. The resulting properties of the receiver in such an act are not necessarily limited to those that are inherited, nor, in general, are *all* of the properties and characteristics of the giver necessarily inherited. This is certainly true for the usual kind of inheritance that takes place in human society, wherein the receivers and givers are people and the relationships are typically based on marriage or birth.

In view of the above reasoning, it might be thought that there is no a priori reason for restricting or imposing a discipline on the use of inheritance in defining objects. If one wants to build a new object that includes a capability already available in some other object, why not simply use inheritance to obtain the desired capability, while ignoring those that are not desired? Certainly it is possible to use inheritance in such an ad hoc fashion; in fact, most OOP languages expressly provide the means of doing so. However, as indicated above, inheritance is normally closely tied to some relationship between the giver and receiver. Ideally, such a relation (e.g., **is-a** relationship) should support formal reasoning about the behavior of objects defined using inheritance and at least be helpful to a programmer's informal reasoning. In the case of the **is-a** relationship, the consistency of reasoning about objects based on inheritance of properties may require all properties to be inherited [Brachman 1985]. Thus, there is a need to be disciplined in the use of inheritance in defining objects so that the implicit relationships between objects that arise from the use of inheritance are not violated or contradicted by the behaviors of these objects. In attempting to provide a formal model for inheritance, this tension between what is reasonable on the basis of an anthropomorphic analogy and what is reasonable in terms of formal logic becomes an important factor.

In Smalltalk, there are no type-checking means or other tools to help a designer ensure that a proposed addition to the class hierarchy makes sense in terms of these implicit relationships. In fact, there are numerous examples in the Smalltalk-80 system in which the expected relationship between objects defined using inheritance is violated [LaLonde et al. 1986]. In Smalltalk, it is quite easy to give a definition of a subclass that includes a method that overrides one of the same name higher up in the class hierarchy, giving the objects of the subclass an inconsistent behavior with respect to objects of an ancestor class. Although overriding is usually used in a principled fashion to provide an extended and *consistent* behavior to objects of a subclass, there is currently no well-defined notion of such an extension that formally distinguishes it from a situation in which the use of overriding produces inconsistent behavior.

A primary motivation for the use of inheritance in programming is that it provides both a specificational structuring mechanism and a means of reusing specifications that is based on common sense notions that are natural to our way of thinking. When using inheritance in the definition of an object, a designer need only specify what is new about the object in comparison with the objects from which it inherits properties. In this sense, we might view inheritance as an ADT constructor. In Smalltalk, a subclass inherits and may extend both the state representation (instance variables) and the operations (methods) of the superclass. Thus, a programmer need only specify any additional instance variables or methods that characterize the new class of object being defined. On the other hand, CommonObjects [Snyder 1987] expressly forbids inheritance of instance variables on the grounds of preserving strong encapsulation of data representa-

tions. This approach attempts to preserve as much as possible the connection between ADTs and objects, while still supporting a useful form of inheritance.

The following issues therefore seem important in trying to model OOP inheritance: First, what is the formal relationship between objects that inheritance is intended to reflect and to what extent should violations of this relationship be controlled or allowed? Second, to what extent should inheritance be allowed to violate principles of data abstraction? Additionally, the authors believe that the first of these issues begs an important question—namely, whether there should be but a single inheritance hierarchy and corresponding relationship. We believe that there should be separate inheritance hierarchies and module relationships corresponding to the reuse of behavioral specification and implementation specification, respectively. Taking such an approach should alleviate the problem of creating inconsistent behaviors when overriding methods and, in many cases, remove the need to perform overriding at all.

The following sections focus on these issues while reviewing recent work aimed at addressing inheritance within a formal framework. Following this, we review a number of OOP languages, considering how they address these issues and how they relate to the formal frameworks that have been suggested.

2.1 Ordering Relations on Types

Key to a formal basis for type inheritance is the definition of an order relation on the set of type expressions. Ordering the set of types expressible within a type system has been used in several different aspects of programming language design

(1) to organize application of coercions in compilers [Hext 1967; Mitchell 1984a],
(2) to support resolution of overloading [Jones and Muchnick 1976; Kaplan and Ullman 1980],
(3) to support type checking and inferencing [Mitchell 1984b], and
(4) to support inheritance [Ait-Kaci and Nasr 1986; Cardelli 1984b; Cardelli and Wegner 1985; Dahl and Nygaard 1966; Futatusugi et al. 1985; Goguen and Meseguer 1987; Ingalls 1978].

Although it is not the purpose of this section to discuss topics (1)–(3) in detail, we mention them since they are often closely intertwined with inheritance. In Smalltalk, for instance, *overloading* and *polymorphism* are integrated via the class hierarchy.

2.1.1 Overloading

Overloading refers to the use of a single syntactic identifier to refer to several different operations (methods in Smalltalk), discriminated by the types of the arguments to the operations. Overloading, along with coercion, is considered a form of ad hoc polymorphism [Cardelli 1986], thus suggesting an unprincipled and incoherent syntactic mechanism. An example of overloading is the use of "+" for the addition operation over integers, rationals, and reals (not to mention vectors and matrices). It is hard to see what is unprincipled or incoherent here, since the operation of addition is common to all these cases; but in the absence of a coherent framework, we could also imagine "+" being used to indicate the set union operator. The use of a well-founded class hierarchy based on inheritance can provide a framework for the principled use of overloading. In the above example, integers, rationals, and reals are considered successive specializations of one another. Thus, the various operations associated with "+" are defined over successively restricted domains.

2.1.2 Polymorphism

A polymorphic operation is one that can be applied to different types of arguments. We use the term *functional polymorphism* to refer to the situation in which a single, specific function can be applied uniformly and independently of (some aspects of) the types of its arguments. An example is the **length** operation, which takes a list composed of elements of any type and returns the number of elements in the list. This operation can be naturally defined so that it is independent of the type of elements contained in its argument list. Another ex-

ample of such an operation is the **identity** operation, which maps an argument of any type to itself. Many researchers, including Cardelli and Wegner [1985], call this notion *parametric* polymorphism. This is because the types of the arguments for such functions appear as implicit (in ML) or even explicit (in FUN) type parameters in the polymorphic function definition. The *generic_line* example of Figure 5 shows this. We prefer not to call this parametric polymorphism for the following reasons.

There are actually two different approaches to polymorphism. The style introduced by Milner [1978] and popularized by the ML language is the one we refer to as functional polymorphism. It is characterized by the uniform behavior of a single function over a range of types. Type quantification to define this range may be either implicit or explicit (as required for type checking in FUN), but types are *not* supplied as explicit parameters to functions, and functions do *not* make reference to or use the types of the arguments in any operational fashion. The other style of polymorphism has its roots in Reynolds [1974], involves explicit quantification over types, and is obtained by supplying types as explicit parameters to functions. The Russell language supports polymorphism in this way. Calling both approaches "parametric" seems to be asking for confusion, since they seem different, both operationally and semantically. We therefore recognize this distinction (when appropriate) by using either *functional* or *parametric* as a qualifier. When the distinction is not important, we omit the qualifier.

A natural refinement of polymorphism is *inclusion*, or *bounded* polymorphism, in which "an operation may be applied to objects of different types related by inclusion" [Cardelli and Wegner 1985]. Inclusion polymorphism provides a means of characterizing restricted applicability of a polymorphic function. The form of inclusion polymorphism normally seen in current OOP languages is functional polymorphism that relies on a common representation among different types related by inclusion.

In Amber [Cardelli 1984a] and FUN [Cardelli and Wegner 1985], for instance, inclusion is defined over record types, and bounded polymorphic operations operate on such records by using fields that are common to the record types involved. Examples of this approach are given in Section 2.3. In Smalltalk, inheritance of methods from a superclass may be viewed as inclusion polymorphism. In this case, commonality of representation arises from the fact that the instance variables upon which the inherited operation depends are inherited also. The set of subclasses of the class defining the method represents the set of legal classes that the bounded polymorphic method applies to. Thus, a method defined within the class **Object** can be viewed as a universally polymorphic operation (i.e., it may be applied to all objects).

We now discuss in more detail the approaches to inheritance and ordering of types that we see in the work of Goguen, Cardelli, and Ait-Kaci. These approaches are based on order-sorted algebras, record orderings, and term orderings, respectively.

2.2 Order-Sorted Algebras

This section discusses the work of Goguen and others [Futatsugi et al. 1985; Goguen and Meseguer 1986, 1987]. Each of the references just given includes extensive citations and discusses a system intended to highlight a particular cluster of features: OBJ2 [Futatsugi et al. 1985] is a functional system based on equations; EQLog [Goguen and Meseguer 1986] is a full logic programming system based on the Horn logic with equality; and FOOP [Goguen and Meseguer 1987] is based on an approach to supporting objects with local state (as in Smalltalk) in a functional setting. In the following discussion, we use OBJ to refer in a generic fashion to all of this work—not just the system described in Futatsugi et al. [1985].

OBJ uses two interrelated methods of subtyping: subsorts and relations on modules. An OBJ module provides a packaging of one or more abstract data type definitions. The definition of an abstract data type will involve a set of equations, possibly with conditions, that mention one or more sorts. In Section 1.1 on algebraic approaches, the notion of a data algebra was

```
module LIST[Object::Trivial] is
  using NAT .
  sorts NeList List .
  subsorts NeList < List .
  ops
    nil : -> List ,
    cons : Object List -> List [associative with identity = nil] ,
    hd : NeList -> Elt ,
    tl : NeList -> List ,
    size : List -> Nat .
  vars
    X : Object ,
    L, L' : List .
  axioms
    hd(cons(X,L)) = X .
    tl(cons(X,L)) = L .
    size(nil) = 0 .
    size(cons(X,L)) = 1 + size(L).
endmod LIST
```

Figure 9. AN OBJ2 module.

introduced. In the work of Goguen, there is a standard method of obtaining an algebra from the definition of an ADT. In particular, one can think of a *sort* (e.g., Boolean) in an ADT definition as an uninterpreted identifier that has a corresponding carrier in the standard (initial) algebra.

Figure 9 shows an example of an OBJ2 module. It declares a signature and set of operations for lists of objects. The module parameter makes explicit the assumptions made for objects—namely, they satisfy the theory named *Trivial*. This theory is declared within another module (not shown) and has no axioms. Because *Trivial* has no axioms, any object satisfies this theory, and any object may be put on a list. The **subsorts** declaration defines an ordering to hold on the sorts *List* and *NeList*. In particular, the operators **cons** and **size** will be inherited by the sort of *NeList* of nonempty lists. Initial algebras will correspond to each of the two sorts introduced by this module. The set corresponding to the sort *NeList* will be a subset of the one for *List*, and the algebra for *NeList* will be a subalgebra of the one for *List*. The **using** clause indicates that the definition of this module depends on some of the properties defined in the module *NAT*. In this case the sort named *Nat* is the codomain of the operator **size**.

In OBJ, another form of inheritance is achieved via a hierarchy of modules. The schematic module definition shown in Figure 10 illustrates this. It defines a subclass of *LIST*s that can be sorted by virtue of a partial order being defined over the elements of the *LIST*s. This is indicated by the module parameter type being restricted to *POSET*. *POSET* is the theory of partially ordered sorts. Just as in Smalltalk, a subclass may add new message selectors and methods that do not occur in the superclass; here the module defines two new operators over the sort *List* of the module *LIST*. Note also that because of the subsort relation defined in *LIST*, the sort *NeList* inherits the two new operators also.

All of the mechanisms involved in subsorts and the module hierarchy can be given a semantics in terms of order-sorted algebras, which in turn may be mapped into many-sorted algebras, which are well understood. In this sense, all of the OBJ2 inheritance mechanisms can be considered a form of syntactic sugar that does not increase the expressiveness of the resulting language. In practice, however, the notational and conceptual economies gained are considerable.

2.3 Set-Inclusion Orderings

The foundation for Cardelli's work with inheritance [Cardelli 1984b; Cardelli and Wegner 1985] is provided by the "types-as-

```
module SORTABLE-LISTS[X::POSET] is
  using BOOL .
  extending LIST[X]
  ops
        sorted : List -> Bool ,
        sort : List -> List .
  .
  .
  .
end SORTABLE-LISTS
```

Figure 10. Module hierarchies in OBJ.

ideals" approach, developed by MacQueen, Sethi, and Plotkin [MacQueen and Sethi 1982; MacQueen et al. 1984]. It seems serendipitous that a theory originally developed as a semantic foundation for type checking functionally polymorphic languages should also provide a foundation for understanding inheritance. As we show here, however, the FUN typing system described in Section 1.2.2 can be extended in a natural way to support inclusion polymorphism and, thereby, a form of inheritance.

2.3.1 Types as Ideals

We have explained the reasoning behind viewing types as sets. Even though all types may be viewed as sets of values, viewing all possible sets of values as types is not so reasonable. Since values of the same type normally have a common structure (such as being pairs, functions, or integers) associated with them, a first step in developing a semantics for a typed language is to be precise about the nature of the sets that make reasonable types.

In the theory of types as ideals, a set is a type if (and only if) the set is an ideal. Ideals are nonempty sets that satisfy two properties: They are (1) downward closed and (2) consistently closed under a complete partial ordering (cpo) on the domain of possible values, V. The set V and the cpo are the usual mathematical constructs used to present the denotational semantics of a lambda-calculus-based language. V is the set of all denotable values, and the cpo is an ordering based on information content.[10]

The two properties given above that define ideals reflect intuitive ideas concerning types and the way in which the elements of V are built up from approximations involving primitives such as Bool and $\perp$, the least element of V, called *Bottom*. Requiring a type (set) to be downward closed means that the approximations of individual elements of the type are also in the type. Requiring a type (set) to be consistently closed means, roughly, that the least upper bound of an approximation (subset) of the type is itself in the type.

There are two important results that derive from restricting types to ideals in this way. First, the set of all types (i.e., the set of all the ideals of V) becomes a complete lattice under the set inclusion ordering $\subseteq$ [MacQueen and Sethi 1982]. This is an important result because it makes it easy to reason about subtypes. Second, given some minor restrictions, recursive type equations have solutions [MacQueen et al. 1984]. This is of crucial importance because many useful functions have recursive types. For instance, in order to express the type of x in the expression $x(x)$, the type equation $s = s \rightarrow t$ must have a solution (this solution is the type of x). The example in Figure 6 used recursive types.

For future reference, we identify the set of all types (the set of all ideals of V) as *TYPE*. As we have mentioned, *TYPE* is a complete lattice when ordered according to $\subseteq$. At the top of this lattice is the type called *TOP*, whose elements are exactly those of V.

2.3.2 Functional Polymorphism in the Ideal Model

Functional polymorphism has been characterized as the ability of a function to operate uniformly and independently of (some aspects of) the types of its arguments. The identity function Id is a classic

[10] Two excellent sources of information concerning the denotational approach to language semantics may be found in the work of Dana Scott [1976] and Joseph Stoy [1977]. In these semantics, the domain V is required to satisfy a recursive domain equation, such as $V = \text{BOOL} + \text{INT} + [V \times V] + [V \rightarrow V]$.

example of such a function, as is the length function Ln, which returns the number of elements in a list. The fact that Ln cannot be applied to all types of arguments (only lists) is the reason for use of the phrase "some aspects of" in the characterization. Universal and bounded universal quantification let us express exactly what aspects these are for polymorphic functions.

Universal quantification of type variables was introduced in Section 1.2.2.1 and used in Figure 5. We now explain why the ideal model allows us to represent functional polymorphism using universal quantification and inclusion polymorphism (a form of inheritance) as bounded universal quantification. We begin by using the identity function as an example.

Id is a member of the type $TOP \rightarrow TOP$, but this fails to capture the essence of Id. A function of this type could map integers to booleans, for instance. To be more precise, note that Id maps integers to integers, booleans to booleans, and so on, for any type t. Therefore, it is quite natural to use universal quantification and say that Id has the type expressed by the formula $\forall t.t \rightarrow t$. Although this seems reasonable, what does it mean? If types are ideals, such a formula must have an interpretation in terms of ideals.

In the ideal model, according to MacQueen et al. [1984], universal quantification corresponds to taking intersections of ideals containing certain total functions. Let $D \rightarrowtail E$ be the set of total functions in V that map elements of the ideal D to elements of the ideal E. Stated more precisely,

$$D \rightarrowtail E \equiv \{f \in TOP \rightarrow TOP \mid x \in D \Rightarrow f(x) \in E\}.$$

For any D and E that are ideals, this set is an ideal and is therefore a valid type in the ideal model.

There is an ideal (a type) in V that contains all total functions that map Booleans to Booleans, represented as Bool $\rightarrowtail$ Bool. We do not care what these functions do to other values, but they always map Booleans to Booleans. Id is in this ideal, so we write Id $\in$ Bool $\rightarrowtail$ Bool. But we can also write Id $\in$ Int $\rightarrowtail$ Int. In fact, for any type T, Id $\in T \rightarrowtail T$. Because Id is in all of these ideals, it is in their intersection, so we can write Id $\in \bigcap_{T \in TYPE} T \rightarrowtail T$. This, then, is the meaning of the expression Id: $\forall t.t \rightarrow t$. A similar interpretation provides the meaning of Ln: $\forall t.LIST(t) \rightarrow INT$, namely Ln $\in \bigcap_{T \in TYPE} LIST(T) \rightarrowtail INT$.

Because of the lattice structure of $TYPE$, an equivalent representation of the type of Id is Id $\in \bigcap_{T \leq TOP} T \rightarrowtail T$. The ordering $\leq$ is that generated by $\subseteq$ on $TYPE$. Generalizing over syntax, then, we write Id: $\forall t \leq TOP.t \rightarrow t$. It is this more general form that suggests bounded quantification in which the type TOP in this expression is replaced by some other type in the $TYPE$ lattice. Thus, the general form for representing bounded universal quantification becomes $\forall t \leq T.texp(t)$, where T is some type in $TYPE$, and $texp(t)$ is a type expression that can depend on the type variable t.

To summarize, the two important results of this approach with respect to inheritance are

(1) $TYPE$ (the set of all types) is a complete lattice ordered by set inclusion;
(2) both universal and bounded universal quantification have natural interpretations in terms of the ordering defined by this lattice.

2.3.3 Examples of Inheritance

Inclusion polymorphism captures a form of inheritance used in OOP. The use of inclusion polymorphism for this purpose, however, requires that the hierarchy normally expressed in terms of superclass/subclass relationships be reflected in type/subtype relationships. For this purpose, Cardelli [1984a] has introduced *records* and their types. Records are then used to model class instances, and functional record components are used to model methods [Cardelli and Wegner 1985].

A record is a finite association of values to labels, for example: ⟨age = 5; speed = 20⟩. The type of this record is indicated by

the expression, ⟨age, speed:int⟩. The ordering of labels is not important (i.e., for all practical purposes, ⟨age = 5; speed = 20⟩ ≡ ⟨speed = 20; age = 5⟩) because the only way fields can be accessed is via their label. The examples in Section 1.2.2 used records to store ADT functions and values, so they could be referenced by name (as opposed to position within the record).

The aspect of records that is new to this discussion concerns the way in which their types may be related. By definition [Cardelli 1984b], a record type R' is a subtype of type R (written $R' \leq R$) if (and only if) R' has all the fields of R (possibly more) and the common fields of R' and R are in the subtype relation. This definition of subtyping for records is motivated by (1) the desire to represent objects using records and (2) the desire that inclusion polymorphism provide results on records that correspond with our intuitions concerning object descriptions. The essential intuition embodied in this ordering is that if, for instance, every car is a thing (i.e., the type containing all cars is itself contained in the type containing all things), then when one describes a car, one describes a thing. When descriptions are given via the fields of record types, this ordering does, in fact, model *some* of our intuitions concerning OOP—namely, the relationship between instance variables of classes in a subtype/supertype relationship. Unfortunately, as we show, this ordering seems the inverse of the correct ordering for methods.

Figure 11 shows some example record types and the subtype relationships that arise from this definition. The first two type declarations can be read as "all things have an age" and "all vehicles are things that have a speed." Note that in this example, *vehicle_type* and *machine_type* are not related by ≤ (because of the way in which the types were defined). This illustrates the fact that in this typing system, subtypes arise solely from *implementation* decisions concerning the representation of objects, not from observed or intended behavioral relationships between objects in an application. In FUN, one never declares one type to be a subtype of another. Thus,

Types:

thing_type = <age: int>
vehicle_type = <age: int; speed: int>
machine_type = <age: int; fuel: int>
car_type = <age: int; fuel: int; speed: int>

Type Relationships:

car_type ≤ vehicle_type ≤ thing_type
car_type ≤ machine_type ≤ thing_type

Values:

thing = <age = 5>
vehicle = <speed = 10; age = 5>
car = <fuel = 20; age = 10; speed = 80>

Figure 11. Some example record types.

the types of objects that are not related in an application domain may be related in a corresponding FUN program, simply as a result of coincidental representations. Similarly, the types of objects in an application domain that *are* perceived to be related will not be so related in a FUN program unless their representations are appropriately designed. This focus on implementation representations for defining subtype relationships is understandable; static type checking (and the absence of run-time type errors) is a primary objective of the FUN type system. The above observation does, however, underscore a need to use separate behavioral and implementation hierarchies. These could be used as a database for a system development environment, whose purpose would be the controlled generation of FUN programs.

Let us now ask what benefit can be derived from the valid conclusion that *car_type* ≤ *thing_type*. We first define a function called *mk_thing_older* that will make a thing older. To help us express this function, we introduce a syntax for dealing with records. The expression *record* **with** ((*label*1, *value*1), (*label*2, *value*2), . . .) will be taken to denote a new record value whose fields are identical to those of *record*,

except for those fields whose labels are *label*1, *label*2, and so on, which fields contain, respectively, *value*1, *value*2, and so on. Figure 12 shows two possible *mk_older* functions and their use on the values defined in Figure 11. Note that the types of the parameters for these functions are indicated using "≤" instead of ":", a natural extension of parameter type declaration in polymorphic function definitions.

As seen in Figure 12, the typing system will allow *mk_thing_older*(*car*), because *car_type* ≤ *thing_type*, and *mk_thing_older* is a polymorphic function that operates on any type ≤ *thing_type*. This shows the exact sense in which inclusion polymorphism models inheritance. In Smalltalk, for instance, the function *mk_thing_older* could be placed in the *thing* class as the *mk_older* method, along with the instance variable, age. The *mk_thing_older* function could be *inherited* as the *mk_older* method by subclasses of thing such as car (which subclasses also inherit the instance variable, age). Or the car subclass might override this "default" method by providing the *mk_car_older* function as the *mk_older* method for instances of car and its subclasses.

Also, as shown in Figure 12, the typing system correctly prevents an application of *mk_car_older* to thing. This is certainly a desirable result, since thing does not have the speed field assumed by *mk_car_older*. Thus, the typing system prevents an application of a polymorphic function to an argument for which it is unsuited. This is a necessary result in a useful typing system.

But is what we have shown so far really sufficient for capturing the essence of OOP? To answer this question, we pursue the consequences of the claimed correspondence of records to class instances and methods to record fields. We attempt to construct a record that corresponds to an instance of the thing class, as described above, and construct another record that corresponds to an instance of the car class. Some unexpected problems arise.

Figure 13 shows the types that result if the records of Figure 11 are extended to include the functions of Figure 12. The label names are motivated by our above description of the corresponding Smalltalk classes and *mk_older* methods. Note that these type expressions are recursive.

Unfortunately, the types in Figure 13 are not related as we had expected. If we assume that *car_type* ≤ *thing_type*, this leads to a contradiction. In particular,

$$(car\_type \le thing\_type) \Rightarrow (\forall o \le thing\_type.o \rightarrow o \le \forall c \le car\_type.c \rightarrow c)^{11}$$

violates the conditions for *car_type* ≤ *thing_type* (in particular, that typeof(*car_type.mk_older*) ≤ typeof(*thing_type.mk_older*)). This contradiction indicates the falsity of assuming *car_type* ≤ *thing_type* in this case.

This difficulty of placing polymorphic methods within records while maintaining the ordering of the record types is surprising, but there is an explanation. Recall that the ordering defined for record types was chosen to support the intuition that (for instance), if *car_type* ≤ *thing_type*, a description of a car can (by ignoring fields not found within things) be seen as a description of a thing. But should a polymorphic method designed for operation on cars necessarily be a method for a thing? We think not. The *mk_car_older* method provides a concrete counterexample, and this is exactly what the ordering on polymorphic types expressed using bounded universal quantification tells us.

Thus, if we locate polymorphic car methods within a record representing a car, we would not expect these methods to be valid for things in general, as required by Cardelli's ordering on record types. In fact, we should expect just the opposite—that there will be many functions on cars that are not functions on things because cars

[11] It is straightforward to verify this relationship by recalling the meaning of universal quantification in terms of intersected ideals. If car_type ≤ thing_type, then every intersection done for $\forall c \le$ car_type.$c \rightarrow c$ is done for $\forall o \le$ thing_type.$o \rightarrow o$ and more besides. Thus $(\forall o \le \text{thing\_type}.o \rightarrow o) \le (\forall c \le \text{car\_type}.c \rightarrow c)$.

Function Definitions:

$$\text{mk\_thing\_older} : \forall o \leq \text{thing\_type}.o \rightarrow o$$
$$= \lambda o \leq \text{thing\_type} \,.\, o \text{ with } ((\text{age},\text{age}+1))$$

$$\text{mk\_car\_older} : \forall c \leq \text{car\_type}.c \rightarrow c$$
$$= \lambda c \leq \text{car\_type} \,.\, c \text{ with } ((\text{age},\text{age}+1), (\text{speed},\text{speed}-1))$$

Example Usage:

mk_thing_older(thing) = <age=6>
mk_thing_older(car) = <fuel=20; age=11; speed=80>

mk_car_older(thing) = WRONG (Type error)
mk_car_older(car) = <fuel=20; age=11; speed=79>

Figure 12. Aging functions.

Types:

thing_type = <age: int; mk_older: $\forall o \leq$ thing_type.o$\rightarrow$o>

car_type = <age,fuel,speed: int; mk_older :$\forall c \leq$ car_type.c$\rightarrow$c>

Subtype Relations:

none -- thing_type and car_type are not related

Figure 13. Problems using record types as classes.

are more *special* than things. For this reason, Cardelli's ordering on record types does not allow us to place polymorphic methods within records describing objects while maintaining the desired ordering on these records.

Irrespective of this situation, we want to emphasize that bounded universal quantification is a powerful and useful framework within which to represent the types of functions that exhibit inclusion polymorphism. Bounded universal types provide a distinct improvement in ordering the types of polymorphic functions, and this can be useful when requiring strong type checking of programs that make use of inheritance. As an example of this, we compare the ordering of the types of *mk_thing_older* and *mk_car_older*, as given in Figure 12, with the results that would obtain without recognition of the polymorphic nature of these functions.

Using the basic definitions and orderings of Figure 11, if we were not to avail ourselves of bounded universal quantification, as in Figure 12 where *mk_thing_older* and *mk_car_older* were defined, we would probably represent the types of *mk_thing_older* and *mk_car_older* as shown now in Figure 14.

Whereas the bounded universal typings given in Figure 12 correctly show that the type of *mk_thing_older* is included in the type of *mk_car_older* (thus allowing us to apply *mk_thing_older* to a car), the above characterization of the types of these

```
mk_thing_older' : thing_type → thing_type
mk_car_older' : car_type → car_type
```

Figure 14. Alternative typings.

functions results in no ordering at all between these types. This is because the function constructor in the ideal model is antimonotonic in its first argument.[12] Thus if some function, say *g*, requires as an argument a function of type *car* → *car*, we cannot pass *mk_thing_older'* to *g*, since its type is not included in *car* → *car*. This points out the fact that for functions that support inclusion polymorphism, it would be a mistake not to attempt to express and use this information in their typings. Luckily, bounded universal quantification allows us to express this information for polymorphic functions and achieve the orderings we intuitively expect.

Cardelli's ordering on records is an expedient device for providing useful relationships between record types in Amber [Cardelli 1984a], a language that does not include bounded universal types. As we have shown, this ordering does not extend to supporting class instances with polymorphic methods. There are two approaches toward this "problem" that appear to make sense. First, one might simply give up associating polymorphic methods closely with the instance variables that describe an object (by not placing both within a record). This approach is suggested by the successful examples of Figures 11 and 12, and finds additional support in the work of Ait-Kaci [Ait-Kaci 1984; Ait-Kaci and Nasr 1986], which is reviewed in the next section.

The other approach is reminiscent of the example in Figure 6 and uses the recursive record definitions developed for Amber. By using a recursive definition (in which the thing defined is made available within the scope of the definition), fields of a record are available to functions located within the record. Of course, the resulting method types are no longer polymorphic. Figure 15 shows the result of using this approach.

This seems to be a reasonable approach. There is a straightforward mapping from the situation in which polymorphic methods are maintained separately from the records describing objects, and the situation illustrated in this example, in which such a polymorphic function has been moved inside the record by transforming it to make use of the recursive record definition. Perhaps method inheritance in Smalltalk could be viewed as making use of such transformations under the assumption of common (i.e., inherited) instance variable representations.

Since the approach shown in Figure 15 maintains the desired ordering on record types, FUN can make use of polymorphic functions defined over bounded ranges of these types. Records can thus be used to represent useful objects and associated nonpolymorphic methods for these objects, and the types of these records can be ordered to allow such records as arguments to polymorphic functions. This is a promising result, but it requires choosing the appropriate (most specific) polymorphic function for a given set of arguments. Although the inheritance hierarchy of Smalltalk supports this automatically, it could also be done explicitly by the programmer or be supported with the concept of discriminators as in CommonLoops (discussed in Section 3.3). This approach was not considered in FUN owing to the belief that bounded universal quantification combined with functional record fields provided all that was necessary for modeling inheritance in OOP. As shown, however, the claimed correspondence between records and objects fails when attempting to include inherited polymorphic methods in the analogy.

[12] That is to say, if domains are related as $A \leq B$ and $C \leq D$, then $B \rightarrow C \leq A \rightarrow D$ [MacQueen and Sethi 1982]. Although initially perplexing, this rule is correct in the ideal model, given that nothing else is known about the behavior of the functions involved. As an example, if a function requires for its argument a function of type NAT → INT and NAT ⊆ INT, then supplying a function of type INT → NAT should be acceptable, since a function of type INT → NAT will certainly map NATs to INTs. On the other hand, if an NAT → NAT is required, an INT → INT will not do, or vice versa.

Types:

```
thing_type = <age: int; mk_older: → thing_type>
car_type = <age,speed: int; mk_older: → car_type>
```

Type Relationships:

```
car_type ≤ thing_type
```

Values:

```
thing = rec self .
        <age=5; mk_older()=self with ((age, age+1))>

car = rec self .
        <age=10; speed=80; mk_older()=self with ((age, age+1) (speed, speed-1))>
```

Figure 15. A final example of records.

2.4 Term Orderings

In Ait-Kaci's dissertation [1984] and subsequent work [Ait-Kaci 1985; Ait-Kaci and Nasr 1986], an alternative approach toward OOP is based on generalizing and ordering first-order terms. Although these terms are used to represent objects, there is no attempt to hold methods within these terms; the functions and Prolog-like relations that perform or guide computations are kept separate from the first-order terms manipulated and created by these computations.

Ait-Kaci's approach replaces Cardelli's records with generalized first-order terms called Ψ-terms. These terms are structured data types consisting of the following:

(1) a *head symbol* that determines a class of objects (namely, all those objects in some chosen domain of interpretation whose description begins with the head symbol),
(2) optional *attributes* or *fields* that describe particular features possessed by the class of objects being described (these fields are identified by labels and are themselves given by Ψ-terms),
(3) optional *coreference constraints* that are used to signify equality constraints between attributes that are satisfied by the class of objects being described (this is a source of increased expressive power over records).

Aside from coreference constraints, which offer a useful and important mechanism for describing constraints on objects, Ψ-terms are different from first-order terms primarily because a Ψ-term with a given head symbol may have any number of fields (normally, the head symbol of a first-order term determines the arity of the term).

One difference between Cardelli's approach and that of Ait-Kaci (aside from the availability of coreference constraints in Ψ-terms and the ability to place functional fields in records) is the fact that the ordering relationship in Cardelli is on the types of records (not the records themselves), whereas Ait-Kaci's ordering is on Ψ-terms, which are actual objects of computation. Ait-Kaci's approach can be thought of as computing with types. Alternatively, it could be viewed as being somewhat similar to the use of prototypes, since in that approach objects themselves may be viewed as classes [Stein 1987].

Figure 16 shows some Ψ-terms. The last example shows a more complex structure

person
A term with a head and no attributes. Intuitively, this could represent the set of all persons (sic) -- that is, persons with any attributes whatsoever.

person(name => Jim)
A term with a head and a single attribute with the label "name". This could represent the set of people named Jim.

student(name => Alex; sex => male)
The set of all students named Alex that are male.

person(id => name(last => X: string)
father => person(id => name(last => X)))

Figure 16. Example Ψ terms.

that expresses the constraint that the last name of a person and that of the person's father must be the same.

The use of arrows is intended to be suggestive; Ψ-term labels may be considered to be functions that return a Ψ-term value for the indicated field.

The type symbols used for Ψ-term heads (e.g., *person*, *student*) are chosen from a partially ordered signature, where this ordering is intended to reflect set inclusion within the interpretation universe of objects described by the type symbols. The signature ordering is not based on representation—an important difference between this approach and Cardelli's record type ordering. An ordering on Ψ-terms is then defined in a manner similar to that used for records, in this case by reference to the ordering on the head symbols from which Ψ-terms are created. The resulting inclusion ordering on Ψ-terms supports the same intuitions concerning object descriptions as that for Cardelli's records. Figure 17 shows an example of type inclusion.

The expressive power of Ψ-terms with respect to objects is similar to that of records—the head symbols in Ψ-terms could be placed in record fields whose types are related according to the original relationship among head symbols. Although the ordering relationships would be between the resulting record types (as opposed to the records themselves), this need not result in practical differences in how the relationship is used (or could be used) in supporting inheritance.

2.4.1 A Calculus of Ψ-Terms

The ordering on the signature of type symbols can be very useful if the signature is a lattice. That is, if least upper bounds and greatest lower bounds are defined for any subsets of the type symbol signature, the set of Ψ-terms will also be a lattice and there will be a least upper bound (lub) and greatest lower bound (glb) for any pair of Ψ-terms. Most interestingly, the lub and glb operations on Ψ-terms turn out to be the natural extensions of generalization [Reynolds 1970] and unification [Robinson 1965] on first-order terms. Figure 18 presents an example signature that is a lattice and two types represented as Ψ-terms built from this signature. Because the signature is a lattice, these two types have a lub (generalization) and glb (unification) that are themselves types, and these are also presented.

The correspondence of Ψ-terms with first-order terms and of Ψ-term unification (i.e., the glb operation on Ψ-terms) with unification on first-order terms naturally suggests an extension of Prolog that makes use of Ψ-terms. The resulting language provides an excellent example of straightforward and theoretically sound integration of

Type Symbol Orderings:

```
student ≤ person
Austin ≤ cityname
Dallas ≤ cityname
"abc" ≤ string
"einstein" ≤ string
        ...
```

Ψ-terms:

```
t1 =
        student(
                id => name(last => X:string);
                lives_at => Y:address(city=>Austin);
                father => person(id => name(last => X);
                                 lives_at => Y));

t2 =
        person(
                id => name;
                lives_at => address(city => cityname);
                father => person);
```

Resulting Ψ-term Ordering:

```
t1 ≤ t2
```

Figure 17. Ψ-term inclusion.

inheritance with logic programming [Ait-Kaci and Nasr 1986]. Because polymorphic functions may also be defined over domains denoted by Ψ-terms (by using techniques similar to those employed on records), it seems that Ψ-terms may provide a pleasing base for integrating both logical and functional object-oriented computing. This direction of investigation is currently being pursued [Ait-Kaci et al. 1987] and appears very promising.

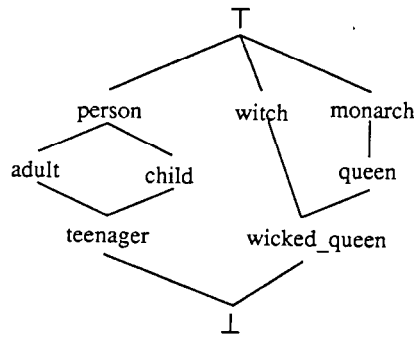
3. TYPES AND EXISTING OOP LANGUAGES

In this section we quickly review a few approaches and ideas relevant to type checking OOP for LISP-like and other languages. Our primary focus is the interesting connections between ad hoc suggestions that have been put forth for types in OOP and the ideas developed and implied by the models reviewed in this paper. By ad hoc, we mean type checking rules or language ideas put forth without an underlying model. Ad hoc rules are not necessarily less desirable for the lack of a model; it seems to us that if such rules lead to a consistent and useful programming methodology, there probably *is* a satisfactory model. Certainly LISP-like languages have been useful and important independently of the question of their model.

3.1 Type Checking Smalltalk

In the proposal of Johnson [1986], a number of alternatives to type checking Smalltalk programs are reviewed. Reasons why static checking would lead to important

A Signature:



Two Types:

```
child
        (knows => X:person(knows => queen; hates => Y:monarch);
        hates => child(knows => Y; likes => wicked_queen);
        likes => X);

adult
        (knows => adult(knows => witch);
        hates => person(knows => X:monarch; likes => X));
```

Their Generalization and Unification:

```
person
        (knows => person;
        hates => person(knows => monarch; likes => monarch));

teenager
        (knows => X:adult(knows => wicked_queen; hates => Y:wicked_queen);
        hates => child(knows => Y; likes => Y);
        likes => X);
```

Figure 18. A lattice signature and four types.

increases in program efficiency are outlined. The connection with this paper that we wish to highlight concerns the antimonotonic rule for function type inclusion described in Section 2.3.3.

Borning and Ingalls [1982] have suggested a natural characterization of the type of a variable in Smalltalk: the nearest common superclass of the objects that can be stored in that variable. Johnson rules this approach out because, among other reasons, it violates the antimonotonicity rule. Johnson's reasoning is as follows: Arrayof: Integer cannot be a subtype of Arrayof: Object, since a procedure that is given an Arrayof: Object can store any object in it, whereas only integers can be stored in an Arrayof: Integer. Therefore, a procedure that requires an Arrayof: Object as a parameter cannot accept an Arrayof: Integer. To see why antimonotonicity gives this result, we assume that Integer ≤ Object, and then examine the two respective array storage operations that we expect to find within Arrayof: Integer and Arrayof: Object, namely,

$$\textit{store\_int}: \text{Integer} \times \text{Arrayof: Integer} \rightarrow \text{Arrayof: Integer},$$

$$\textit{store\_obj}: \text{Object} \times \text{Arrayof: Object} \rightarrow \text{Arrayof: Object}.$$

Now, if Arrayof: Integer ≤ Arrayof: Object, the types of *store_obj* and of *store_int* are incomparable because of the antimonotonic ordering on function types.[13]

3.2 Emerald

Another language effort oriented toward typing OOP is described by Black et al. [1986]. Here again, antimonotonicity plays a major role in type checking. In a manner similar to Russell, an abstract type in Emerald defines a collection of *operation signatures*—operation names and the types of their arguments and results. All identifiers in Emerald are typed: The programmer must declare the abstract type of the objects that an identifier can name, and type checking amounts to verifying that assignments to identifiers *conform* to their types. In Emerald, a type S (read subtype) conforms to a type T if

(1) S provides at least the operations of T,
(2) for each operation in T, the corresponding operation in S has the same number of arguments and results,
(3) the abstract types of the results of S's operations conform to those of T's operations,
(4) the abstract types of the arguments to T's operations conform to those of S's operations.

Rules 1 and 2 correspond to Cardelli's ordering on record types, whereas rules 3 and 4 express the antimonotonicity of function types in terms of the Emerald programming metaphor.

As we have indicated in our examples, antimonotonicity can be a double-edged sword. There is no question that this rule prevents erroneous application of higher order functions to arguments for which they are unsuited. At the same time, however, we cannot help but feel that it is sometimes too restrictive and prevents a monotonic ordering when this would be both natural and useful. For example, it seems useful to consider natural numbers as *conforming* (in the above sense) to integers; yet if the addition operation on INT is types as INT × INT → INT, and the analogous, homogeneous operation on NAT is typed as NAT × NAT → NAT, then rule 4 is violated.[14]

Black does a nice job of clarifying the essential difference between type conformity in Emerald and subclasses of Smalltalk:

> In Emerald, the relationship between an object and the abstract type(s) that it implements is one of shared *interface*. An object supports a superset of the operations defined by its abstract type and each of the supported operations must conform to the corresponding operations in the abstract types. In Smalltalk, the relationship between a subclass and its superclass is one of shared *implementation*. A subclass is free to redefine the signatures of the messages that it receives, but it necessarily shares the superclass's representation (instance variables) and typically shares many methods as well. [Black et al. 1986, p. 80]

An aspect of Emerald that is appealing is the way in which the functions of Smalltalk classes have been *unbundled*. Smalltalk classes perform at least four functions: They express a specification hierarchy; they generate instances (the **new** method is available for all subclasses of Object); they act as a repository for the methods of an instance; and they express an implementation hierarchy. In Emerald, although there is no direct support for the specification hierarchy, the other functions are all mediated through separate mechanisms.

3.3 CommonLoops

The CommonLoops proposal [Bobrow et al. 1986] represents an attempt at merging the OOP style with the procedure-oriented

[13] As before, one way around this problem might be to "separate" the methods from the class definition, as done in CommonLoops and as suggested by Ait-Kaci's approach. If this is done, there is the possibility of using a polymorphic storage function, store: $\forall t \leq$ Object. $t \times$ Arrayof: $t \rightarrow$ Arrayof: t. Of course, this approach must be consistent with the way in which the programmer wishes to use the array.

[14] In some sense, we are almost disappointed that OOP systems that are not forced into the use of antimonotonicity (by an underlying model that requires it) have not come up with a better characterization of what constitutes type checking for OOP. We have a few thoughts on how one might relax the requirement for antimonotonicity, and these are mentioned in the conclusion.

philosophy of LISP. The objective is not so much to support type checking as to provide a consistent framework within which "types" may be used to choose the correct method for operating on a set of arguments. The important ideas introduced include *multimethods* and *discriminators*.

CommonLoops provides a **define-method** operation for defining a function applicable to specific types or "classes" of arguments. Classes, in turn, are defined using an extension of the Common LISP **defstruct** command. Thus, a class instance is similar to a record. Modifications to **defstruct** allow explicit importation of attributes from other classes, which are then treated as superclasses of the class being defined.

The **define-method** operation allows the programmer to indicate the intended types for all of its arguments. Therefore, unlike most other object-oriented schemes, CommonLoops allows method lookup to be based on more than the class of a single object. Such methods are called multimethods. Ultimately, many methods of the same name might be defined for manipulation of different classes of arguments. This set can be thought of as representing a generic function.

Associated with each generic function is a discriminator function whose purpose, given a generic function application, is to choose the most specific method suitable for operating on the given arguments, depending on their types. Often, a single method for each such set is defined without any type restrictions. This method then serves as a default method for its generic function.

Although the "types" of arguments passed to a generic function are used to determine the appropriate function to be applied, this facility is provided in an ad hoc manner, and no theory is used for determining (or checking) the result types of applications. If a theory providing a formal lattice-based ordering on types were available for this, our feeling is that the work of Ait-Kaci suggests a foundation for automatically determining the most specific method.

3.4 OakLisp and CommonObjects

Other interesting extensions of LISP dialects to support OOP include OakLisp and CommonObjects. OakLisp [Lang and Perlmutter 1986] is distinguished by its treatment of operations and object classes as first-class entities (objects) within the language. Types in OakLisp are regarded as sets of objects, and these types are themselves objects. OakLisp's designers have axiomatized the resulting type system and have shown that Russell's paradox (which can arise when dealing with a system whose elements may be members of sets represented by other elements) is not a problem. The key to this result is the existence of a total function *GetType*, which returns the smallest type containing the object to which it is applied. Such a smallest nontrivial type exists for every possible object in an OakLisp program. Like CommonLoops, OakLisp supports a general method lookup mechanism.

CommonObjects [Snyder 1987] is of specific interest here because of its concern with the tension resulting from supporting both encapsulation and inheritance in OOP. Inheritance in most OOP languages can inadvertently expose details of an object's implementation to its clients. This occurs in Smalltalk, for instance, when instance variables are inherited. In most OOP languages, it is not possible to define a data abstraction so that the internal variables of the implementation may be renamed without potentially affecting clients. In contrast with these languages, CommonObjects provides for full support of encapsulation with inheritance. The key to CommonObject's approach is that only methods may be inherited, and the language allows a designer to specify the type hierarchy independently of the inheritance hierarchy.

3.5 Exemplars versus Classes

The approach taken by CommonObjects highlights the potential for conflict between inheritance and information hiding. Others have perceived an equally important distinction between two ways of using inheritance. Inheritance in a system may be

used both for specification of common behavioral characteristics and for expressing common implementation representations. An interesting paper by LaLonde et al. [1986] suggests that the hierarchies for these separate purposes be expressed and maintained separately—as opposed to the situation for Smalltalk, in which they are indistinguishable. The result can be a system based on prototypical objects and behavioral specifications, or *examplars*, as LaLonde terms them.

Inheritance in FUN was based on inclusion polymorphism, and this in turn was based on an ordering on records. The objective and result were to allow functions to make use of shared representations between packages in an efficient and type-secure fashion. But there are other advantages to be achieved from the ability to express common (or inherited) behaviors even when the representations used to produce these behaviors are not shared. Primary among these include the ability to deal with the logical structure of a system independently of its implementation.

In Smalltalk, the fact that the relationships between classes and instances are reflected in the same hierarchy results in the following:

(1) All instances of a class must have identical representations and methods. Instances cannot have specialized methods, and multiple representations for instances are not supported.
(2) Specializations of classes with individualized representations are not allowed. Subclasses must have a representation that includes the superclass representation.
(3) Since the class hierarchy and instance hierarchy are intertwined, either the class hierarchy must be made to conform to the instance hierarchy, or vice versa.

A programming technique of using "abstract classes" that have no instance variables (such classes are intended solely for use as superclasses of other classes, i.e., never for direct instantiation) is common in Smalltalk and Flavors [Moon 1986] and addresses points (1) and (2). The exemplar-based system described by LaLonde et al. [1986] deals with all the above points and introduces a new form of inheritance (called OR-inheritance) as well.

4. SUMMARY

This paper has reviewed a variety of formal approaches to types that support in one way or another the notions of ADTs and inheritance. The last section highlighted some OOP systems that provide the practical benefits offered by these ideas, without necessarily providing an associated formal basis for their application to the checking and controlled derivation of system descriptions. Our indirect objective throughout has been to determine to what extent current research in the area of type theories can be utilized to provide such a basis, by providing straightforward representations for the objects and behaviors observed in OOP. Unfortunately, none of the currently available type theories appear perfectly suited to this task.

Although there are exciting areas of correspondence between currently available type theories and OOP languages, there are also mismatches. Although the theories reviewed produced languages with capabilities similar to those seen in OOP languages, none of these languages provides a complete set of capabilities directly equivalent in power to the OOP languages currently in use.

The algebraic approach supports inheritance in a direct fashion, but it is first order. Although record and Ψ-terms structures may be used to represent objects, polymorphic methods may not be included in these structures without destroying the desired subtype relationship between the structures. The appropriate response to this in the case of both record and term orderings was to avoid storing polymorphic methods in these structures, leaving only instance variables to be inherited.

The OOP language most closely resembling this approach is CommonLoops, in which methods are defined separately from

representational data structures and discriminators are used to select the most appropriate methods for performing application of generic functions. Although the record-based approach to subtyping in FUN could support an approach to inheritance similar to that seen in CommonLoops, this was never suggested in FUN, the language used to showcase that theory. On the other hand, it *is* expected that languages developed on top of the term-ordering model will make use of automatic compile-time discrimination for method selection. Operationally, this approach can provide the same advantages as method inheritance; but the CommonLoops approach does not assume inheritance of instance variables, whereas both the record- and term-oriented models of subtyping do.

CommonObjects, which is closer to the usual OOP model, expressly forbids inheritance of instance variables in the interest of strong data abstraction. It focuses instead on method inheritance as a central capability of OOP. Not discussed in this paper was the use of bounded existential quantification, which may be used to express *partially* abstract types [Cardelli and Wegner 1985]. This ability is interesting in this context because it allows limited knowledge concerning the representation type of packages to be made available. Although at first glance this capability might appear to be useful in providing a spectrum of possibilities with respect to data abstraction in the presence of inheritance—from completely open (as in the case of Smalltalk) to completely closed (as in CommonObjects)—in fact this is not the case. This is because inheritance based on ordering the types of records requires that these types *not* be abstract (i.e., inheritance in FUN is based entirely on representation) and therefore requires an open interface: thus, to the extent that existentials are used to hide the representation of object state in FUN, inheritance is not available.

None of the type theories reviewed consider the possibility of overriding inheritance. Although the use of separate hierarchies for behavioral and implementational description may alleviate the need for unprincipled use of this technique, overriding seems a natural approach to incremental enhancement of behavior when performed in a principled fashion. In the case of record or term type orderings, a discriminator-based approach to method selection could be used to achieve the desired effect—if the discriminators for generic functions can themselves be defined by the programmer (as in CommonLoops). The algebraic approach could be considered to support the same objectives as principled overriding through the device of theory extension.

All of the theories reviewed provide satisfactory models for ADTs. Although the approach of Russell toward support of ADTs seems more flexible and natural than the others reviewed, Russell's type system does not support inheritance. On a positive note, however, if Russell were to support subtyping, the result could bear a very strong resemblance to the method inheritance provided by CommonObjects, since types in Russell are sets of operations. Extending Russell in this way therefore seems a fruitful direction for research.

Almost all of the work to date on object-oriented languages and typing systems uses a single hierarchy for organizing specifications. On the one hand, this hierarchy is intended to capture generalization and specialization from a behavioral perspective; on the other, it is used to support composition of imperative implementations. That these two forms of hierarchy should not in general be equated follows from the following observations.

First, a given data structure and associated procedures that represent an abstraction are often extended by adding additional methods in order to represent another abstraction that is not behaviorally related to the first in a reasonable way. An example of this sort of anomaly has been identified in the Smalltalk-80 implementation [LaLonde et al. 1986]: In practice the class *Dictionary* is a subclass of *Set* owing to the intent to share the implementation of *Set* with *Dictionary*, but the *Dictionary* abstraction is not behaviorally a subclass of *Set*.

Second, composing fragments of imperative specification as is done with **sendsuper** does not in general preserve notions of behavioral equivalence or more general notions of behavioral ordering. This is especially true with respect to weak encapsulation under inheritance as discussed by Snyder [1987]. The use of a single hierarchy for two conceptually distinct relationships among specifications when coupled with the early binding implied by static typing leads to type systems that are very conservative when compared with the practice of Smalltalk or the various LISP object-oriented extensions.

It therefore seems necessary to acknowledge these hierarchies as independent means of organizing specifications. It is not adequate to make use of multiple inheritance. To see this, consider a subhierarchy rooted in the class *Matrix*. It is common in the literature to cite *Full_Matrix* and *Sparse_Matrix* as examples of the use of inheritance; so these are designed as subclasses of *Matrix*. (The examples in this paper of *Point* with subclasses *Cart_Point* and *Polar_Point* are similar.) Now, in actual problems involving matrices, such subclasses as *Positive_Matrix* and *Orthogonal_Matrix* are also relevant. Indeed, it may even be useful to restrict some algorithm to *Positive_Orthogonal_Matrix*. These notions are behavioral, and should be independent of the choice of implementation, that is, full or sparse. Using multiple inheritance within a single hierarchy that combines behavioral and implementation aspects means that the client of the class library must navigate through a class population that includes *Full_Positive_Orthogonal_Matrix* and *Sparse_Positive_Orthogonal_Matrix* when in fact the decision about full or sparse implementation should be made independently, possibly at a later time in the design and possibly even during execution of a program.

It is not farfetched to consider that there may be a variety of representations of full or sparse matrices, depending on the kinds of numbers and types of operations that will be performed. These issues must be addressed as object-oriented languages supported by type systems seek to bridge the gap between rapid prototyping and production quality implementations.

4.1 Desiderata for OOP

This paper generally assumes that OOP represents a positive step toward the design and implementation of complex software systems. In terms of this goal, our desiderata include the following.

(1) We want to support an object-oriented approach to the description of system components so that in the context of parallel and distributed computational systems we have a means of packaging, in a coherent manner, the elements of data that "go together" in a variety of computations.

(2) We want to support a more flexible and symmetric style of associating operations with objects than that exhibited by, say, Smalltalk so that we can define operations that make use of knowledge concerning the representations of all objects involved in a computation. Then, for instance, methods could be written that handle two argument vectors composed of different element types. On a parallel architecture, we could perform pairwise operations on respective vector elements followed by $O(\log\ n)$ reduction of the results.

(3) We want to support inheritance of description components in a systems description environment. This will provide a means of (a) enhancing productivity through reuse of description and (b) enhancing understandability through person-oriented classification techniques—generalization and specialization.

(4) We want to support the idea that one notion may have a variety of implementations because performance-oriented system description requires a means of specifying and selecting representations and algorithms (possibly instantiated in hardware) that are most appropriate in a given context. For example, whether to attempt an $O(\log\ n)$ reduction on a particular underlying architecture depends on the scale of the individual operations with respect to the scale of the allocation and control operations on that architecture.

(5) We want to incorporate an explicit typing system that can support but not require static typing of descriptions because flexibility in system description is closely related to the ability to control the time at which components are bound and the strength of such bindings. A typing system can provide much of the information needed to effect binding decisions properly.

4.2 Evaluation

Several systems address aspects of objectives (1)–(3) and even some of (4). Examples are CommonLoops [Bobrow et al. 1986] and OakLisp [Lang and Perlmutter 1986]. These systems are for the most part dynamically typed, offering little support (in a formal way) for control over the degree of binding that can be obtained from well-informed compilation. The metaclasses and generic functions of CommonLoops do provide a framework for addressing this issue. Most systems employing a formal type system do not support dynamic typing. An exception in this regard is Emerald. The type system that Emerald uses, however, does not appear to support objective (3). This is because Emerald views the instances of objects as incorporating the operations valid on those objects via the notion of a signature and further uses antimonotonic ordering on operation types. This prohibits such situations as passing a natural (number) to a routine that types its argument as an integer.

In many cases, an operation that is valid on an operand of type T should be valid on an object whose type is a subtype of T, although the method chosen to implement that operation might be a specialization of the generic method. This suggests that a different treatment of the association of operations of a type with objects of the type is needed in order to provide the appropriate expressiveness. Although the antimonotonic order is a valid order in some contexts, it is not appropriate when trying to capture the operational use of inheritance that is exhibited by Smalltalk and more generally CommonLoops and OakLisp. In these cases, we want to have a notion of being able to select the "most specific" instance of an operation for the given types of the arguments. This is essentially a mixture of inclusion polymorphism and operator overloading.

To capture the notion of the most specific operation applicable, we want to use a monotonic order so that, given $S \leq T$, operations that apply to objects of type S are more specific than those for T. This is essentially what the method lookup procedures of CommonLoops and OakLisp are able to accomplish. On the other hand, when considering a situation such as representation of an individual that incorporates an instance variable that is required to be of some function type, the antimonotonic order should apply to determine type correctness. Higher order function applications also require use of the antimonotonic ordering.

None of the approaches really address multiple implementations of the same abstraction in a fully effective manner. We basically contend that it is necessary to provide for a behavioral description separate from the (several) implementation descriptions of a notion. This provides a common yardstick against which to measure the implementations. This is not a new view, but one that seems to have fallen into disfavor as the focus has shifted to direct implementations of pure or nonimperative descriptive notations (e.g., OBJ). The need for such a separation has been recognized recently by others [LaLonde et al. 1986; Snyder 1987] and has been discussed above.

It is worth noting that the notion of multiple implementations actually has two dimensions: first, that there are separate representations and mechanisms for accomplishing the same thing and, second, that there may be different versions of the same approach (usually differing in time) that involve different compatibility requirements. The Cedar system provides advanced support for the second notion through the device of "configurations" [Swinehart et al. 1986].

The notion of existential types is intended to address such ideas, but it is not yet well integrated with inclusion polymorphism and overloading, nor is there

provision for behavioral description. The basic approach is one of representing an abstraction via a signature much as in Emerald. There are still technical difficulties regarding the status of the representation type.

The OBJ work is most advanced as a behavioral notation, and we believe that many of the techniques that are developed there can be fruitfully integrated with an operational notation that is used to give implementation descriptions and behavioral descriptions in separate hierarchies that are connected via pragma modules. These would describe the intensional applicability of the available implementations of a notion. The behavioral hierarchy could be oriented toward classifying notions in terms of common behavioral characteristics; the hierarchy of implementations would allow reuse of common representation and implementation mechanisms.

To conclude, we believe that both the theory and practice of OOP may be improved in substantial ways: More work is required in the area of type systems to provide the necessary basis for checking and controlled derivation of efficiently implemented systems using the OOP metaphor, and the practice of OOP itself, as represented by currently available OOP languages, should evolve by generalizing inheritance in order to make use of separate behavioral and implementation hierarchies. The OOP languages reviewed represent a variety of exciting departures from the model provided by Smalltalk. There are many inventive approaches to OOP that were not covered here because of the primary focus on type theories. With the growing interest in OOP, this trend of language development should continue. A future synthesis of formal typing systems with the powerful and flexible capabililties of OOP represents a goal requiring cooperation between theorist and practitioner; its attainment will be an important advance for computer science.

ACKNOWLEDGMENTS

We would like to express our appreciation for the supportive research environment provided by MCC—specifically the MCC Parallel Processing Program and its director, Steve Lundstrom, under whose direction this work was performed. In addition, we wish to thank the following researchers whose helpful comments concerning their work aided substantially in our review: Hassan Ait-Kaci, Hans Boehm, Luca Cardelli, and Joseph Goguen.

REFERENCES

AGHA, G. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, Mass.

AIT-KACI, H. 1984. A lattice-theoretic approach to computations based on a calculus of partially-ordered type structures. Ph.D. dissertation, Computer and Information Science Dept., Univ. of Pennsylvania, Philadelphia.

AIT-KACI, H. 1985. Integrating data type inheritance into logic programming. In Persistence in Data Types, Papers for the Appin Workshop (Aug.). Dept. of Computational Science, Univ. of St. Andrews, Scotland.

AIT-KACI, H., AND NASR, R. 1986. Logic and inheritance. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages* (St. Petersburg, Fla., Jan.). ACM, New York, pp. 219–228.

AIT-KACI, H., LINCOLN, P., AND NASR, R. 1987. A logic of inheritance, functions, and equations. Talk presented at joint U.S./Japan Workshop on Logic of Programs, Honolulu, Hawaii.

AMERICA, P. 1986. Operational semantics of a parallel object-oriented language. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages* (St. Petersburg, Fla., Jan.). ACM, New York, pp. 194–208.

AMERICA, P. 1987. POOL-T: A parallel object-oriented language. In *Object-Oriented Concurrent Programming,* A. Yonezawa and M. Tokoro, Eds. MIT Press, Cambridge, Mass.

BATES, J. L., AND CONSTABLE, R. L. 1985. Proofs as programs. *ACM Trans. Program. Lang. Syst. 7,* 1 (Jan.), 113–136.

BLACK, A., HUTCHINSON, N., JUL, E, AND LEVY, H. 1986. Object structure in the Emerald system. In OOPSLA Conference Proceedings (Portland, Oreg., Sept.). *ACM SIGPLAN 21,* 11 (Nov.). ACM, New York, pp. 78–87.

BOBROW, D., KAHN, K., KICZALES, G., MASIUTER, L., STEFIK, M., AND ZDYBEL, F. 1986. Common loops: Merging Lisp and object oriented programming. In OOPSLA Conference Proceedings (Portland, Oreg., Sept.). *ACM SIGPLAN 21,* 11 (Nov.). ACM, New York, pp. 17–29.

BORNING, A., AND INGALLS, D. 1982. A type declaration and inference system for Smalltalk. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages* (Albuquerque, N. Mex., Jan.). ACM, New York, pp. 133–141.

BRACHMAN, R. 1985. I lied about the trees. *AI Magazine* (fall).

BRINCH-HANSEN, P. 1977. *The Architecture of Concurrent Programs.* Prentice-Hall, Englewood Cliffs, N.J.

BURSTALL, R., AND GOGUEN, J. 1982. Algebras, theories and freeness: An introduction for computer scientists. CSR-101-82, Dept. of Computer Science, Univ. of Edinburgh, Scotland.

BURSTALL, R., AND LAMPSON, B. 1984. A kernel language for ADTs and modules. In *Semantics of Data Types*, G. Kahn, D. MacQueen, and G. Plotkin, Eds. Springer-Verlag, New York.

CARDELLI, L. 1984a. Amber. Tech. Memo 11271-840924-10TM, AT&T Bell Labs, Murray Hill, N.J.

CARDELLI, L. 1984b. A semantics of multiple inheritance. In *Semantics of Data Types*, G. Kahn, D. MacQueen, and G. Plotkin, Eds. Springer-Verlag, New York.

CARDELLI, L. 1986. A polymorphic lambda calculus with type: Type. DEC Systems Research Center Rep. No. 10, Digital Equipment Corp., Palo Alto, Calif.

CARDELLI, L. 1987. Basic polymorphic typechecking. *Sci. Comput. Program. 8*, 147–172.

CARDELLI, L., AND WEGNER, P. 1985. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv. 17*, 4 (Dec.), 471–522.

CARDELLI, L., AND MACQUEEN, D. 1985. Persistence and type abstraction. In Persistence in Data Types, Papers for the Appin Workshop (Aug.). Res. Rep. 16, Dept. of Computational Science, Univ. of St. Andrews, Scotland, pp. 231–240.

CONSTABLE, R., AND ZLATIN, D. 1984. The type theory of PL/CV3. *ACM Trans. Program. Lang. Syst. 6*, 1 (Jan.), 94–117.

COX, B. 1984. Message/object programming: An evolutionary change in programming technology. *IEEE Software 1*, 1 (Jan.).

CURRY, H., AND FEYS, R. 1958. *Combinatory Logic.* North-Holland, Amsterdam.

DAHL, O. J., AND NYGAARD, K. 1966. SIMULA—An ALGOL-based simulation language. *Commun. ACM 9*, 9 (Sept.), 671–678.

DALLY, W. 1986. A VLSI architecture for concurrent data structures. Ph.D. dissertation, Dept. of Computer Science, California Institute of Technology, Pasadena.

DEMERS, A., AND DONAHUE, J. 1979. Revised report on Russell. TR79-389, Dept. of Computer Science, Cornell Univ., Ithaca, N.Y.

DEMERS, A. J., AND DONAHUE, J. E. 1983. Making variables abstract: An equational theory for Russell. In *Proceedings of the 10th ACM Symposium on Principles of Programming Languages.*

DONAHUE, J., AND DEMERS, A. 1985. Data types are values. *ACM Trans. Program. Lang. Syst. 7*, 3 (Jul.), 426–445.

FUTATSUGI, K., GOGUEN, J., JOUANNAUD, J-P., AND MESEGUER, J. 1985. Principles of OBJ2. In *Proceedings of 12th Annual Symposium on Principles of Programming Languages* (New Orleans, La., Jan.). ACM, New York, pp. 52–66.

GIRARD, J. 1971. Une extension de l'interpretation de Gödel à l'analyse, et son application à l'èlimination des coupures dans l'analyse et la théorie des types. In *Second Scandinavian Logic Symposium*, J. E. Fenstad, Ed. North-Holland, Amsterdam.

GOGUEN, J., AND MESEGUER, J. 1986. EQLog: Equality, types, and generic modules for logic programming. In *Logic Programming*, D. DeGroot and G. Lindstrom, Eds. Prentice-Hall, Englewood Cliffs, N.J.

GOGUEN, J., AND MESEGUER, J. 1987. Unifying functional, object-oriented and relational programming with logical semantics. In *Research Directions in Object-Oriented Programming*, B. Shriver and P. Wegner, Eds. MIT Press, Cambridge, Mass.

GOLDBERG, A., AND ROBSON, D. 1983. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, Reading, Mass.

GUTTAG, J. 1980. Notes on type abstraction. *IEEE Trans. Softw. Eng. SE-6*, 1 (Jan.), 13–23.

HEWITT, C. 1985. The challenge of open systems. *Byte* (Apr.), 232–242.

HEXT, J. 1967. Compile time type-matching. *Comput. J. 9*, 365–369.

HOOK, J. 1984. Understanding Russell—A first attempt. In *Semantics of Data Types*, G. Goos and J. Hartmanis, Eds. Springer-Verlag, New York.

INGALLS, D. 1978. Smalltalk-76 programming system design and implementation. In *Proceedings of 5th ACM Symposium on Principles of Programming Languages* (Tucson, Ariz., Jan.). ACM, New York, pp. 9–15.

ISHIKAWA, Y., AND TOKORO, M. 1986. A concurrent object-oriented knowledge representation language Orient84/K: Its features and implementation. In OOPSLA Conference Proceedings (Portland, Oreg., Sept.). *ACM SIGPLAN 21*, 11 (Nov.), ACM, New York, pp. 232–241.

JOHNSON, R. 1986. Type-checking Smalltalk. In OOPSLA Conference Proceedings (Portland, Oreg., Sept.). *ACM SIGPLAN 21*, 11 (Nov.). ACM, New York, pp. 315–321.

JONES, N., AND MUCHNICK, S. 1976. Binding time optimization in programming languages. In *Proceedings of 3rd ACM Symposium on Principles of Programming Languages* (Atlanta, Ga., Jan.). ACM, New York, pp. 77–94.

KAPLAN, M., AND ULLMAN, J. 1980. A scheme for the automatic inference of variable types. *J. ACM 27*, 1 (Jan.), 128–145.

KAY, A. 1972. Smalltalk-72 instruction manual. Xerox PARC Rep. SSL-76-6, Xerox Palo Alto Research Center, Palo Alto, Calif.

LALONDE, W., THOMAS, D. T., AND PUGH, J. 1986. An exemplar based Smalltalk. In OOPSLA Conference Proceedings (Portland, Oreg., Sept.). *ACM SIGPLAN 21*, 11 (Nov.), 322–330.

LANDIN, P. J. 1966. The next 700 programming languages. *Commun. ACM 9*, 3 (Mar.), 157–166.

LANG, K., AND PERLMUTTER, B. 1986. OakLisp: An object-oriented scheme with first class types. In

OOPSLA Conference Proceedings (Portland, Oreg., Sept.). *ACM SIGPLAN 21*, 11 (Nov.), 30–37.

LIEBERMAN, H. 1986. Using prototypical objects to implement shared behavior in object oriented systems. In OOPSLA Conference Proceedings (Portland, Oreg., Sept.). *ACM SIGPLAN 21*, 11 (Nov.), 214–223.

LISKOV, B., SNYDER, A., ATKINSON, R., AND SCHAFFERT, C. 1977. Abstraction mechanisms in CLU. *Commun. ACM 20*, 8 (Aug.), 564–576.

MACQUEEN, D. 1985. Modules for ML. *Polymorphism Newsletter* (Oct.).

MACQUEEN, D. 1986. Using dependent types to express modular structure. In *Proceedings of 13th Annual ACM Symposium on Principles of Programming Languages* (St. Petersburg Beach, Fla., Jan.). ACM, New York, pp. 277–286.

MACQUEEN, D., AND SETHI, R. 1982. A higher order polymorphic type system for applicative languages. In *Proceedings of 1982 Symposium on Lisp and Functional Programming*. ACM, New York, pp. 243–252.

MACQUEEN, D., PLOTKIN, G., AND SETHI, R. 1984. An ideal model for recursive polymorphic types. In *Proceedings of 11th Annual ACM Symposium on Principles of Programming Languages* (Salt Lake City, Utah, Jan.). ACM, New York, pp. 165–174.

MARTIN-LÖF, P. 1982. Constructive logic and computer programming. In *Proceedings of 6th International Congress for Logic, Methodology, and Philosophy of Science*. North-Holland, Amsterdam.

MATTHEWS, D. 1983. Programming language design with polymorphism. Ph.D. dissertation, Computer Lab., Univ. of Cambridge, Cambridge, England.

MESEGUER, J., AND GOGUEN, J. 1983. Initiality, induction, and computability. Tech. Rep. CSL-140, Computer Science Laboratory, SRI, Menlo Park, Calif.

MEYROWITZ, N., ED. 1986. Intermedia: The architecture and construction of an object-oriented hypermedia system. In OOPSLA Conference Proceedings (Portland, Oreg., Sept.). *ACM SIGPLAN 21*, 11 (Nov.), 186–201.

MEYROWITZ, N., ED. 1987. OOPSLA Conference Proceedings (Orlando, Fla., Oct.). *ACM SIGPLAN 22*, 12 (Dec.).

MILNER, R. 1978. A theory of type polymorphism in programming. *J. Comput. Syst. Sci. 17*, 348–375.

MITCHELL, J. 1984a. Coercion and type inference (Summary). In *Proceedings of 11th Annual ACP Symposium on Principles of Programming Languages* (Salt Lake City, Ut., Jan.). ACM, New York, pp. 175–185.

MITCHELL, J. 1984b. Type inference and type containment. In *Semantics of Data Types*, Lecture Notes in Computer Science, vol. 173. Springer-Verlag, New York, pp. 51–67.

MITCHELL, J., AND PLOTKIN, G. 1985. Abstract types have existential type. In *Proceedings of 12th Annual ACM Symposium on Principles of Programming Languages* (New Orleans, La., Jan.). ACM, New York, pp. 37–51.

MOON, D. 1986. Object-oriented programming with flavors. In OOPSLA Conference Proceedings (Portland, Oreg., Sept.). *ACM SIGPLAN 21*, 11 (Nov.), 1–8.

PARNAS, D. 1972. A technique for software module specification. *Commun. ACM 15*, 5 (May), 330–336.

REYNOLDS, J. 1970. Transformational systems and the algebraic structure of atomic formulas. In *Machine Intelligence*, D. Michie, Ed., vol. 5. Edinburgh University Press, Edinburgh, Scotland, chap. 7.

REYNOLDS, J. 1974. Towards a theory of type structure. In *Colloquium sur la Programmation*, Lecture Notes in Computer Science, vol. 19. Springer-Verlag, New York.

ROBINSON, J. 1965. A machine-oriented logic based on the resolution principle. *J. ACM 12*, 1 (Jan.), 23–41.

SCHAFFERT, C., COOPER, T., BULLIS, B., KILIAN, M., AND WILPOLT, C. 1986. An introduction to Trellis/Owl. In OOPSLA Conference Proceedings (Portland, Oreg., Sept.). *ACM SIGPLAN 21*, 11 (Nov.), 9–16.

SCHERLIS, W. 1986. Abstract data types, specialization, and program reuse. In *Proceedings of the International Workshop on Advanced Programming Environments*. Springer-Verlag, New York.

SCOTT, D. 1976. Data types as lattices. *SIAM J. Comput.* (Sept.), 522–587.

SHRIVER, B., AND WEGNER, P., EDS. 1987. *Research Directions in Object-Oriented Programming*. MIT Press, Cambridge, Mass.

SKARRA, A., AND STEIN, J. 1987. Type evolution in an object-oriented database. In *Research Directions in Object-Oriented Programming*, B. Shriver and P. Wegner, Eds. MIT Press, Cambridge, Mass.

SNYDER, A. 1987. Inheritance and the development of encapsulated software components. In *Proceedings of the 20th Hawaiian International Conference on Systems Sciences*. Software Track, Western Periodicals, North Hollywood, Calif., pp. 227–238.

STEFIK, M., AND BOBROW, D. 1986. Object-oriented programming: Themes and variations. *AI Mag. 6*, 4, 40–62.

STEIN, L. 1987. Delegation is inheritance. In OOPSLA Conference Proceedings (Orlando, Fla., Oct.). *ACM SIGPLAN 22*, 12 (Dec.), 138–146.

STOY, J. 1977. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, Mass.

STROUSTRUP, B. 1986. *C++*. Addison-Wesley, Reading, Mass.

SWINEHART, D., ZELLWEGER, P., AND BEACH, R. 1986. A structural view of the Cedar programming environment. *ACM Trans. Program. Lang. Syst. 8*, 4 (Oct.), 419–489.

TURNER, R. 1984. *Logics for Artificial Intelligence.* Halsted Press, New York.

U.S. DEPARTMENT OF DEFENSE. 1983. *Ada Reference Manual.* ANSI/MIS-STD 1815, U.S. Printing Office (Jan.), Washington, D.C.

YONEZAWA, A., AND TOKORO, M., EDS. 1987. *Object-Oriented Concurrent Programming.* MIT Press, Cambridge, Mass.

YONEZAWA, A., SHIBAYAMA, E., TAKAKA, T., AND HONDA, Y. 1987. Modelling and programming in an object oriented concurrent language ABCL/1. In *Object-Oriented Concurrent Programming*, A. Yonezawa and M. Tokoro, Eds. MIT Press, Cambridge, Mass.

Received February 1987; final revision accepted February 1988.