On Type Systems for Object-Oriented Database Programming Languages

YURI LEONTIEV
Intuit Canada Limited
M. TAMER ÖZSU
University of Waterloo
and
DUANE SZAFRON
University of Alberta

The concept of an object-oriented database programming language (OODBPL) is appealing because it has the potential of combining the advantages of object orientation and database programming to yield a powerful and universal programming language design. A uniform and consistent combination of object orientation and database programming, however, is not straightforward. Since one of the main components of an object-oriented programming language is its type system, one of the first problems that arises during an OODBPL design is related to the development of a uniform, consistent, and theoretically sound type system that is sufficiently expressive to satisfy the combined needs of object orientation and database programming.

The purpose of this paper is to answer two questions: "What are the requirements that a modern type system for an object-oriented database programming language should satisfy?" and "Are there any type systems developed to-date that satisfy these requirements?". In order to answer the first question, we compile the set of requirements that an OODBPL type system should satisfy. We then use this set of requirements to evaluate more than 30 existing type systems. The result of this extensive analysis shows that while each of the requirements is satisfied by at least one type system, no type system satisfies all of them. It also enables identification of the mechanisms that lie behind the strengths and weaknesses of the current type systems.

Categories and Subject Descriptors: D.3.2 [**Software**]: Programming Languages—*Language Classifications*; D.3.3 [**Software**]: Programming Languages—*Language Constructs and Features*; H.2.3 [**Information Systems**]: Database Management—*Languages*; F.3.3 [**Theory of Computation**]: Logics and Meaning of Programs—*Studies of Program Constructs*

General Terms: Design; Languages; Theory;

Additional Key Words and Phrases: OODB, OODBPL, Object-oriented database programming language, Type checking, Type system, Typing

Authors' addresses: Y. Leontiev, Intuit Canada Limited, 7008 Roper Road, Edmonton, Alberta T6B 3H2, Canada; Yuri_Leontiev@intuit.com
M. T. Özsu, University of Waterloo, School of Computer Science, Waterloo, Ontario N2L 3G1, Canada; tozsu@uwaterloo.ca
D. Szafron, University of Alberta, Department of Computing Science, Edmonton, Alberta T6G 2H1, Canada; duane@cs.ualberta.ca

1. INTRODUCTION

From its early days, object orientation (OO) was considered to be one of the most influential and useful programming paradigms. Its impact on research in virtually all areas of computing science can only be compared to that of relational algebra, or that of the functional and logic programming paradigms. Much of the power of object orientation lies in the fact that it provides conceptual and modeling capabilities that allow it to express real-world entities with relative simplicity. Another source of the appeal of object orientation is its support for incremental software construction, provided in the form of code reuse.

Object-oriented database programming languages (OODBPLs) have the potential to combine, in a single uniform framework, the modeling and software construction power of the object-oriented paradigm, extensive and efficient data storage and retrieval techniques of modern database systems, and the efficiency and power of today's programming languages.

Both the modeling and software construction powers of object-oriented languages are rooted in their type and inheritance systems. A properly designed, rich, and theoretically sound type system can greatly increase the power of a language, while a poor, inflexible type system can render almost all power inherent in the object-oriented paradigm useless.

While the type system of an object-oriented language greatly affects its characteristics, the type system of an OODBPL affects its characteristics even more. The reason is the presence of different and sometimes contradictory requirements that are imposed on the type system by an OODBPL's database and programming language components.

The purpose of this paper is to answer two key questions: "What are the requirements that a modern type system for an object-oriented database programming language should satisfy?" and "Are there any type systems developed to-date that satisfy these requirements?".

In order to answer the first question, the set of requirements put forth in the literature for type systems (Section 2.2), modern object-oriented programming languages (Section 2.3), database programming languages (Section 2.4), and object-oriented database systems (Section 2.5) is considered to yield a set of required type system features. The resulting combination of requirements is presented in Section 2.6. Section 2.7 presents the test suite that is used to evaluate the existing type systems reviewed in this paper.

Section 3 presents an extensive review of more than 30 languages and type systems. These type systems are evaluated with respect to the requirements and the test suite presented in Section 2.6. The result of this extensive analysis (presented in Section 4) shows that while each of the requirements is satisfied by at least one type system, no type system satisfies all of them. Section 4 also identifies the mechanisms that lie behind the strengths and weaknesses of the current type systems. The knowledge obtained this way can be used to aid in the design and development of a type system that satisfies all of the above requirements (e.g. [Leontiev 1999]), even though a new language design is not the focus of this paper.

2. TYPE SYSTEM REQUIREMENTS

2.1 Terminology

We start by establishing some terminology to be used throughout the rest of this paper. This is necessary as many of the terms used in the object-oriented language research area have no clear definition and are used differently by different authors. We note that this is not a tutorial on object-oriented concepts, but only a consistent set of definitions (even if they are not universally accepted) that we use throughout this paper. Most of the terminology below comes from [Cardelli 1989; Black and Palsberg 1994].

Object. A primitive term for a data item used to model a concept or a real-world entity.

Method. A procedure to be executed when an object is sent an appropriate message.

Message. A part f of an invocation $x.f$, where x is an object; an identification for a related set of methods.

Function name. A part f of invocation $f(x)$; also can be termed as a message not associated with any type. Sometimes called a *free-floating function.*

Function. A procedure to be executed when a function invocation is requested. Function relates to a function name the same way a method relates to a message.

Interface type. A description of messages applicable to an object.

Implementation (representation) type. A description of an object's structure.

Type. A shorthand for interface or implementation type or both, depending on the context.

Mutable object type. A type of an object that is capable of changing its state at run-time. For example, variables and arrays are mutable objects. These types are sometimes called *imperative types.*

Parametric (parameterized) type (message, function). A type (message, function) that describes a family of types (messages, functions) by using (an) explicit parameter(s). For example,

```
type T_List(X) {
      getAt(T_Integer): X;
};
```

Constrained type. A parametric type that places a constraint on its parameter(s). The mechanism that allows a programmer to specify such a type is called *bounded quantification.* For example

```
type T_PersonList(X) where X subtype of T_Person;
```

specifies a constrained parametric type `T_PersonList`.

Intersection (greatest lower bound) types. An intersection of a set of types is a type that represents the greatest lower bound of the set in the type lattice. Intersection types are useful for typing of set-theoretic operations and queries.

Union (lowest upper bound) types. A union of a set of types is a type that represents the lowest upper bound of the set in the type lattice. Union types are useful for typing of set-theoretic operations, queries, and conditionals.

Inheritance. A mechanism for making one (interface or implementation) type from another. A *single inheritance* requires that a type has at most one immediate ancestor (parent) in the inheritance chain; *multiple inheritance* lifts this restriction.

Interface subtyping. A partial order on interface types. An interface type A is a subtype of another interface type B when an object of the interface type B can be thought of as an object of the interface type A (e.g., a type `T_Student` is a subtype of `T_Person` since every student can be thought of as a person).

Implementation subtyping (code reuse). A partial order on implementation types. An implementation type A is a subtype of another implementation type B if it is possible to use code written for B to manipulate objects that have the implementation type A.

Structural subtyping. Subtyping automatically determined by object structure. Sometimes termed as *implicit subtyping.* Its opposite, *explicit* or *user-definable subtyping*, relies on user-supplied subtyping clauses to determine the subtyping relationship between types.

First-class object. An object that is capable of receiving messages.

Non-first-class object (value). An immutable object that lacks the ability to receive messages. Traditionally, it is assumed that values have no interface, just a set of operations defined on them. In this paper, such a set of operations will be considered an interface of a value.

Primitive (atomic) type. A primitive type is a type of basic (primitive) system-defined objects or values (such as integers, reals, characters, etc.). Primitive types usually have a special status in non-uniform systems and languages.

Soundness of a type system. Inability of a successfully typechecked program to produce run-time type errors. Sometimes divided into *static* and *dynamic* soundness. Dynamic soundness is weaker than static soundness. It makes sense for languages with dynamic type checking and assures that run-time type errors will be caught.

Verifiability of a type system (decidable typechecking). The ability to *verify* that a program does not contain type errors with respect to a particular type system; equivalently, the presence of a decidable typechecking algorithm. Note that verifiability does not imply soundness. For example, the language Eiffel [Meyer 1988] has a decidable typechecking algorithm, but a successfully typechecked program can produce run-time type errors.

Substitutability. A property of a type system (language) that guarantees that an object of a subtype can be used everywhere the object of its supertype can (e.g. if a type `T_Person` is a supertype of `T_Student`, then a student can be legally used wherever a person can be).

Dispatch. A process of finding out at run-time which *method* of a particular *message* to execute. *Single dispatch* bases its decision on the type of the first argument (receiver) only, while *multiple dispatch* takes into account types of other arguments as well. The term *static dispatch* will be used to refer to a compile-time process of method determination.

Static/dynamic typing (typechecking). Static typechecking is done at compile-time, while dynamic typechecking is done at run-time.

Implicit/explicit typing. Languages with explicit typing require the programmer to insert type annotations in the program. Languages with implicit typing *infer* the types of expressions without any help from the programmer. Intermediate approaches are also possible.

Inclusion polymorphism. This term refers to a combination of subtyping and substitutability. Inclusion polymorphism allows an object to be viewed as belonging to many different types that need not be disjoint; that is, there may be inclusion of types.

Parametric polymorphism. Parametric polymorphism is present when parametric specifications (e.g., parametric types) are supported. This kind of polymorphism allows for specifications that take types as parameters.

Covariance (contravariance, novariance). Covariance means that changes in a particular type are parallel to the direction of the type hierarchy. In the following example the result type of the method `getAt` changes *co*variantly, as it is `T_Person` in definition 1 (which occurs in the type `T_PersonList`) and `T_Student` (a *subtype* of `T_Person`) in definition 2 (which occurs in the type `T_StudentList`, a *subtype* of `T_PersonList`).

```
type T_Student subtype of T_Person;

type T_PersonList {
      getAt(T_Integer): T_Person;  // 1
};

type T_StudentList subtype of T_PersonList {
      getAt(T_Integer): T_Student; // 2
};
```

The reverse direction is termed as *contravariant*. *Novariance* forbids any type changes along the type hierarchy. In this example, the argument type `T_Integer` changes novariantly, i.e. does not change at all.

The terms *class* and *subclassing* are deliberately avoided in this paper as they are too overloaded. In object-oriented literature, the term *class* is used to denote a mechanism for object construction, an equivalent of an implementation type, an equivalent of an interface type, a set of objects satisfying a particular condition, or some combination of the above four notions. The definition of *subclassing* is equally overloaded.

Also, the term *type* is qualified as being either interface type or implementation type. This will prove useful when considering type systems where the two type notions are distinct.

2.2 Essential features of a type system

The major goals of a type system in today's programming languages and database systems include [Cardelli 1989; 1997; Black and Palsberg 1994]:

(1) Provide a programmer with an efficient way of catching programming errors *before* a program (or a part of it) is actually executed. This is often considered to be the major objective of a type system in the programming language

community.

(2) Serve as a data structuring tool for design and modeling purposes. Many design technologies that have emerged through the past decade rely partially or fully on type systems to provide a convenient design and documentation framework for a system development process. This is especially true of object-oriented design technologies. This property is often considered to be the major goal of a type system in the database community.

(3) Provide a convenient framework for program maintenance. This includes documentation at the production stage of program evolution as well as the ability of a programmer to understand the functionality and interfaces of a completed product.

(4) Provide sufficient information for optimization purposes. The information provided by the type system can be used by an optimizing compiler, interpreter, or a query optimizer (in persistent systems) to improve the efficiency of a program.

In order for a compiler (or interpreter) to be able to typecheck a program (or a part of it), there must exist a *typechecking algorithm*. Existence of such an algorithm for a given type system is termed as *verifiability* of the type system [Cardelli 1997]. Thus, a type system should be *verifiable*. It is preferable that a type system should be *decidably verifiable*; however, one may have to put up with an undecidable type system just as one puts up with undecidable programs if enough expressive power is desired [Black and Palsberg 1994]. The verifiability property also implies that a type system should be *provably sound*, i.e. there should exist a formal proof that a successfully type-checked program does not generate any type errors at run-time.

If a compiler finds a type error and reports it to a programmer, the latter should have sufficient information to be able to understand the reason for the type error in order to correct it. Thus, the type system should be *transparent*.

A type system should also be *enforceable* in order to prevent an execution of type-incorrect programs. This implies that programs have to be written with as much type information as possible to prevent "false alarms".

Finally, a type system should be *extensible*. This requirement stems from the fact that none of the existing type systems were found to be satisfactory for all possible applications. Therefore the chance that any new type system will satisfy *all* application domains is remote. If a type system can not be extended, it will sooner or later be abandoned for a type system that can adapt to new application requirements. Switching from one type system to another is extremely costly both in terms of people resources (that have to be reeducated) and in terms of data conversion costs.

2.3 Object-oriented programming language requirements

Pure object-oriented programming languages pose some specific requirements on their type systems. These requirements will be constructed by considering features essential for pure object-oriented languages and reformulating them in terms of type system features.

For a language to be called *a pure object-oriented language*, it should possess at least the following properties, some of which are often claimed as absolutely necessary for future object-oriented languages [Tsichritzis et al. 1992]:

(1) Encapsulation. This property is usually considered as one of the characteristic features of object-oriented languages and greatly facilitates code reuse. It refers to the ability of a language to shield internals of an object implementation from outside access.

(2) Inheritance. This is a characteristic property of object-oriented languages as well. The inheritance mechanism promotes and facilitates well-structured software design and reusability of the code. Multiple inheritance is highly desirable, as its absence leads to clumsy or limited type specification in some important cases.

(3) Uniformity. Primitive values (integers etc.), types, and messages (methods) should be first-class objects. If this requirement is not satisfied, the language will have to handle the non-object entities in some non-object-oriented way, and will therefore not be *purely* object-oriented. Note that the existence of methods that are able to operate on types and other methods is a consequence of this property. This is also advocated in [Hauck 1993].

(4) Object access and manipulation uniformity. An object can only be manipulated by methods defined for it. Together with uniformity, this property provides for purely object-oriented programming.

(5) Method uniformity. This refers to the absence of distinction between *stored* and *computed* methods or, equivalently, the absence of public instance variables. This requirement is important as its violation breaks encapsulation and may effectively hinder the usefulness of the code reusability provided by inheritance.

(6) Separation between interface and implementation inheritance (sometimes termed as separation between type and class hierarchies). This is actually a consequence of encapsulation, inheritance, and object manipulation uniformity. Additional arguments in favor of such separation can be found in [LaLonde and Pugh 1991; Castagna 1996; Taivalsaari 1996; Baumgartner et al. 1996; Leontiev et al. 1998].

(7) Multi-methods (multiple dispatch). This refers to the ability of a language to use types of all arguments during dispatch. Traditionally, only the type of the first argument (the receiver) is used. This property of the language is essential to adequately model binary methods [Bruce et al. 1996; Castagna 1996; Leavens and Millstein 1998; Fisher and Mitchell 1996] and certain object-oriented design patterns [Baumgartner et al. 1996].

Using these requirements for pure object-oriented languages, it is now possible to formulate desirable features of type systems for such languages. These requirements are:

(1) Inheritance mechanisms for both interface and implementation inheritance. This requirement is a direct consequence of the language requirements 2 and 6 above.

(2) Type system reflexivity. This is necessary to ensure uniformity of the language (requirement 3), since types (classes) have to be objects[1]. Since every object

[1]The term "type system closure" is sometimes used to denote this property.

has a type, types and classes need to have types as well. Thus, the type system needs to be reflexive.

(3) Method types. This is also a consequence of uniformity. Indeed, since methods have to be objects to ensure uniformity, they will have types. Moreover, when methods are manipulated as objects (e.g., passed as arguments to other methods), their types should be descriptive enough to ensure the validity of type-checking.

(4) Method uniformity at the type level (no distinction between types of *stored* and *computed* methods). This is a direct consequence of the language requirement 5.

(5) Support for multi-methods (multiple dispatch). This is a consequence of the language requirement 7.

(6) Substitutability (at least for interface inheritance). This property is essential to support extensibility and inclusion polymorphism. The latter is used to achieve one of the primary goals of the object-oriented paradigm: code reuse.

2.4 Database programming language requirements

Database programming languages (DBPLs) possess their own set of distinguishing features that poses additional requirements on type systems. The approach of the previous subsection will be used to derive the type system features from the following list of necessary features of a persistent language that is taken[2] from [Atkinson and Buneman 1987; Atkinson and Morrison 1995].

(1) Persistence independence (the form of the program is independent of the longevity of data the program operates upon). This is necessary to provide seamless integration between the database and the language and to significantly reduce the amount of code necessary to deal with persistent data.

(2) Orthogonality of type and persistence (data of all types can be persistent as well as transient). This is an aid to data modeling as it ensures that the model can be independent of the longevity of data. It also eliminates the need of explicit persistent-to-transient data conversions.

(3) User-defined type constructors. This requirement is due to the necessity of modeling new, potentially complex data structures in a uniform and consistent manner.

(4) Information hiding (also known as encapsulation). Encapsulation allows for data modeling at a higher (more abstract) level as it hides the implementation details and gives the programmer an ability to deal with abstract interfaces rather than concrete data structures. It greatly facilitates modeling, code reuse, and component integration.

(5) Polymorphism (parametric, inclusion, or both). Parametric and inclusion polymorphisms make the specifications more succinct and precise. They also allow for a significant reduction in the amount of code that needs to be written to specify and implement a particular data model, as a significant portion of the specifications are reused via genericity achieved by the use of polymorphic constructs.

[2]Features not related to type system are dropped from the list.

(6) Static and strong typing with provisions for partial type specification (which necessitates the presence of a type mechanism similar to one formed by type constructor `dynamic` and operator `typecase`). This is necessary in order to deal with data that come from a persistent store whose structure is only partially known at the time the program is written. An example of a program that requires such capabilities is a generic database browser that is supposed to work on any database independently of its structure.

(7) Incremental program construction and modularity. This principle ensures that most of the program modifications can be done locally, without affecting the rest of the code. While this property is very important for programming languages in general, it is even more important for database programming, since databases tend to exist and evolve for extensive periods of time. As a database evolves, the programs designed to operate on it have to evolve as well. Modularity is one of the major features that significantly reduce the overhead of such an evolution.

(8) Query facilities. One of the main reasons behind the success of the relational data model was its ability to support declarative, simple, yet powerful data access/query languages. In order for object systems to be successful, they must provide querying capabilities equal or exceeding those of the relational systems.

(9) Ability of a program to deal with state change. This requirement is necessary as persistent data outlive the program and if a program is not able to change data, the state of persistent data will never change.

From this list of requirements, the following properties of the type system can be derived:

(1) Types (classes) in the type system should not be specified as either persistent or transient. This is the DBPL requirement 2 reformulated in terms of type system terminology.

(2) User-defined type constructors. This is the same as the DBPL requirement 3.

(3) Encapsulation. This follows from the DBPL requirement 4.

(4) The presence of parametric types. This follows from the DBPL requirements 5 (parametric polymorphism) and 3. It is also desirable to handle bounded (constrained) parametric types as it increases the modeling power of the type system.

(5) Possibility of partial type specification and dynamic type checking. This follows from the DBPL requirement 6.

(6) Verifiable and sound type system (as a consequence of the DBPL requirement 6).

(7) Incremental type checking (as a consequence of the DBPL modularity requirement).

(8) The ability of the type system to correctly type declarative queries. This stems from the DBPL requirement 8. According to [Buneman and Pierce 1999], this requires that the type system can support union (least upper bound) types.

(9) The ability of the type system to deal with types of mutable objects (later referred to as mutable object types) and assignment. This is a direct consequence of the DBPL requirement 9.

In addition to the above, [Kirby et al. 1996] advocates the use of reflection in persistent object systems.

The combination of object-orientedness and persistence poses some additional requirements described in the next subsection.

2.5 Object-oriented database system requirements

A list of features needed or desirable in object-oriented database management systems (OODBMS) and the rationale behind it are given in [Atkinson et al. 1992]. This is the most comprehensive of such lists published so far. The following list[3] is the part of it that is related to type system issues. It contains additional requirements to those already listed in the previous subsections.

(1) Complex objects (orthogonal type constructors should include at least sets, tuples, and lists). This is necessary to ensure that the modeling power of the system is sufficient to deal with modern applications, such as CAD/CAM, medical and geographical information systems.

(2) Extensibility: user-defined and system types should have the same status and the user should be able to add new "primitive" types to the system. This is also due to the necessity of dealing with new demanding application areas. It is impossible to anticipate all the data types that will be required to model the data structures in those areas since some of them are yet to emerge. By providing the same status to system and user-defined types, the system guarantees that its capabilities are not decreased when it is applied to a new application domain. Extensibility is also advocated in [Matthes and Schmidt 1991].

(3) Views. A view is, in a sense, an ability to transparently change the appearance of data for different users (clients). The importance of this concept as well as its usefulness and power have been convincingly demonstrated by years of experience with relational databases.

(4) Dynamic schema evolution. This requirement is based on the necessity to maintain (and change) the database structure over extensive periods of time. While it is sometimes possible to create a completely new database with a new schema and migrate data to it, this approach is usually quite expensive and results in a substantial down time, which may not be acceptable in large distributed applications such as hospital or banking systems. Dynamic schema evolution makes it possible to change a database structure transparently to its users.

These additional requirements have to be taken into account when designing a type system for a pure object-oriented database programming language. Next, the requirement summary will be presented and a short overview will be given.

[3]Some issues often considered as deficiencies of object-oriented systems (for example, in [Kim 1993]) but deemed optional in [Atkinson et al. 1992] are listed here as mandatory. The reason for that is the understanding that if object-oriented databases are to be the next step in the database development, they should utilize the advances already made in relational databases.

2.6 Summary of requirements

The following is the compilation of all type system requirements presented so far. The categorization of the requirements presented here is subjective, but it does provide a useful structure to the extensive set of requirements compiled so far. Each requirement listed below contains a reference to the section where it has been introduced and explained.

(1) Theoretical requirements
   (a) Verifiability (preferably decidable) and provable soundness of the type system. These features are necessary for the type system to be useful for program verification (see Section 2.2 and Section 2.4).

(2) Inheritance requirements
   (a) Inheritance mechanisms for both interface and implementation inheritance (Section 2.3).
   (b) Substitutability property (at least for interface inheritance) (Section 2.3).

(3) Expressibility requirements
   (a) Method types (Section 2.3).
   (b) Parametric types (Section 2.4).
   (c) Orthogonal type constructors (at least sets, tuples, and lists) (Section 2.4).
   (d) Encapsulation (Section 2.3 and Section 2.4).
   (e) The ability of the type system to deal with mutable object types and assignment (Section 2.4).
   (f) The ability of the type system to correctly type multi-methods (Section 2.3).
   (g) The ability of the type system to correctly type SQL-like queries (Section 2.4).

(4) Uniformity requirements
   (a) Extensibility (user-defined and system types should have the same status) (Section 2.5).
   (b) Types (classes) in the type system should not be specified as either persistent or transient (Section 2.4).
   (c) Method uniformity at the type level (no distinction between types of *stored* and *computed* methods) (Section 2.3).

(5) Reflexivity requirements
   (a) Type system reflexivity (Section 2.3).

(6) Dynamic requirements
   (a) Possibility of partial type specification and dynamic type checking (Section 2.4).
   (b) Provisions for schema evolution (Section 2.5).

(7) Other requirements
   (a) Transparency of the type system for a programmer (Section 2.2).
   (b) Incremental type checking (Section 2.4).
   (c) The ability to define views in a type-safe fashion (Section 2.5).

Some of the above requirements are complementary, while others are contradictory. Most notably, decidable verification conflicts with reflection (as shown, for example, in [Cardelli 1986]). Also, enforceability conflicts (to a degree) with partial

type specification. Another conflict is that the complexity of the type system that satisfies the expressibility requirements will most probably make the resulting type system much less transparent for a programmer than one would like it to be. [Connor et al. 1991] also identified a conflict between substitutability, mutable types, and static type safety. The presence of such contradictory requirements makes the task of designing a type system that satisfies them particularly difficult.

2.7 Test programs

The following is a set of test programs that a type system should be able to type correctly[4]. These tests are primarily designed to test type systems for their expressibility as this is the most difficult set of requirements to check; however, the last test is the test for reflexivity and uniformity. The example programs are written in an object-oriented pseudo-language and are used to illustrate the tests rather than to suggest specific language constructs.

These programs are designed to test known problem areas of object-oriented type systems. They are also used to verify the ability of the type system to consistently and orthogonally combine parametric and inclusion polymorphism with mutable types and assignment. This has to be done because soundness, verifiability, parametricity, substitutability, and mutable types are all among the requirements for an OODBPL type system.

Many of the expressibility tests are adapted from [Black and Palsberg 1994] which presented a benchmark for testing type system expressibility. However, their benchmark was designed to measure the expressibility of a type system for an object-oriented programming language and not for an object-oriented *database* programming language. Thus, some tests related to the additional expressibility requirements presented above were added.

The requirement related to mutable object types and assignment is probed by each of the tests below. This is done in order to verify that a type system can deal with mutable types in combination with parametric and inclusion polymorphisms — a well-known problem area in object-oriented type systems [Connor et al. 1991].

(1) Types `T_Person` and `T_Child` with method `getAge` that returns `T_Integer` when applied to a person and `T_SmallInteger` when applied to a child. (*PERSON*)

```
type T_Integer;
type T_SmallInteger subtype of T_Integer;

type T_Person {
      getAge(): T_Integer;
};
type T_Child subtype of T_Person {
      getAge(): T_SmallInteger;
};
```

[4]Note that the terms "test" and "test program" are not used here in the traditional software engineering sense. Our tests are actually benchmarks that validate expressive power.

```
T_Person p := new T_Person (...);
T_Child  c := new T_Child (...);
T_Integer i;
T_SmallInteger si;

i := p.getAge();       // should be Ok
i := c.getAge();       // should be Ok

si := p.getAge();      // should cause compile-time error
si := c.getAge();      // should be Ok
```

This test is designed to verify that subtyping does not necessitate the absence of changes. Surprisingly, there are a considerable number of languages that do not allow *any* changes while inheriting, only additions. This significantly limits the power of the type system and forces the designer to use less specific type information.

(2) Types `T_Point` and `T_ColorPoint`, with equality on both. The equality between color points should take color into account, while the equality between two points or between a point and a color point should ignore it. (*POINT*)

```
type T_Point {
    equal(T_Point p):T_Bool implementation  ... ;
                                                // equal1
};
type T_ColorPoint subtype of T_Point {
    equal(T_ColorPoint p):T_Bool implementation  ... ;
                                                // equal2
};

T_Point p1 := new T_Point (...);
T_ColorPoint p2 := new T_ColorPoint (...);

p1.equal(p1);                   // should call equal1
p1.equal(p2);                   // should call equal1
p2.equal(p2);                   // should call equal2
p2.equal(p1);                   // should call equal1

p1 := new T_ColorPoint (...);

p1.equal(p1);                   // should call equal2
p1.equal(p2);                   // should call equal2
p2.equal(p2);                   // should call equal2
p2.equal(p1);                   // should call equal2
```

This test is adapted from [Black and Palsberg 1994]. It tests the type system's ability to deal with *binary methods*, a well-known problem area in object-oriented type systems [Bruce et al. 1996; Fisher and Mitchell 1996].

(3) Types `T_Number` (with unrelated subtypes `T_Real` and `T_Radix`) and `T_Date` with comparison methods such that comparing two numbers or two dates is

legal, while their cross-comparison is not. All method code below, except for the code for the method `less`, should be reused. (*COMPARABLE*)

```
interface I_Comparable {
    less(selftype c):T_Bool;
    greater(selftype c):T_Bool
              implementation { return c.less(this); };
};
type T_Number implements I_Comparable {
    less(T_Number n):T_Bool implementation  ... ; // less1
};
type T_Real subtype of T_Number { ... };
type T_Radix subtype of T_Number { ... };
type T_Date implements I_Comparable {
    less(T_Date d):T_Bool   implementation  ... ; // less2
};

T_Date d1, d2;
T_Number n1, n2;
T_Radix radixVar := 0xF;
T_Real realVar := 1.0;

d1 := '2/3/99'; d2 := '3/4/78';

n1 := realVar;
n2 := radixVar;

n1.less(n2);            // should call less1
n2.greater(n1);         // should eventually call less1

n1.less(realVar);       // should call less1
n2.greater(radixVar);   // should eventually call less1

d1.less(d2);            // should call less2
d1.greater(d2);         // should eventually call less2

d1.less(n1);            // should cause compile-time error
n1.less(d1);            // should cause compile-time error
```

This is an additional test for binary method handling. It tests whether the subsumption property can be maintained in the presence of binary methods. This test is necessary due to the fact that some approaches to the binary method problem (most notably, matching [Bruce et al. 1995; Bruce et al. 1996]) abandon substitutability, which is one of the requirements for an OODBPL type system.

(4) A parametric input/output/IOstream type hierarchy such that the three types are parameterized by the type of objects readable/writable to/from a particular stream, with `T_IOstream` being a subtype of both input and output stream types. (*STREAMS*)

```
type T_InputStream(covar X) {
    get():X;
    isEmpty():T_Bool;
};
type T_OutputStream(contravar X) {
    put(X arg);
};
type T_IOStream(novar X)
        subtype of T_InputStream(X), T_OutputStream(X);

type T_Point { ... };
type T_ColorPoint subtype of T_Point { ... };

T_Point p := new T_Point (...);
T_ColorPoint cp := new T_ColorPoint (...);

T_InputStream(T_Point) isp;
T_OutputStream(T_Point) osp;
T_IOStream(T_Point) iosp;

T_InputStream(T_ColorPoint) iscp;
T_OutputStream(T_ColorPoint) oscp;
T_IOStream(T_ColorPoint) ioscp;

... // Initialization of the above streams

p := isp.get();           // should be Ok
p := iscp.get();          // should be Ok
cp := isp.get();          // should cause compile-time error
cp := iscp.get();         // should be Ok

osp.put(p);               // should be Ok
oscp.put(p);              // should cause compile-time error
osp.put(cp);              // should be Ok
oscp.put(cp);             // should be Ok

isp := iosp;              // should be Ok
isp := ioscp;             // should be Ok
iscp := iosp;             // should cause compile-time error
iscp := ioscp;            // should be Ok

osp := iosp;              // should be Ok
osp := ioscp;             // should cause compile-time error
oscp := iosp;             // should be Ok
oscp := ioscp;            // should be Ok
```

This test is designed to verify that a combination of parametric and inclusion

polymorphism in the type system does not adversely affect either of them. In other words, it tests the orthogonality of the two polymorphisms. The presence of both parametricity and inclusion polymorphism (subtyping + substitutability) in the type system is among the requirements compiled earlier. The unrestricted combination of the two polymorphisms is known to be difficult [Day et al. 1995; Bracha et al. 1998c].

(5) Sorting of arbitrary objects under the constraint that all the objects have a comparison method. (*SORT*)

```
interface I_Comparable {
    less(selftype c):T_Bool;
};
type T_Number implements I_Comparable { ... };
type T_Person { ... };
type T_List(novar X) { ... };

sort(T_List(X) list): T_List(X)
   where (X implements I_Comparable) implementation ... ;

T_List(T_Number) ln;
T_List(T_Person) lp;

... // Initialization of list variables

ln := sort(ln);         // should be Ok
lp := sort(lp);         // should cause compile-time error
lp := sort(ln);         // should cause compile-time error
ln := sort(lp);         // should cause compile-time error
```

This test is due to [Black and Palsberg 1994]. It is designed to verify the ability of the type system to deal with bounded quantification of the form "for all types satisfying a given condition". This is yet another aspect of the binary method problem. It also tests the ability of the type system to provide a link between parametricity and subtyping.

(6) Generic sort with a comparison method (<) as a parameter. The generic sort can be applied to a set of any objects provided that an appropriate comparison method is supplied. (*GENSORT*)

```
type T_List(X) { ... };
type T_Number {
    less(T_Number arg):T_Bool;
};
type T_Date {
    compare(T_Date arg):T_Bool;
};

sort(T_List(X) list, (X,X):T_Bool comparison): T_List(X)
                implementation ... ;
```

```
T_List(Number) ln;
T_List(Date) ld;

... // Initialization of list variables

ln := sort(ln,less);     // should be Ok
ld := sort(ld,compare);  // should be Ok
ln := sort(ld,compare);  // should cause compile-time error
ln := sort(ln,compare);  // should cause compile-time error
```

This test is also from [Black and Palsberg 1994]. It is designed to verify that the type system is capable of combining parametricity, method typing, and substitutability.

(7) A single-linked list node type and a double-linked list node type, where the second type inherits from the first one. A single-linked list node type is a recursively defined type with a mutable attribute that represents the list node linked to the given one. A double-linked list node type has an additional mutable attribute to represent the second link. The type system should not allow links between different node types.

Note that in this example the double-linked node type is *not* a subtype of a single-linked node type; however, the type system should be flexible enough to allow code reuse between them. The code sample uses the keyword `extends` in order to describe this relationship between types. (*LIST*)

```
type T_LinkedListNode {
    selftype next;
    getNext():selftype implementation { return next; };
    attach(T_LinkedListNode node)
             implementation  ... ;        // attach1
};
type T_DoubleLinkedListNode extends T_LinkedListNode {
    selftype prev;
    getPrev():selftype implementation { return prev; };
    attach(T_DoubleLinkedListNode node)
             implementation  ... ;  // attach2
};

T_LinkedListNode lln1, lln2;
T_DoubleLinkedListNode dlln1, dlln2;

... // Initialization of list node variables

lln1 := dlln1;           // should cause compile-time error
lln1 := lln2;            // should be Ok
lln1.attach(lln2);       // should call attach1
dlln1.attach(dlln2);     // should call attach2
lln1.attach(dlln1);      // should cause compile-time error
dlln1.attach(lln1);      // should cause compile-time error
```

This test is also adopted from [Black and Palsberg 1994]. It checks whether the type system supports code reuse beyond that provided by subtyping. In other words, it checks if code reuse is possible between types that are not in a subtyping relationship with each other. Situations analogous to the one described in this test occur frequently when dealing with mutable types.

(8) Set union and intersection for immutable sets. (*SET*)

```
type T_Set(X) {
    union(T_Set(Y) summand): T_Set(lub(X,Y));
    intersection(T_Set(Y) summand): T_Set(glb(X,Y));
};

type T_Person { ... };
type T_Student subtype of T_Person { ... };

T_Set(T_Person) sp1, sp2;
T_Set(T_Student) ss1, ss2;

... // Initialization of set variables

sp1 := sp1.union(sp2);   // should be Ok
sp1 := sp1.union(ss1);   // should be Ok
sp1 := ss1.union(sp1);   // should be Ok
sp1 := ss1.union(ss2);   // should be Ok

ss1 := sp1.union(sp2);   // should cause compile-time error
ss1 := sp1.union(ss1);   // should cause compile-time error
ss1 := ss1.union(sp1);   // should cause compile-time error
ss1 := ss1.union(ss2);   // should be Ok

sp1 := sp1.intersection(sp2); // should be Ok
sp1 := sp1.intersection(ss1); // should be Ok
sp1 := ss1.intersection(sp1); // should be Ok
sp1 := ss1.intersection(ss2); // should be Ok

ss1 := sp1.intersection(sp2);
           // should cause compile-time error
ss1 := sp1.intersection(ss1); // should be Ok
ss1 := ss1.intersection(sp1); // should be Ok
ss1 := ss1.intersection(ss2); // should be Ok
```

This test is designed to verify that set operations used in SQL-like declarative queries can be successfully and precisely typed. This is necessary to ensure seamless integration of such queries into the language.

(9) Function *apply*. (*APPLY*)

```
apply((X):Y msg, X obj):Y implementation ... ;

type T_Integer { ... };
```

```
type T_SmallInteger subtype of T_Integer { ... };

type T_Person {
      getAge():T_Integer;
};
type T_Child subtype of T_Person {
      getAge():T_SmallInteger;
};

T_Person p := new T_Person (...);
T_Child  c := new T_Child (...);
T_Integer i := 1000;
T_SmallInteger si := 5;

i := apply(getAge,p);    // should be Ok
i := apply(getAge,c);    // should be Ok

si := apply(getAge,p);   // should cause compile-time error
si := apply(getAge,c);   // should be Ok
```

This test is analogous to the "λ-calculus" test presented in [Black and Palsberg 1994] (it differs in that this test also includes assignment). It is designed to verify the ability of the type system to deal with method types and uniformly treat methods as objects in the system.

(10) General database browser. The browser should be able to deal with databases that have an unknown schema[5]. (*BROWSER*)

```
printNumber(T_Number num)  implementation ... ;
type T_Person {
      getAge():T_Integer;
};

T_Object root;
T_Database db;

db.open();

root := db.getRoot();
typecase root.typeOf() {
    subtype of T_Number: {
              printNumber(root); ...          };
                                          // should be Ok
    subtype of T_Person: {
              printNumber(root.getAge()); ... };
                                          // should be Ok
```

[5]The given code only tests the ability of the type system to deal with dynamic type information in a type-safe manner. A complete browser would also require the ability to examine the type structure of previously unknown types.

```
        otherwise: {
                    print("Something else"); };
    };

    root.getAge();
                // should cause compile-time error
    printNumber(root.getAge());
                // should cause compile-time error
```

This test checks the ability of a type system to handle partial type specifications and dynamic type checking. Both are on the list of requirements for an OODBPL type system.

This set of requirements and tests will be used in the next section to evaluate existing languages and type systems.

3. TYPE SYSTEMS REVIEW

In this section, type systems of many current languages[6] as well as theoretical developments in the area will be reviewed. These type systems and languages are listed in Table I. The table gives references and sections in this paper where a given system is reviewed.

3.1 C++

C++ [Stroustrup 1991] is currently one of the most widely used object-oriented programming languages. C++ types combine the characteristics of interface and implementation types in that they define both the interface and the structure of their objects. Classes in C++ are special cases of types: classes specify properties of first-class objects, while types specify properties of non-first-class objects. C++ inheritance rules are novariant; however, C++ allows polymorphic function and method specifications by using a method (function) signature instead of a name for identification purposes. In the presence of static type checking, this is equivalent to a restricted form of static multiple dispatch. Non-first-class objects in C++ are operated upon by free functions; only objects (instances of classes) have methods. C++ also provides parametric types (templates) that can take an arbitrary number of parameters; parametric types can be subtyped.

The C++ type system is not verifiable in general due mostly to its unrestricted use of pointer conversions; however, partial verification is possible and is performed at compilation time. The C++ type system combines interface and implementation inheritance and thus violates the first inheritance requirement. It is not completely uniform as it distinguishes between "data" and "objects" and treats attributes in a special way. C++ provides a limited substitutability for object pointers (not for objects). In terms of expressibility, the C++ type system is quite powerful as it has function types, parametric types, orthogonal type constructors, and deals with mutable object types. However, when used on the test suite, the C++ type system

[6]Due to the enormous number of languages constantly being developed by the scientific community, this review has to be incomplete. However, every effort has been made to include known languages with interesting type system features (or their analogs). Thus we hope that none of the essential type systems are left out.

Table I. Type system index

	References	Reviewed in Section
C++	[Stroustrup 1991]	3.1
E	[Richardson et al. 1993]	3.2
O++	[Agrawal and Gehani 1989]	3.3
O_2	[Lécluse et al. 1992]	3.4
Java 1.1	[Arnold and Gosling 1996], [Drossopoulou and Eisenbach 1997], [Saraswat 1997]	3.5
Generic Java (GJ)	[Bracha et al. 1998c; 1998a; 1998b]	3.5
Java parametric extension (JPE)	[Myers et al. 1997]	3.5
Pizza	[Odersky and Wadler 1997; Agesen et al. 1997]	3.5
Virtual types in Java (JVT)	[Thorup 1997; Bruce et al. 1998]	3.5
ODMG/OQL 2.0	[Cattell et al. 1997; Alagić 1997; 1999]	3.6
SQL-99	[SQL 1999]	3.7
XQuery 1.0	[W3C 2002]	3.8
Modula-2	[Wirth 1983]	3.9
DBPL	[Schmidt and Matthes 1994]	3.10
Modula-90	[Lutiy et al. 1994]	3.11
Modula-3	[Harbison 1992]	3.12
Oberon-2	[Mössenbök and Wirth 1993; Roe and Szyperski 1997]	3.13
Lagoona	[Franz 1997]	3.14
Theta	[Day et al. 1995]	3.15
Ada 95	[Kempe Software Capital Enterprises 1995]	3.16
Eiffel	[Meyer 1988]	3.17
Sather	[Stoutamire and Omohundro 1996]	3.17
Emerald	[Raj et al. 1991]	3.18
BETA	[Madsen et al. 1993]	3.19
VML	[Klas and Turau 1992]	3.20
Napier88	[Morrison et al. 1996]	3.21
Smalltalk	[Goldberg and Robson 1989]	3.22
Strongtalk	[Bracha and Griswold 1993]	3.22
Cecil	[Chambers 1993; 1992]	3.23
BeCecil	[Chambers and Leavens 1996; 1997]	3.23
Mini-Cecil	[Litvinov 1998]	3.23
Transframe	[Shang 1997]	3.24
CLOS	[Bobrow et al. 1988]	3.25
Dylan	[Apple Computer, Inc. 1994]	3.26
TM	[Bal et al. 1993]	3.27
ML	[Miller et al. 1990; Wright 1993]	3.28
Machiavelli	[Ohori et al. 1989; Buneman and Ohori 1996]	3.29
Fibonacci	[Albano et al. 1995]	3.30
$ML_{\leq}$	[Bourdoncle and Merz 1996a; 1996b]	3.31
Constrained types in ML	[Aiken et al. 1994], [Aiken and Wimmers 1993], [Pottier 1998], [Rehof 1998]	3.31
Constrained types in Erlang	[Marlow and Wadler 1997]	3.31
λ&-calculus	[Castagna et al. 1995]	3.31
PolyTOIL	[Bruce et al. 1995; 1994]	3.32
Loop	[Eifrig et al. 1995; Eifrig et al. 1995b; 1995a]	3.33
TL	[Matthes and Schmidt 1992; Matthes et al. 1994]	3.34
TooL	[Gawecki and Matthes 1996]	3.35
TOFL	[Qian and Krieg-Brueckner 1996]	3.36

fails all tests except for *GENSORT*. Other tests can be programmed, but only with significant type-checking lapses. The reason for this is the fact that C++ does not handle the typing of binary methods since it lacks multiple dispatch. Parametric types in C++ can only be used when fully instantiated, thus limiting a programmer's ability to define polymorphic functions (methods). Type parameters are always unrestricted and novariant. The variant of C++ that includes run-time type information (RTTI) allows for dynamic type checking (allowing it to tentatively pass the *BROWSER* test). Intersection and union types can not be represented by the C++ type system. The C++ type system is also quite complicated.

3.2 E

E [Richardson et al. 1993] is derived from C++ and borrows much of the type and class system from it. Differences between E and C++ lie in the specification of parametric types and in the fact that E is a persistent language. In E (as opposed to C++) parametric types (called *generic classes*) can be specified using a general type parameter restriction mechanism; however, subtyping for parametric classes can not be defined in E. This mechanism allows a programmer to specify restrictions on methods of a parameter type. For example, it is possible to require

a certain function (method) parameter type to have a `compare` method with a given specification, thus making the test *SORT* succeed. As a persistent language, E is not completely uniform as its persistence is type-dependent. E also fails the *BROWSER* test.

3.3 O++

O++ [Agrawal and Gehani 1989] is another persistent language derived from C++. It is similar to C++ in all respects except for provisions for persistence, queries, and a limited form of type-reflection. Persistence in O++ is not type-based, thus its type system is uniform in this respect[7]. Type reflection in O++ is provided by the means of the `is` operator that checks whether a given object has a given type. However, the *BROWSER* test in O++ still fails with respect to type checking even though it can potentially be programmed. The *SETS* test also fails as O++ queries do not provide the full power of SQL even though the typing for `forall` and `suchthat` is present in O++ at the operator (non-user-definable) level.

3.4 O_2

O_2 [Lécluse et al. 1992] provides a family of languages, but its type system is the same in all of them, so our discussion is based upon the CO_2 language, which is based on C. The type system of O_2 makes a distinction between first-class and non-first-class objects (called *values* in O_2; however, values in O_2 are mutable). O_2 uses the term *type* to refer to implementation types of non-first-class objects, and the term *class* to refer to types of first-class objects, which combine properties of interface and implementation types. Thus, the type system of O_2 is not completely uniform in that only first-class objects are manipulated by methods, much like in C++. Inheritance in O_2 is based on implementation subtyping, with an additional ability to add or modify methods. O_2 adopts a covariant signature refinement rule, thus providing for more natural data modeling. However, in the absence of multiple dispatch, this covariant rule results in the loss of static type safety, and thus the type system of O_2 is unsound. O_2 does not provide any kind of parametricity; methods, messages, and types are not objects in O_2. O_2 is uniform in terms of persistence (it is orthogonal to the type). O_2 also makes provisions for dynamic schema evolution; however, this evolution is not type-safe. O_2's type system essentially fails all tests for type system expressiveness, since even the tests that could be programmed would pass type checking and fail at run-time. In [Boyland and Castagna 1995], multimethods are used to provide type safety for covariant specifications in the O_2 programming language. With the addition of the mechanisms described in the article, test *POINT* would succeed, while the others would still fail.

3.5 Java

Java [Arnold and Gosling 1996] has recently become a popular language in both academic and industrial communities. Its type system shares many features with that of C++, therefore the discussion will focus primarily on its differences from the latter. The advantages of the Java type system include separation between

[7] O++ still does not have orthogonal persistence, as its persistence is declaration-based; also, persistent object creation is different from transient object creation.

interface and implementation, better handling of run-time type information, and simplification of the overall type system design resulting in a much more transparent type system. A non-reflexive Java fragment has been shown to be type-safe [Drossopoulou and Eisenbach 1997], while the full language has certain type deficiencies [Saraswat 1997]. On the other hand, the Java type system lacks method types (all methods when considered as objects have the same type in Java 1.1), parametric types (except for statically unsafe parametric arrays, which are built-in), and inherits some of the problematic features of the C++ type system discussed above. Java fails all tests except for *SORT*, the latter being successful due to the presence of *interfaces*. Java fails the *GENSORT* test as its method types are not sufficiently expressive for this test.

Recently, several parametric extensions to the Java type system have been proposed. Generic Java (GJ) [Bracha et al. 1998c; 1998a; 1998b] adds parametric types to the static type system, while using the same run-time model (type parameters are erased and do not exist at run-time). Parametric types in GJ are novariant and parameters can only be of reference (class) types. There are also several restrictions on the usage of parametric types and methods, related to the particular type inferencing algorithm used in GJ. GJ passes the test *COMPARABLE* and the union part of the *SET* test in addition to the *SORT* test passed by Java 1.1.

Another parametric type extension of Java is proposed in [Myers et al. 1997] (JPE). It allows usage of non-reference types as type parameters and also uses *where*-clauses to express requirements on parameter types. In this extension, parametric types are also novariant. The type-checking algorithm is not presented in [Myers et al. 1997]. This extension has test suite performance identical to that of Generic Java, but provides a more uniform system. These two approaches are informal in that they do not have a theoretical proof of their soundness.

Yet another Java extension is Pizza [Odersky and Wadler 1997] which extends Java with parametric and function types. The approach taken in [Odersky and Wadler 1997] is similar to that of [Myers et al. 1997], but is much better formalized. In fact, Pizza would have been statically type-safe if not for covariant arrays which were left in Pizza for Java compatibility. It is shown that the resulting type system does not have the subsumption property. However, the same is true of all Java extensions considered so far as well as of Java itself. Pizza passes tests *GENSORT* and *APPLY* in addition to *COMPARABLE* and *SORT* due to the presence of method types. A similar set of extensions is proposed in [Agesen et al. 1997], but without method types and a supporting theory. The latter approach, however, does lift several restrictions placed on the usage of type parameters in Pizza.

A different approach is taken by Thorup [Thorup 1997], where Java is extended with *virtual types* (JVT). Here the choice is made in favor of convenience, and both static typing and (dynamic) substitutability are sacrificed. Because of this, tests *PERSON*, *COMPARABLE*, and *LIST* could be programmed, but would give run-time rather than compile-time errors in incorrect cases. A modification of this approach is presented in [Bruce et al. 1998], which also compares parametric and virtual type approaches.

[Boyland and Castagna 1997] extends Java with multi-methods that are introduced via so-called *parasitic methods*. The goal of this extension is to add support

for multi-methods to the existing language providing full compatibility with Java. [Boyland and Castagna 1997] contains a proof of soundness for the resulting system.

3.6 ODMG Object Model

The Object Database Management Group has developed a set of standards for object database management systems [Cattell et al. 1997]. Two of these standards specify an object model (ODMG Object Model) and an object query language (OQL). The object model includes types that specify abstract properties and abstract behavior of their objects. Types are categorized as *interfaces* (abstract behavior only), *classes*, and *literals* (abstract state only). Types are implemented by language-specific *representations*; a single type can have several representations, but only one of them can be used in a single program. Interfaces support multiple inheritance, while classes only support single inheritance (*class extension*). Thus, the ODMG Object Model provides separation between interface and implementation. Interfaces in this model can not be instantiated. Abstract properties in the ODMG Object Model include abstract state and abstract relationships (two-way mappings). Relationships can only be defined between classes. Abstract behavior is specified as a set of *operations*. Behavior specifications are novariant in both their argument and return types. The ODMG Object Model supports single dispatch. Several system-defined parametric container classes are present in the object model, but the user is not allowed to define new ones. Type parameters can only be used in property specifications; operation specifications can not be parameterized. OQL is a strongly typed query language that provides a possibility of explicit dynamically checked type conversions. Set operations in OQL can only be performed on "compatible types" (types that have the least upper bound). Since the notion of the greatest lower bound is not available in OQL, all set operations use the least upper bound for typing purposes. For example, intersecting a set of students with a set of persons would return a result of type "set of persons" even in the case when the type of students is a subtype of the person type. ODMG/OQL fails all tests except for *BROWSER* and the union part of the *SET* test. It should be noted, however, that ODMG/OQL is not a general-purpose database programming language, and therefore its performance on the test suite is not fully indicative of the merits of the ODMG Object Model.

The ODMG object model has also been analyzed in [Alagić 1997] from the point of view of type checking. It has been shown that sound and verifiable typechecking of OQL queries is impossible to achieve. In [Alagić 1999] it has also been shown that OQL queries could be verifiably typechecked if both the ODMG type system and the ODMG language bindings supported parametric polymorphism and constrained types. This extension to the ODMG model does not change its performance on the test suite.

3.7 SQL-99

The SQL-99 language standard [SQL 1999] extends the industry standard database query language SQL with a significant number of new features, including computationally complete semi-procedural language and object-oriented extensions such as user-defined data types and dynamically dispatched methods.

The type system of SQL-99 consists of *predefined* (built-in), *constructed*, and

user-defined types. *Constructed types* include *array types*, *reference types*, and *row types*. *Constructed types* are parameterized: an array type specifies the type of its elements and maximum allowed cardinality, a reference type specifies its referenced type, and a row type specifies a set of type-value pairs (row types are primarily used for table specifications). The constructed types are special: the user is not allowed to change their behavior or define new constructors. Data (objects) of different types do not have the same status in the language.

In addition to types, there is also the notion of *domain*. *Domain* is essentially a data type plus a set of predicates restricting the possible values of that type. Domain usage, however, is restricted: for example, domain can not be used to specify the type of a procedural parameter.

User-defined types can be either *structured* or *distinct*. *Distinct types* are distinct synonyms of predefined types. User-defined subtyping can not be specified for *distinct types*. *Structured types* have single inheritance. There is an additional requirement that every subtype family must have exactly one common supertype. It is also possible to specify that a user-defined type is abtract (non-instantiable) or final (non-subtypable). SQL-99 has the subsumption property.

User-defined types can define methods. For structured types, methods can be novariantly overridden and dynamically dispatched according to the receiver type. It is also possible to define `CONSTRUCTOR` and `STATIC` methods. Each data type attribute automatically defines *accessor* and *mutator* methods; however, these methods can not be overridden.

In SQL-99, the user can specify functions and routines that are not methods (i.e., they are not attached to any type). These routines can be overloaded; the routine to execute is determined statically, according to the type specificity of all the parameters. It is required that such determination be unambiguous. Routine parameters can be specified as input, output, or input-output. A limited form of type parametricity is allowed: a function (or a method) can be *type-preserving*. In this case, one of the input parameters can be specified as a `RETURN` parameter that will give its type to the result of function application. However, the validity of this assertion is not statically guaranteed. Therefore, a run-time type check is performed when the value is returned to ensure that its run-time type is the same as (or a subtype of) the run-time type of the specified parameter.

A mechanism is provided for user-defined sorting and comparison that allows the user to define comparison procedures for all user-defined types. However, this mechanism is special, and can not be applied to binary methods other than comparison.

SQL-99 routines, types, methods, and functions can be persistent, either as parts of a schema or as parts of a *persistent module*.

SQL-99 has an extensive declarative query mechanism based on a significantly extended version of the standard SQL `SELECT` statement. The query mechanism is integrated with array indexing, reference processing, and path expressions that allow direct access to object attributes and methods. Procedural capabilities, including assignment, are also supported; however, there is no concept of instance-update methods. In other words, applying a "mutator" method always "creates" a new instance which must in turn be assigned to the old instance. All system-generated

mutator methods are therefore type-preserving.

SQL-99 is statically and strongly typed. It is also fully reflexive, allowing full dynamic schema modification and manipulation. Run-time type inspection is also supported.

Overall, the SQL-99 type system is verifiable and fully reflexive, while being non-uniform and lacking most of the advanced expressibility features. As a result, it fails all the tests except for the *BROWSER* test and the union part of the *SET* test. The *SORT* test can also be programmed, but only using the special mechanism designed specifically for sorting and comparison.

However, just as in the case of the ODMG object model, it should be noted that the main focus of SQL-99 is not a general-purpose object-oriented database programming language, and therefore its performance on the test suite is not fully indicative of the merits of the SQL-99 standard. For example, useful notions such as triggers, access rights, transaction atomicity, while being extremely important, are beyond the scope of this survey since they do not affect the type system.

3.8 XQuery

XQuery 1.0 [W3C 2002] is a query language for retrieving and interpreting information stored in XML format. An XQuery program is a set of XQuery functions; there is no notion of method dispatch, since data are considered to be passive elements of processing. In XQuery, data types are regular grammars. For example, data of the type denoted as $((A|B)*, C+, A*)$ consist of a sequence of one or more data items of type A or B, followed by one or more data items of type C, followed by zero or more data items of type A. Since XQuery is a purely functional language, it does not need typing rules for assignment or mutable data. XQuery has subsumption, and its (structural) subtyping is based on grammar inclusion (type t_1 is a subtype of t_2 if and only if the language generated by t_1 grammar is a subset of the language generated by t_2). XQuery supports static typechecking. The XQuery type system also supports statically typed `typecase` statement that allows type-safe queries over data with a structure that is not known at compile time. On the other hand, there is currently no support for overloaded functions, user-defined higher-order functions, or user-defined parametric types. Thus, this type system lacks most of advanced expressibility features.

At the time of writing, XQuery is still being developed, and many of its type system features have not been finalized. For example, it has not been decided whether user-defined subtyping will be added; whether user-defined polymorphic functions will be allowed; how metadata queries should be constructed and typed; or whether static typechecking will be mandatory or merely suggestive. Many theoretical aspects of the proposed type system also need further development, such as semantics and handling of intersection types (*interleaved types* in XQuery). This makes it impossible to reliably apply the tests to the XQuery type system at this time.

3.9 Modula-2

The following group of languages is based, directly or indirectly, on Modula-2 [Wirth 1983]. Modula-2 is neither an object-oriented nor a persistent language. Its type checking is verifiable. Modula-2 has a construct for separating interface and imple-

mentation in the form of *interface* and *implementation modules*, and the languages based on Modula-2 also use this approach. This mechanism has proven to be very convenient and robust for procedural languages. However, its advantages and disadvantages for object-oriented languages with their primarily type-based approach to both interface and implementation specification are yet to be evaluated.

3.10 DBPL

DBPL [Schmidt and Matthes 1994] is one of the persistent languages based on Modula-2. In DBPL, modularity is achieved by using the native language (Modula-2) modularization mechanisms with a special `DATABASE MODULE` construct. In the absence of module persistence, it does not cause problems with orthogonality. Transactions are supported as special procedures. Partial SQL compatibility is provided by the use of a `RELATION OF` type constructor with the appropriate set of operations and first-order constructs `ALL IN`, `SOME IN`, and `FOR EACH`. DBPL also allows updatable and non-updatable views (via `SELECTOR` and `CONSTRUCTOR` procedural specifications). DBPL is not object-oriented; however, it does offer implementation types (with no methods). The DBPL type system is static and non-reflexive. The tests described in Section 4 are not applicable to DBPL directly as it is not object-oriented. DBPL would tentatively pass only the test *GENSORT*. The DBPL type system is uniform in that system and user-defined types have the same rights.

3.11 Modula-90

Another persistent language based on Modula-2 is Modula-90 [Lutiy et al. 1994]. Modula-90 has some rudimentary object-oriented capabilities (single inheritance) and is similar to C++ in that method signatures are novariant, object types are different from other types, implementation and interface hierarchies coincide, and there is only a limited support for function types. Thus, the Modula-90 type system is not uniform. An interesting property of Modula-90 is the presence of type `DYNAMIC`. Data of this type are pairs of values and their types expressed as values of a compound type `DATATYPE`. Type `DATATYPE` is the type of the representations of all data types in the system; it can be used independently of the `DYNAMIC` type to store, retrieve, and operate upon various data types. Any value can be coerced to the type `DYNAMIC`. Modula-90 provides a special operator `TYPECASE` that provides a type-safe interface to the values of type `DYNAMIC`. The language also provides a set of predefined type operators that can be used to operate on values of type `DATATYPE`. Thus, Modula-90 provides type-safe immutable type reflection. The Modula-90 type system is static and non-parametric; however, it does provide orthogonal persistence. The only expressibility tests that succeed in Modula-90 are *GENSORT* and *BROWSER*. Modula-90 provides incremental type checking, including its dynamic variety.

3.12 Modula-3

Modula-3 [Harbison 1992] is another language with object-oriented extensions based on Modula-2. Modula-3 is not a persistent programming language and its object extensions are similar to those of Modula-90; it also has `TYPECASE` statement that gives a programmer the ability to request dynamic type checking in a type-safe

manner. Modula-3 has parameterized *modules*; however, parameterized *types* are not allowed. Modula-3 passes the tests *COMPARABLE*, *SORT*, *GENSORT*, and *BROWSER*.

3.13 Oberon-2

Another descendant of Modula-2 is Oberon-2 [Mössenbök and Wirth 1993]. Subtyping in Oberon-2 is based on *record extension*, also known as *structural subtyping*. The subtyping relationship is predefined for atomic types. Thus, Oberon-2 has multiple subtyping. Oberon-2 supports single dispatch. Subtyping of procedure (method) types is based on novariance of argument and result types; only the receiver type is covariant. Parametricity is not supported by Oberon-2. Separation between interface and implementation is supported at the level of *modules* in the same manner as in all Modula-2 based languages. Oberon-2 also has an extended `WITH` statement that allows a programmer to dynamically inspect the type of an object and act according to it. This statement is similar to the `TYPECASE` statement in Modula-3 and Modula-90 discussed earlier. The type system of Oberon-2 would pass the tests *SORT*, *GENSORT*, and *BROWSER*.

There has been a proposal for adding parametric types and methods to Oberon [Roe and Szyperski 1997]. In this proposal, parametric types are novariant, and the type checker ensures that a parameter type substitution exists that satisfies the rules for matching arguments of a particular call. Unfortunately, neither the typechecking algorithm nor the proof of its soundness are present in [Roe and Szyperski 1997].

3.14 Lagoona

Lagoona [Franz 1997] is another descendant of Oberon. It focuses on *messages*, *objects*, and message passing. In contrast to most object-oriented languages, a message in Lagoona is an independent entity. An interface (*category* in Lagoona) is a set of messages. Each message has a type specification. There can be several *methods* implementing a message, each for a different receiver type. Implementation in Lagoona is specified by a *class* (usually a record type); when a *category* meets a *class*, a *type* (combination of interface and implementation) is born. No code inheritance is possible in Lagoona, only structure inheritance is possible. Variables in Lagoona can be specified using either *category* or *class*. Lagoona thus provides a complete separation between interface, implementation (representation), and code (methods). When a message is sent to an object, an object can forward (`resend`) it to another object. Other features of the Lagoona type system include single dispatch and single inheritance. Unfortunately, the article [Franz 1997] provides insufficient information to test the performance of Lagoona on the test suite.

3.15 Theta

The language Theta [Day et al. 1995] combines the expressive power of parametric and inclusion polymorphisms. It also provides *where*-clauses that give its type system a flexibility similar to that provided by matching. Theta has single dispatch, multiple inheritance for types (interfaces), and single inheritance for classes (implementations). Theta only allows novariant parametric types. Method types can be specified in Theta and methods (functions) can be used as first-class values. Theta

also has reflexive capabilities in the form of the `typecase` operator. Theta's type system is verifiable, uniform, and static. It is also quite expressive. However, its soundness has not been proven. Theta passes the tests *COMPARABLE*, *SORT*, *GENSORT*, *LIST*, and *BROWSER*.

3.16 Ada 95

The language Ada 95 [Kempe Software Capital Enterprises 1995] is a procedural language with some object-oriented features. Ada 95 has a remarkably strong support for parametricity: *generic packages* (parametric modules) and *generic subprograms* (parametric functions) can not only be instantiated, but also used as parameters of other generic entities, thus providing for great flexibility. Parameter types can be constrained by a `with` clause requiring the type to have a method with a particular signature, which can also be parameterized. However, Ada generics have to be fully and explicitly instantiated before they can be used. Another interesting and powerful feature of Ada is the notion of an *access type* which generalizes such notions as "pointer" and "reference", allowing also for user-defined access types. Ada also has function types, and function arguments can be specified as `in`, `out`, or `inout` according to the role they play. Ada uses its *package* mechanism (similar to the module mechanism of Modula-2) to provide separation between interface and implementation. On the other hand, object-oriented Ada 95 features appear to be relatively weak: in order to be used for dynamic dispatch, a type has to be explicitly declared as `tagged`. In order to be able to use subtyping (inclusion) polymorphism, a programmer has to use a special form of parameter specification. Ada provides single dispatch and single inheritance with novariant methods. Subclassing in Ada is limited to record extension. In Ada, types can be examined dynamically; data of a type can be converted to any other type, and such a conversion is dynamically type-checked. The Ada type system is very complicated and it places emphasis on *static* rather than *dynamic* polymorphism. Ada will pass the tests *COMPARABLE*, *SORT*, *GENSORT*, *LIST*, and *BROWSER*, provided generic packages rather than types are used.

3.17 Sather

Next, the object-oriented language Sather [Stoutamire and Omohundro 1996] will be considered. Sather is based on the much better known Eiffel [Meyer 1988]; however, Eiffel is not discussed here as one of the primary goals behind the design of Sather was to remedy typing problems present in Eiffel.

Sather has multiple interface and implementation inheritance hierarchies almost independent of each other (concrete implementation types called *classes* in Sather must have leaf interface types). Implementation subtyping in Sather corresponds to textual inclusion with replacements. Sather uses single dispatch and is strongly and statically typed. Therefore its methods are covariant on the receiver and contravariant on other arguments. Sather also provides partial closures of its methods and iterators as first-class values. It has parametric implementation types that can use a form of bounded polymorphism to constrain type parameters. These parametric types are novariant and they can not be used until fully instantiated. Method arguments in Sather can be specified as `in`, `out`, or `inout`, according to the role they play. The Sather type system correctly handles all these cases. Argument types

and local variables can be specified by using either Sather types or Sather classes. In the former case, the parameter class has to be a subtype of the given type; in the latter case, the parameter class has to match the specification exactly. The language also provides a novel notion of *iters* (iterators) that are an object-oriented generalization of loop control structures. Sather also provides a special compound type `TYPE` and operators `typeof` and `typecase`. Thus, it has type-safe immutable type reflection. The type system of Sather therefore possesses verifiability (even though it has not been formally proven) and satisfies inheritance and (partially) uniformity and expressibility requirements. It passes tests *PERSON*, *COMPARABLE*, *SORT*, *GENSORT*, *LIST*, and *BROWSER* and fails on tests *POINT*, *STREAMS*, *SET*, and *APPLY*. Note that *APPLY* fails because parametric types in Sather require full instantiation before they can be used.

3.18 Emerald

Emerald [Raj et al. 1991] is a non-traditional object-oriented language that combines features of class-based and delegation-based object-oriented languages. Objects are created in Emerald by a special syntactic form called *an object constructor*. The object constructor plays a triple role: first, it defines the object's implementation; second, it defines a publicly visible object interface (*type* in Emerald); third, it denotes the process of object creation itself. Subtyping in Emerald is structural, as a type is understood as a set of signatures. Types are also objects that can be created by object constructors; thus, a user can define new types, including parametric ones. The type checker uses user-defined types along with system-supplied ones to typecheck a program statically. Since types are objects, dynamic type checking is also possible. The Emerald type system is therefore verifiable, uniform, reflexive, satisfies inheritance requirements, and is quite expressive. However, its soundness is unknown. Emerald passes the tests *PERSON*, *STREAMS*, *SORT*, and *BROWSER*.

3.19 BETA

BETA [Madsen et al. 1993] is a unique object-oriented language that unifies the notions of class, object, and procedure (method) via its notion of a *pattern*. Patterns can contain other patterns (such as member objects, code fragments, and types). BETA also has *fragments* that play the role of modules and can also be used to separate interface from implementation. BETA fragments can be regarded as high-level restricted patterns since they operate on the same basic principles. Patterns can be *virtual*; a virtual pattern can be extended by adding code fragments in places specified by the `inner` placeholder (a.k.a. method extension as in Simula-67 [Birtwistle et al. 1979]), by extending a virtual pattern with additional members (a.k.a. record extension), or by supplying a class pattern in place of a virtual one (a.k.a. parametric instantiation). Virtual patterns have to be explicitly declared as such. Unification of all language concepts using the notion of pattern makes it possible to design a very powerful language based on only a couple of orthogonal principles. While the language design simplicity is very impressive, the resulting language is quite unconventional. Structural subtyping in conjunction with a class substitution mechanism makes the type system of BETA statically unsound; dynamic type checks are inserted to ensure type safety. Due to the uniformity of BETA, all tests except for *POINT* can be coded in it, but only *SORT*

and *GENSORT* have static type safety. Illegal usage in other tests would cause run-time type errors.

3.20 VML

In the persistent object-oriented language VML [Klas and Turau 1992], all objects that are instances of object types (called *classes*) are persistent while all (non-first-class) objects of non-object types (called *data types*) are transient. However, a value of a non-object type can become persistent if it is referenced from a persistent object. Thus, VML does not have orthogonal persistence. VML object types are first-class objects and as such belong to their respective *metaclasses*. Metaclasses define some of the essential class methods, for example the methods for the *inheritanceBehavior* message that is used when a method for a message is not found in the receiver class. Thus, VML allows user-defined method inheritance due to its "classes are objects" concept. However, VML data types are not first-class objects and thus VML only partially satisfies the type reflexivity requirement. Inheritance in VML can be tailored to specific application needs by using the user-definable *inheritanceBehavior* message found in an appropriate metaclass. VML does not have a verifiable type system, therefore the tests are not applicable to it.

3.21 Napier88

The persistent programming language Napier88 (version 2) [Morrison et al. 1996] is not an object-oriented programming language; however, its type system is quite powerful. In Napier88, parametric types can be built freely from the basic types, type constructors, and type variables. Parametric procedures (procedures with parametric types) can also be defined and are first-class objects. Napier88 types are purely implementation types; however, Napier88 provides the type constructor `abstract` that may be considered as an interface type as it provides existential quantification over witness type(s). Thus, the type system of Napier88 satisfies all expressibility requirements not related to the notions of subtyping and inheritance. However, since Napier88 does not have the notions of subtyping and inheritance, it fails the requirements related to those notions. In Napier88, parametric types can not be used until fully instantiated. Napier88 provides support for a limited form of linguistic reflection via dynamic *environments*. Environments are dynamically typed structures that provide bindings of names to Napier88 entities. Environments can be dynamically modified, inspected and used in expressions. Environments can be persistent.

Napier88 has a special type `any` that is a union type of all types in the system. A special `project` operator can be used to deal with values of type `any` in a type-safe manner. This operator is similar to `TYPECASE` of Modula-90. Since any value can be injected into type `any`, the above mechanism makes run-time data type inspection possible. Thus, the type system of Napier88 has type-safe type reflexive capabilities.

The persistent store of Napier88 is an object of type `any` that holds a typed collection of objects. The objects from the persistent store can be projected from `any` and operated upon transparently after that. Napier88 uses a persistence model based on reachability from the root (persistent store) object. Thus, the persistence in Napier88 is orthogonal to type.

In spite of the power of its type system, Napier88 only unconditionally passes the test *LIST*. However, it tentatively passes the tests *COMPARABLE*, *SORT*, *GENSORT*, and *APPLY* if usage of explicit type parameters in parametric calls is allowed. In other words, the burden of inferring correct parametric instantiation from the code in Napier88 is placed on a programmer rather than on the type checker.

3.22 Strongtalk

Strongtalk [Bracha and Griswold 1993] is a statically typed version of Smalltalk [Goldberg and Robson 1989]. Smalltalk is widely regarded as the first purely object-oriented language; however, it has no static type checking.

In Strongtalk, everything (including types, called *classes* in Strongtalk, methods, and messages) is a first-class object. Thus, Strongtalk is uniform[8]. All Strongtalk objects can be operated upon and modified at run-time and therefore Strongtalk is reflexive (the reflexivity is type-unsafe). Strongtalk separates interfaces (*protocols*) from implementations (*classes*), provides subtyping and matching. It also has parametric (novariant) types and messages, block types, and union types. However, the user has to explicitly specify a mechanism to guide parametric type inferencing, resulting in the loss of substitutability. It is also unclear from [Bracha and Griswold 1993] whether Strongtalk allows bounded parametric types. Strongtalk provides single inheritance and single dispatch. Subtyping is structural, but the user can use *brands* to restrict it. Even though Strongtalk was not designed to be a persistent language, it does provide uniform persistence of its objects in a so-called *image*. Overall, the Strongtalk type system is quite expressive, uniform, and reflective. It passes the tests *COMPARABLE*, *LIST*, *BROWSER*, and, possibly, *GENSORT* (if parametric type bounds are allowed). The union part of the *SET* test is also passed. Strongtalk would fail the tests *PERSON*, *POINT*, *STREAMS*, as well as *APPLY* and *SORT* (the latter two could be programmed, but only with blocks rather than messages).

3.23 Cecil

Cecil [Chambers 1993; 1992] is a delegation-based language that has both implementation and interface types (the former are called *representations* while the latter are called *types*). Types in Cecil are used for suggestive type-checking only as Cecil's multiple dispatch is done according to representations. Cecil's type checking is suggestive because it might report false errors or miss real ones, therefore its type discipline is not enforceable. Cecil's does not support incremental type checking. Cecil uses multiple dispatch and provides covariant specification of specialized arguments together with contravariant specification of unspecialized ones. Closures and methods are first-class objects in Cecil; they are contravariantly typed. Cecil also provides novariant parametric types and methods as well as type parameter bounds. Parametric types in Cecil can be instantiated either explicitly or implicitly; in the latter case, the user has to provide a hint to the type inferencing algorithm. This behavior results in loss of substitutability. Cecil has multiple inheritance, union and intersection types. Overall, the Cecil type system is quite expressive

[8] Except for a possible uniformity breach in the form of direct attribute access.

and uniform, while being non-reflexive[9] and static. It also satisfies the inheritance requirements. Since Cecil type checking is only suggestive, it is difficult to apply the tests to it. However, tests that Cecil would tentatively pass include *PERSON*, *POINT*, *SORT*, *GENSORT*, *LIST*, *SET*, *APPLY*, and *BROWSER*. It would fail the tests *COMPARABLE* (due to the restrictions on parametric type bounds) and *STREAMS* (due to the novariance of parametric types).

There are several extensions/modifications to the original Cecil language. One of these extensions is BeCecil [Chambers and Leavens 1996; 1997]. BeCecil is a statically typechecked version of Cecil. It supports block and modular structure, has extensible objects and an extensible type system, and its type system has been formalized. However, soundness and substitutability properties have yet to be proven. BeCecil also has a novel notion of *acceptors* which can be considered as an object-oriented generalization of the assignment operator. However, BeCecil does not have parametric types while sharing most of the other features with Cecil. The absence of parametricity contributes to the decreased expressibility of the BeCecil type system as compared to that of Cecil. BeCecil passes the tests *PERSON*, *POINT*, *SORT*, *GENSORT*, and *LIST*, and fails the rest.

Another modification (Mini-Cecil) is described in [Litvinov 1998]. Mini-Cecil strives to achieve a combination of static typing and a very general form of parametric polymorphism. Mini-Cecil also has *frameworks* which are basically interfaces with what appears to be an analog of `selftype`. The frameworks can be used to separate interface and implementation, as well as to achieve the effect of matching. Methods in Mini-Cecil are first-class objects; multimethods and multiple inheritance are supported. Subtype clauses can have a form of `forall` $\overline{\alpha}$ C_1 `isa` C_2`:` C_3 `isa` C_4, where $\overline{\alpha}$ is a set of free type variables and C_i are type specifications. The resulting type system is very expressive; it can even be argued that it is *the most* expressive type system possible. However, the typechecking seems to be undecidable. The algorithm proposed is a conservative approximation, and its soundness is yet to be proven. It is also unknown if the resulting type system has substitutability. Mini-Cecil would pass all tests except for *BROWSER* as it does not have reflexive capabilities.

3.24 Transframe

Transframe [Shang 1997] allows the user to specify whether a parameter of a parametric type is to be covariant or novariant (*type-exact*) and to constrain it by giving it an upper bound. Subtyping and subclassing (interface and implementation inheritance) are different concepts in Transframe. There is a distinct name `selfclass` that allows classes to support matching. The language unifies the notions of *class* and *function* (like BETA). Transframe also supports multiple dispatch. There are provisions for dynamic type checking and dynamic schema evolution. Transframe implicitly instantiates parameter types in expressions; unfortunately, there is no formal proof of type safety. In fact, it can be shown that the type system presented in [Shang 1997] is *not* type-safe. Overall, the Transframe type system is

[9]Being a delegation-based language, Cecil has language reflection; however, the type system of Cecil is not reflexive as Cecil types are not objects and can not be manipulated by the language. More precisely, *representations* of Cecil are reflexive while *types* are not.

verifiable, very expressive, almost uniform, and dynamically reflective. However, it is unsound. Transframe would pass all expressibility tests except for the test *STREAMS* (due to its inability to represent contravariant type parameters) and the intersection part of the test *SET* (due to the absence of intersection types).

3.25 CLOS

CLOS (Common LISP Object System) [Bobrow et al. 1988] is a reflexive language, with all the power of Common LISP reflection. CLOS has types and object types (called *classes*), the latter being a subset of the former. CLOS types are implementation types; they do not specify any interface. However, CLOS classes combine implementation and interface definitions. Since CLOS makes a distinction between object and non-object types (where only object types define interfaces and are subject to inheritance), the CLOS type system is not completely uniform. CLOS classes are objects that belong to metaclasses, which are also objects; CLOS messages, methods, and functions are also CLOS objects that can be operated upon and changed at run-time, so CLOS is fully reflexive and dynamic. Subclassing in CLOS is slot collection extension with slot types changed covariantly (a type of a slot is the intersection of the types specified for this slot in all of the class' superclasses). Messages (called *generic functions*) are also covariant. CLOS is not statically typed. A CLOS message can be dispatched to yield an appropriate method (or a combination of methods) according to the class or value of all arguments (multiple dispatch). Methods are covariant on all arguments[10] since CLOS has multiple dispatch. If more than one method is appropriate for the given arguments, a user-definable way of constructing the function to be executed out of *all* appropriate methods is employed. In the body of a method, a special function `call-next-method` can be called to invoke the next applicable method. This capability is analogous to the `inner` construct of Simula-67 [Birtwistle et al. 1979] and is much more powerful. Overall, the dispatch mechanism of CLOS is the most powerful of all known mechanisms, if the consideration is limited to classes and values. CLOS can not dispatch on types. Updates in CLOS are invoked on slots directly or by using an appropriate message. CLOS is not statically type checked, therefore a run-time error is signalled if a value assigned to a slot does not conform to the slot's type specification. Since CLOS is not statically typed, the tests are in general not applicable to it; however, CLOS would pass *POINT*, *APPLY*, and *BROWSER* tests if its type discipline were enforceable.

3.26 Dylan

Dylan [Apple Computer, Inc. 1994] is an imperative programming language similar to CLOS. While there are certain differences between the two, they are almost identical in terms of their type systems. Dylan has more control over the defined classes, as Dylan classes can be sealed (only subclassable by the library where they belong) or open, primary (there is only single inheritance of primary classes) or free, abstract (all superclasses of an abstract class must be abstract) or concrete. There is also support for singleton types, but not singleton classes. Multiple dispatch in

[10] More precisely, methods are covariant on those arguments that are constrained by class specifications.

Dylan is also different from that in CLOS in that in Dylan all arguments are equal, and the method specificity is defined by a class precedence list. Dylan also has modules with import and export lists and module libraries.

3.27 TM

TM [Bal et al. 1993] is an object-oriented persistent language with many functional features. TM has a verifiable type system based on [Cardelli 1988]. However, the soundness proof seems to be missing. TM's type hierarchy includes user-definable sorts (atomic, immutable types) and classes. Sorts and classes have representation (implementation) types that are almost hidden inside of them. Methods and types are not first-class values in TM. TM method specification uses `selftype` to achieve the effect of matching. Since this is the only polymorphic mechanism in TM, specifications are covariant and substitutability does not hold, as functional updates are present. No method redefinition mechanisms are provided in TM. The TM type system extends the type system of Cardelli [Cardelli 1988] with a powerset type constructor. Since TM is stateless, powerset types are covariant. TM allows enumerated as well as predicative sets as primitive language expressions. Enumerated sets in TM can only be homogeneous, while predicative sets can be heterogeneous up to subtyping. Predicative sets in the presence of the record-based type system of [Cardelli 1988] and the absence of updates play a role of embedded queries that are highly integrated with the rest of the language. TM provides several levels of constraint specification mechanisms which are also set-based and resemble relational constraint systems. TM also provides first-order set operations. However, the set operations require special treatment and are not messages in the usual object-oriented sense. It is also unclear how well different type constraints for different kinds of sets interact with each other. TM has modules that define their persistent components by names, and everything those objects refer to is also implicitly persistent. Thus, TM provides a combination of static name-based and dynamic reachability-based persistence, completely orthogonal to the type. Overall, the TM type system is verifiable, sound, supports both interface and implementation inheritance (though they are not completely independent of each other), is almost uniform, partially expressive, and provides support for declarative queries. However, it does not support substitutability and is non-reflexive and static. It would unconditionally pass tests *SORT* and *LIST*. Test *SET* also succeeds because of built-in support for set operations. However, it would not be possible to construct user-defined types with the same functionality.

Most of the following languages borrow much of their expressiveness from ML [Miller et al. 1990]. ML is a language with both functional and imperative flavors; it also has some object-oriented features. This can be said about almost all the languages discussed below.

3.28 ML

Standard ML [Miller et al. 1990] is a functional language with some imperative features. It is strongly typed and provides provably decidable and sound type checking. In the ML type system, all type information is *inferred* by the type checker. Addition of explicit type annotations and declarations is considered in [Odersky and Läufer 1996]. Standard ML also provides a very sophisticated module system,

where each module (*structure*) has its type (*signature*). However, ML's structures are more like abstract (implementation) types than modules, as they are designed to shield their internals from the rest of the program and not to handle separate compilation or similar tasks. There is no notion of subtyping in Standard ML, except for signature matching, which can be considered as restricted structural subtyping for abstract (interface) types. Highly parametric types are supported in Standard ML. They can arbitrarily include type variables and can be user-defined. Thus, the type system of Standard ML is uniform. For interface types (signatures) there are also functions that map signatures to signatures (functors). Function types in ML can also be polymorphic. Polymorphism in Standard ML function types is expressed via type expressions that have to be "pattern-matched". For example, the type of the identity function in Standard ML is `'a` $\rightarrow$ `'a`, where `'a` is a type variable. This kind of polymorphism is uniform in that it allows user-defined parametric types. Standard ML, being in essence functional, allows updates on so-called `ref` types. These types are reference types somewhat similar to pointer types in C or C++. `ref` types have a restricted parametricity, as an argument of a `ref` type must always be a monotype (e.g. see discussion in [Wright 1993]). Types are not objects in Standard ML even though they can be operated upon by functions similar to those that operate on ordinary values. Overall, the type system of Standard ML is theoretically sound, very expressive and uniform, while lacking inheritance and being only partially reflexive. This type system would tentatively[11] pass tests *PERSON*, *SORT*, *GENSORT*, and *APPLY* (for the test *SORT* use of modules rather than types is required).

3.29 Machiavelli

Another language from this family is Machiavelli [Ohori et al. 1989; Buneman and Ohori 1996] which is a persistent language that extends ML by adding more polymorphism as well as query and view support. Machiavelli has a verifiable and provably sound type system. It adds record inclusion polymorphism to ML by using type variables of the form `(a'')`, that correspond to an arbitrary record extension. Thus Machiavelli allows "more polymorphic" types than ML. Machiavelli is also able to automatically maintain more sophisticated (even non-covariant) type constraints on *description types*[12]. It is therefore possible for Machiavelli's type checker to statically infer an error in case a join of two sets of records whose types do not have a greatest lower bound is attempted. Machiavelli's type system treats description and other types differently. Thus, it is not completely uniform. In Machiavelli, a special set type constructor `{}` is introduced and query operations are defined for objects of set types. Machiavelli extends the type system of ML to be able to deal consistently with type inference of generalized relational operations. Machiavelli also provides views similar to those in relational databases. Namely, a Machiavelli view is defined as a function that returns a projection of a given set over some appropriate type. Machiavelli views are not updatable. Overall, the type system of Machiavelli is verifiable, sound, very expressive, reflexive, partially uniform, partially dynamic, and capable of supporting views. However, it lacks

[11] Provided that the notion of subtyping is substituted by the notion of code reuse.
[12] Description types are ML types that do not include $\rightarrow$ outside of `ref`.

interface inheritance. It passes the same tests as ML with the addition of the *SET* test due to the special support of sets by the language and the type system.

3.30 Fibonacci

Fibonacci [Albano et al. 1995] is a persistent object-oriented language that is a descendant of Galileo [Albano et al. 1985] which is, in turn, based on ML. It has some functional and imperative features and possesses a verifiable, provably sound type system; it also has multiple inheritance and single dispatch. In Fibonacci, *object types* are independent from each other in terms of subtyping; however, *role types* form independent directed acyclic graph (DAG) subhierarchies for each *object type*. For example, an object of type `PersonObject` can play roles `Person`, `Employee`, `Student`, and `TeachingAssistant` (that is both `Employee` and `Student`). Fibonacci roles can be dynamically created and Fibonacci objects can dynamically acquire new roles. Fibonacci types are not objects in the language. Method arguments in Fibonacci are contravariant, while their results are covariant. Fibonacci supports a distinction between methods and functions: methods are attached to role types and are not Fibonacci objects, while functions are independent and are first-class values. Fibonacci also has non-object and non-role types, such as basic types, class types, function types, and association types. These types form a hierarchy independent of that of object and role types. Fibonacci defines some built-in parametric types (`Class`, `Sequence`, and `Association`). However, it is unclear from [Albano et al. 1995] if the user can create new parametric types. In Fibonacci, objects of the same object type can have different implementations. These implementations are defined at the time of object creation. Thus, an implementation in Fibonacci is not a part of an object or role type specification. Updates in Fibonacci are allowed on special novariant `Var` types. In Fibonacci, the syntax of message sends determines the strategy of the method lookup. There are two strategies: *upward lookup*, that corresponds to the standard lookup procedure in the presence of multiple inheritance, and *double lookup*, that first tries to find an appropriate method in the subroles of the role it has started from. Fibonacci offers declarative query operators on parametric `Sequence` types. These are types of immutable sequences that are supertypes of their respective mutable `Class` and `Association` types. Thus, the query facilities of Fibonacci are also applicable to Fibonacci classes and associations. Fibonacci has a reachability-based orthogonal persistence model. Everything accessible from the top-level environment automatically persists between sessions. Thus, Fibonacci persistence is orthogonal to both interface and implementation types. Overall, the Fibonacci type system is verifiable, provably sound, provides different inheritance for interface and implementation types, and has inclusion polymorphism (substitutability). It is also expressive, almost uniform, and is capable of supporting query typing. However, it is non-reflexive and static. The type system of Fibonacci would pass the same tests as that of Machiavelli.

3.31 $ML_{\leq}$ and other subtyping mechanisms for ML

The language $ML_{\leq}$ [Bourdoncle and Merz 1996a; 1996b] is an extension of ML with subtyping and higher-order polymorphic multi-methods. It has type inference, strong static type checking, and substitutability. ML-like *type constructors* provide parametric polymorphism. Type constructors in $ML_{\leq}$ can be specified as

covariant, novariant, or contravariant. The formalism used is based on systems of type constraints. In this theoretical language, no separation between interface and implementation is provided. Handling of imperative (mutable) types is borrowed from ML and is quite restrictive w.r.t. polymorphism. Another restriction placed on $ML_{\leq}$'s type system is the requirement that all types that have a subtyping relationship should have the same number of arguments as well as the same variance specification for them. Thus, this type system fails the test *STREAMS* as it requires a subtyping relationship to be established between parametric types of different variances. Overall, the $ML_{\leq}$ type system is expressive, verifiable, sound, and static. It passes the tests *PERSON*, *POINT*, *SORT*, *GENSORT*, *APPLY*, and the union part of the *SET* test. However, the inability of the system presented in [Bourdoncle and Merz 1996a] to deal with recursive constraints makes generalization of the system to unrestricted subtyping of parametric types quite difficult.

There are several other approaches to adding subtyping to ML, but none of them deals with multi-methods. Aiken and Wimmers [Aiken et al. 1994; Aiken and Wimmers 1993] proposed a system that finds a solution for a system of subtyping constraints; this system can deal with recursive constraints. Pottier [Pottier 1998] also proposed a system in which recursive constraints are allowed; instead of solving the constraints, his system proves their consistency (like that of $ML_{\leq}$). [Sequeira 1998] also adds subtyping and user-defined type constructors, as well as constrained types, to an ML-style type system. The resulting system appears to be similar to that of $ML_{\leq}$, but lacks its ability to deal with multi-methods. Complexity results related to solving subtyping systems appear in [Rehof 1998].

[Marlow and Wadler 1997] proposes a type system for a core subset of the purely functional language Erlang [Armstrong et al. 1996]. The type system is similar to the one proposed by Aiken and Wimmers [Aiken et al. 1994; Aiken and Wimmers 1993] and uses constrained type entailment for type verification. The main difference is the absence of function types, general unions, and intersections. The system is provably sound and presumably complete. Addition of function types to the system makes it incomplete, and the proof of soundness in this case is absent. [Marlow and Wadler 1997] includes decidable algorithms for type inferencing, signature verification, and constraint simplification.

Castagna, Ghelli, and Longo [Castagna et al. 1995] proposed an extension of λ-calculus ($\lambda\&$-calculus) dealing specifically with multi-methods. Several important results (generalized subject reduction, Church-Rosser etc) are proven. It is also shown how the calculus can be used to model inheritance, matching, and multiple dispatch.

[Chen and Odersky 1994] presents a type system where a full-fledged support for mutable types is added to an ML-style type system. The approach taken by the authors separates mutable and immutable types by creating a parallel language syntax. In essence, every functional language construct has its imperative counterpart. Interaction between mutable (imperative) and immutable (functional) language components is restricted by a set of type validity rules that restrict polymorphism of data types for the data passed between the two language components. The approach of [Chen and Odersky 1994] appears to be sound, but soundness seems to be achieved at the expense of language simplicity and type system trans-

parency.

3.32 PolyTOIL

PolyTOIL [Bruce et al. 1995; 1994] has a verifiable and sound type system and single subtyping. PolyTOIL identifies subtyping with substitutability, while providing a concept of *matching* (subtyping up to `MyType`). The latter is introduced to allow for covariant method specification. Subtyping in PolyTOIL is structural, as is matching. The language allows for both subtyping and matching constraints. Matching is used as a mechanism of specifying constraints that are weaker than substitutability and is therefore useful for updatable types. In [Bruce 1996] it is suggested that subtyping should be dropped altogether as matching is more intuitive and more flexible. In [Bruce et al. 1995], there are distinct notions of *object types* and *class types*. The former are interface types, while the latter are implementation types. Namely, object types specify signatures (type information) for methods applicable to the objects of this object type, while class types specify instance variables and code for methods applicable to objects that belong to the class. Classes are used to create new objects. They are produced by applying functions whose arguments are values used to initialize the produced objects as well as the argument types for parametric classes to their arguments. Classes can use inheritance with redefinitions. Parametric types in PolyTOIL are pattern functions of their parameter types. A notion of a function type is also present in PolyTOIL. However, it is only used in specifications and during type-checking and is inherently different from either class or object type. Overall, the type system of PolyTOIL is verifiable, sound, expressive, almost uniform, and satisfies inheritance requirements. However, it is static and non-reflexive. It passes tests *COMPARABLE*, *SORT*, *GENSORT*, *LIST*, and *APPLY* and fails the rest. The *PERSON* test fails, because while record subtyping allows method type redefinition, record extension does not.

3.33 Loop

Loop [Eifrig et al. 1995; Eifrig et al. 1995b; 1995a] is a theoretical language similar to PolyTOIL. There are no explicit type annotations in Loop and the typing is inferred automatically. Loop has a concept of *subclassing* where one class *inherits* from several other classes. Subtyping and subclassing in Loop are different concepts. Loop enjoys provably sound type-checking and a state semantics given by its translation to Soop. Function types are present in Loop; however, in the absence of explicit type annotations, they are only used internally for type checking purposes. Loop classes are mechanisms for creating objects. Subclasses do not necessarily correspond to subtypes and class and type inheritance hierarchies are different. The type hierarchy in Loop is implicit, while the class hierarchy is explicitly specified by the programmer. Subclassing can be multiple and both methods and instance variables can be added, inherited, or modified. Since Loop is a "theoretical" language, it does not have any syntactic sugar for subclassing which makes Loop inheritance rather difficult to use. Updates in Loop are allowed for instance variables only. The semantics of these updates is given by their translation to Soop. Loop does not have parametric types. Subtyping in a similar system has been shown to be decidable in [Trifonov and Smith 1996]. The type system of Loop is verifiable, sound, partially expressive, almost uniform, partially reflexive, static, and satis-

fies the inheritance requirements. It passes the same tests as PolyTOIL. The type checking mechanism uses constrained types. The system does not attempt to find a solution to the system of constraints it generates; rather, it verifies that such a system is non-contradictory. The theory guarantees that in this case the system has a solution and the program is considered to be type-correct.

3.34 TL

TL (Tycoon Language) [Matthes and Schmidt 1992; Matthes et al. 1994] is based on the $F_{<:}$ system [Cardelli et al. 1991; Cardelli 1993]. From $F_{<:}$ it borrows constrained (bounded) parametric types and type operators, as well as polymorphic functions and partial type inference. It is also uniform in its treatment of functions (including higher-order ones) and atomic values. In addition to these features, TL has mutable types, modules, and a `typecase` statement. Even though TL is designed on the basis of a formal system ($F_{<:}$), its type system features have not been mathematically proven. Thus, the questions of soundness and decidability remain open for the TL type system. TL is an orthogonally persistent programming language. Since TL is not an object-oriented language, the tests are not applicable to it.

3.35 TooL

In the TooL language [Gawecki and Matthes 1996], an attempt is made to combine the notions of subtyping, matching, and bounded universal quantification. The resulting language is quite powerful in terms of supporting different kinds of relationships between types. However, it has significant complexity and requires good anticipation by type specifiers to correctly choose the kind of type relationship that is needed *before* any subclasses of the class in question are created. The theoretical aspects of the language do not seem to be fully developed, as neither soundness nor decidability of type checking has been proven. TooL supports single dispatch. Interface and implementation subtyping (termed respectively as subtyping and subclassing) are different in TooL. Parametric types can be specified, and the type parameters can be bounded. Information presented in [Gawecki and Matthes 1996] is insufficient to judge the uniformity and reflexivity of the type system; however, it seems to be static and very expressive. [Gawecki and Matthes 1996] also states that TooL is a persistent language; however, nothing else is said about its persistence. The performance of this type system with respect to the test suite is identical to that of the type systems of PolyTOIL and Loop.

3.36 TOFL

The language TOFL [Qian and Krieg-Brueckner 1996] is a theoretical object-oriented functional language. It has multiple dispatch, novariant argument redefinition, function types, and parametric types. TOFL allows subtype specifications of the form "if x is a subtype of Eq, then $list(x)$ is a subtype of Eq for any type x". All parametric types in TOFL are covariant, except for functionals which are novariant in their first argument (function argument position). This is justified since TOFL is a functional (stateless) language. TOFL has a verifiable and provably sound type system. The TOFL type system is also quite expressive as it passes the tests *PERSON*, *POINT*, *SORT*, *GENSORT*, *APPLY*, and the union part of the

test *SET*.

4. CONCLUSIONS

The comparison between the type systems is presented in Table II, Table III, and Table IV. Not all of the languages and systems listed in Table I are present in these tables; those that are superseded by other type systems reviewed in this paper, languages that have no static type systems, and incomplete type systems are excluded from the review tables.

Table II lists features of the reviewed type systems that correspond to the requirements listed in Section 2.6. Verifiability is understood as the presence of a decidable type checking algorithm. Static soundness means that a successfully typechecked program does not produce errors at run-time, while dynamic soundness means that the program reports *all* possible type errors at run-time in a well-defined and predictable manner. Static soundness is strictly stronger then dynamic soundness. The column *uniformity/atomics* means that objects of primitive (atomic) types have the same rights as objects of user-defined types. The column *uniformity/methods* refers to the ability of a language to treat methods (or messages, or both) as objects. *Reflection/typecase* indicates the ability of a language to deal with dynamic type checking in a type-safe manner. Finally, *reflection/evolution* shows whether a given language supports incremental type system evolution.

Table III compares various aspects of type system expressibility. The first two columns indicate whether a given type system has a notion of subtyping, shows what kind of subtyping (structural (implicit) or user-defined (explicit)) it supports, and what kind of inheritance (single or multiple) the type system has. The third column shows whether the type system supports method (function) types. The fourth column addresses the issue of dispatch (single or multiple) in the given type system. Columns 5 through 8 deal with parametricity and its relationship with subtyping. Column 5 indicates whether a given type system supports parametric types. Column 6 shows if a type system can deal with constrained parametric types. Positive indication in column 7 means that a type system makes it possible for different parametric types formed using the same type constructor to have a subtyping relationship with each other (for example, `T_Set(T_Person)` is a subtype of `T_Set(T_Student)`). Column 8 indicates whether the type system is capable of specifying subtyping relationships between parametric types with *different* type constructors (e.g. `T_List(T_Person)` is a subtype of `T_Set(T_Person)`). Column 9 is an indication of the ability of a type system to deal with mutable types. This indication is negative for type systems of purely functional languages. Column 10 shows whether a type system supports intersection (greatest lower bound) types.

Finally, Table IV demonstrates the performance of the reviewed type systems on the test suite.

From the analysis of the results presented in Table IV it can be concluded that languages Mini-Cecil and Transframe rate the best on the test suite. However, none of them has provably sound typechecking; in fact, typechecking Mini-Cecil programs is likely to be undecidable [Litvinov 1998]. Of the systems with sound and verifiable type checking, the most impressive is Sather; however, its type system lacks multiple dispatch and union types. Soundness of the Sather type system

Table II. Type system features

Type system	Verifiability	Soundness		Separation between interface and implementation
		Static	Dynamic	
C++	+	–	–	–
E	+	–	–	–
O++	+	–	–	–
O_2	+	Unknown	Unknown	–
Java 1.1	+	–	Core only	+
GJ	+	–	Unknown	+
JPE	+	–	Unknown	+
Pizza	+	+[1]	+	+
JVT	+	–	Unknown	+
ODMG 2.0	+	Unknown	Unknown	+
SQL-99	+	Unknown	Unknown	–
DBPL	+	Unknown	Unknown	+[2]
Modula-90	+	Unknown	Unknown	+[2]
Modula-3	+	Unknown	Unknown	+[2]
Oberon-2	+	Unknown	Unknown	+[2]
Theta	+	Unknown	Unknown	+
Ada 95	+	Unknown	Unknown	+[2]
Sather	+	+	+	+
Emerald	+	Unknown	Unknown	+
BETA	+	–	Unknown	+[2]
Napier88	+	+	+	+/–
Strongtalk	+	Unknown	Unknown	+
Cecil	+	–	Unknown	+
BeCecil	+	Unknown	Unknown	+
Mini-Cecil	Unknown	Unknown	Unknown	+
Transframe	+	–	Unknown	+
TM	+	Unknown	n/a	–
ML	+	+	n/a	+[2]
Machiavelli	+	+	n/a	–
Fibonacci	+	+	n/a	+/–
$ML_{\leq}$	+	+	n/a	–
PolyTOIL	+	+	n/a	–
Loop	+	+	n/a	+
TL	+	Unknown	Unknown	–
TooL	Unknown	Unknown	n/a	+
TOFL	+	+	n/a	–

Type system	Substitutability	Uniformity		Reflection	
		Atomics	Methods	Typecase	Evolution
C++	+[3]	–	+	–	–
E	+[3]	–	+	–	–
O++	+[3]	–	+	–/+[5]	–
O_2	–	–	–	–	+/–[7]
Java 1.1	+/–	–	–	+	–
GJ	–	–	–	+	–
JPE	–	–	–	+	–
Pizza	–	+	+	+	–
JVT	–	–	–	+	–
ODMG 2.0	–	–	–	+	–
SQL-99	+[4]	–	–	+	+
DBPL	n/a	n/a	+	–	–
Modula-90	+[4]	–	+	+	–
Modula-3	+	–	+	+	–
Oberon-2	+	–	+	+	–
Theta	Unknown	–	+	+	–
Ada 95	–	–	+	+	–
Sather	+	–	+	+	–
Emerald	+	+	–	+	+/–[7]
BETA	+	+	+	+	+/–[7]
Napier88	n/a	n/a	+	+	+
Strongtalk	–	+	+	+	+/–[7]
Cecil	–	+	+	+/–[6]	+
BeCecil	Unknown	+	+	–	+
Mini-Cecil	Unknown	+	+	–	–
Transframe	Unknown	+	+	+	+
TM	–	+	–	–	–
ML	n/a	+	+	–	–
Machiavelli	+	+	+	–	–
Fibonacci	+	+	+	–	–
$ML_{\leq}$	+	+	+	–	–
PolyTOIL	+	+	+	–	–
Loop	+	+	+	–	–
TL	Unknown	+	+	+	–
TooL	Unknown	+	+	–	–
TOFL	+	+	+	–	–

[1] Except for covariant arrays which have been kept in Pizza only for backward compatibility with Java
[2] These features are based on *modules(packages, fragments)* rather than on *types*
[3] Substitutability works for pointers and references only
[4] For object types only
[5] Even though a type can be examined dynamically in O++, typechecking does not take this into account
[6] Classes (implementation types) only
[7] Evolution is type-unsafe

Table III. Type system expressibility

Type system	Subtyping		Method types	Dispatch
		Inheritance		
C++	User-defined	Multiple	+	Single[5]
E	User-defined	Multiple	+	Single[5]
O++	User-defined	Multiple	+	Single[5]
O_2	User-defined	?	–	Single
Java 1.1	User-defined	Multiple[3]	–	Single
GJ	User-defined	Multiple[3]	–	Single
JPE	User-defined	Multiple[3]	–	Single
Pizza	User-defined	Multiple[3]	+	Single
JVT	User-defined	Multiple[3]	–	Single
ODMG 2.0	User-defined	Multiple[3]	–	Single
SQL-99	User-defined	Single	–	Single
DBPL	User-defined	n/a	+	n/a
Modula-90	User-defined	Single	+	Single
Modula-3	User-defined	Single	+	Single
Oberon-2	Structural	n/a	+	Single
Theta	User-defined	Multiple[3]	+	Single
Ada 95	User-defined	Single	+	Single[6]
Sather	User-defined	Multiple	+	Single
Emerald	Structural	n/a	–	Single
BETA	Structural	Single	+	Single
Napier88	–	n/a	+	n/a
Strongtalk	Structural[1]	Single	+[4]	Single
Cecil	User-defined	Multiple	+	Multiple
BeCecil	User-defined	Multiple	+	Multiple
Mini-Cecil	User-defined	Multiple	+	Multiple
Transframe	User-defined	Multiple	+	Multiple
TM	Both	Multiple	–	Single
ML	Structural	n/a	+	n/a
Machiavelli	Structural	n/a	+	n/a
Fibonacci	User-defined[2]	Single	Unknown	n/a
$ML_{\leq}$	Both	Multiple	+	Multiple
PolyTOIL	Structural	n/a	+	Single[7]
Loop	Structural	n/a	+	Single[7]
TL	Structural	n/a	+	n/a
TooL	Structural	Multiple	+	Single[7]
TOFL	User-defined[2]	Multiple[3]	+	Multiple

Type system	Parametric types				Mutable types	Intersection types
		Bounded	Intra	Inter		
C++	+	–	–	+	+	–
E	+	+	–	+	+	–
O++	–	n/a	n/a	n/a	+	–
O_2	–	n/a	n/a	n/a	+	–
Java 1.1	–	n/a	n/a	n/a	+	–
GJ	+	+	–	+	+	–
JPE	+	+	–	+	+	–
Pizza	+	+	–	+	+	–
JVT	+	+	–/+[11]	+	+	–
ODMG 2.0	–/+[8]	n/a	–/+[11]	–	+	–
SQL-99	–/+[13]	–	–/+[11]	–	+	–
DBPL	–	n/a	n/a	n/a	+	–
Modula-90	–	n/a	n/a	n/a	+	–
Modula-3	+[9]	–	n/a	n/a	+	–
Oberon-2	–	n/a	n/a	n/a	+	–
Theta	+	–	–	+	+	–
Ada 95	+	+	–	+	+	–
Sather	+	+	–	+	+	–
Emerald	+	–	n/a	n/a	+	Unknown
BETA	+	+	n/a	n/a	+	–
Napier88	+	–	n/a	n/a	+	–
Strongtalk	+	?	–	+	+	–
Cecil	+	+	–	+	+	+
BeCecil	–	n/a	n/a	n/a	+	+
Mini-Cecil	+	+	+	+	+	+
Transframe	+	+	+/–[12]	+	+	–
TM	+	–	–/+[11]	+	–	–
ML	+	–	n/a	+	+/–	–
Machiavelli	–	n/a	n/a	n/a	+/–	+
Fibonacci	+	–	n/a	n/a	+/–	Implicit
$ML_{\leq}$	+	–	+	+/–	+/–	Implicit
PolyTOIL	+	+	n/a	n/a	+	–
Loop	–	n/a	n/a	n/a	+	Implicit
TL	+	+	–/+[11]	+	+	–
TooL	+	+	–	+	+	–
TOFL	+	++[10]	–/+[11]	+	–	Implicit

[1] Structural subtyping is the default; however, user can explicitly turn it off when needed [2] User-defined for *classes*; structural for *algebraic data types* [3] Only for interfaces; classes have single inheritance [4] Block types only [5] Only *virtual* functions are dispatched [6] Only *tagged types* are dispatched [7] Dispatch modeled as record field extension/execution [8] Only system-defined; parameters can only be used for specification of properties [9] These features are based on *modules* rather than on *types* [10] The language provides mechanism more expressive then bounded quantification [11] Always covariant [12] Only covariant and novariant parameters are allowed [13] Only system-defined: `ARRAY` and `REF`

Table IV. Type system tests

Type system	*PERSON*	*POINT*	*COMPARABLE*	*STREAMS*	*SORT*
C++	–	–	–	–	–
E	–	–	–	–	+
O++	–	–	–	–	–
O_2	–	–	–	–	–
Java 1.1	–	–	–	–	+
GJ	–	–	+	–	+
JPE	–	–	+	–	+
Pizza	–	–	+	–	+
JVT	–/+ [1]	–	–/+ [1]	–/+ [1]	+
ODMG 2.0	–	–	–	–	–
SQL-99	–	–	–	–	–/+ [9]
DBPL	–	–	–	–	–
Modula-90	–	–	–	–	–
Modula-3	–	–	+	–	+
Oberon-2	–	–	–	–	+
Theta	–	–	+	–	+
Ada 95	–	–	+	–	+
Sather	+	–	+	–	+
Emerald	+	–	–	+	+
BETA	–/+ [1]	–	–/+ [1]	–/+ [1]	+
Napier88	–	–	+/– [4]	–	+/– [4]
Strongtalk	–	–	+	–	–/+ [5]
Cecil	+/– [2]	+/– [2]	–	–	+/– [2]
BeCecil	+	+	–	–	+
Mini-Cecil	+	+	+	+	+
Transframe	+	+	+	–	+
TM	–	–	–	–	+
ML	+	–	–	–	+
Machiavelli	+	–	–	–	+
Fibonacci	+	–	–	–	+
$ML_{\leq}$	+	+	–	–	+
PolyTOIL	–/+ [3]	–	+	–	+
Loop	–/+ [3]	–	+	–	+
TooL	–/+ [3]	–	+	–	+
TOFL	+	+	–/+ [1]	–	+

Type system	*GENSORT*	*LIST*	*SET*	*APPLY*	*BROWSER*
C++	+	–	–	–	+/– [7]
E	+	–	–	–	–
O++	+	–	– [6]	–	–/+ [8]
O_2	–	–	–	–	??
Java 1.1	–	–	–	–	+
GJ	–	–	union	–	+
JPE	–	–	union	–	+
Pizza	+	–	union	+	+
JVT	–	–/+ [1]	–/+ [1]	–	+
ODMG 2.0	–	–	union [6]	–	+
SQL-99	–	–	union	–	+
DBPL	+	–	–	–	–
Modula-90	+	–	–	–	+
Modula-3	+	–	–	–	+
Oberon-2	+	–	–	–	+
Theta	+	+	–	–	+
Ada 95	+	+	–	+	+
Sather	+	+	–	–	+
Emerald	–	–	–	–	+
BETA	+	–/+ [1]	–/+ [1]	–/+ [1]	–/+ [1]
Napier88	+/– [4]	+	–	+/– [4]	+
Strongtalk	+	+	union	–/+ [5]	+
Cecil	+/– [2]	+/– [2]	+/– [2]	+/– [2]	+/– [2]
BeCecil	+	+	–	–	–
Mini-Cecil	+	+	+	+	–
Transframe	+	+	union	+	+
TM	–	+	– [6]	–	–
ML	+	–	–	+	–
Machiavelli	+	–	+/– [6]	+	–
Fibonacci	+	–	+/– [6]	+	–
$ML_{\leq}$	+	–	union	+	–
PolyTOIL	+	+	–	+	–
Loop	+	+	–	+	–
TooL	+	+	–	+	–
TOFL	+	–	union	+	–

[1] Relies on dynamic type checking
[2] Only suggestive type-checking
[3] While the subtyping relationship required by the test holds, one type can not be derived from the other
[4] If type parameters are explicitly instantiated
[5] This test can be only programmed with blocks rather than messages
[6] Built-in operators for dealing with sets are provided; however, their typing is special (non-user-definable)
[7] If RTTI is present
[8] Even though a type can be examined dynamically, typechecking does not take this into account
[9] Only using a special support for sorting and comparison in SQL-99.

has not been formally proven. Almost all "theoretical" type systems show similar performance on the test suite.

Sound type checking, substitutability, parametricity, method types, and multi-methods appear together in only one type system: that of $ML_{\leq}$. However, $ML_{\leq}$ (as well as most ML clones) severely restricts usage of mutable types and does not deal well with certain aspects of binary methods and parametricity (failed tests *COMPARABLE* and *STREAMS*).

The test *STREAMS* proved to be the most difficult one. This is surprising as the test outlines the situation that occurs in almost every language dealing with I/O operations at a relatively high level. Only the type systems of Emerald and Mini-Cecil were able to pass this test; however, none of these type systems has a proof of soundness.

Overall, type systems with nice theoretical properties show only moderate performance on the test suite, while type systems that perform well on tests lack a theoretical basis.

It can, therefore, be concluded that none of the languages reviewed completely satisfies the requirements laid down in Section 4. None of the provably sound type systems has passed the majority of the tests. However, every test was passed by at least one type system thus showing that the necessary mechanisms have already been developed. It is their consistent and theoretically sound combination that remains elusive so far.

A type system that satisfies the requirements has been developed [Leontiev 1999]. This type system was not reviewed in the paper since it was developed specifically to conform to the requirements listed in Section 2. However, it demonstrates that these requirements are consistent in that they can be satisfied together in a single theoretically sound type system. It is our belief that this type system will be an important first step towards the development of theoretically sound, reflexive, uniform, and dynamic persistent object-oriented programming language.

REFERENCES

Agesen, O., Freund, S. N., and Mitchell, J. C. 1997. Adding type parametrization to the Java language. In *Proceedings of the OOPSLA'97*.

Agrawal, R. and Gehani, N. H. 1989. ODE (object database and environment): The language and the data model. In *Proceedings of the ACM-SIGMOD 1989 International Conference on Management of Data*. 36–45.

Aiken, A. and Wimmers, E. L. 1993. Type inclusion constraints and type inference. Tech. Rep. RJ 9454 (83075), IBM Research Division. August.

Aiken, A., Wimmers, E. L., and Lakshman, T. K. 1994. Soft typing with conditional types. In *Conference Record of POPL '94, 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 163–173.

Alagić, S. 1997. The ODMG object model: Does it make sense? *SIGPLAN Notices 32,* 10, 253–270.

Alagić, S. 1999. Type-checking OQL queries in the ODMG type systems. *ACM Transactions on Database Systems 24,* 3 (September), 319–360.

Albano, A., Cardelli, L., and Orsini, R. 1985. Galileo: A strongly-typed, interactive conceptual language. *ACM Transactions on Database Systems 10,* 2 (June), 230–260.

Albano, A., Ghelli, G., and Orsini, R. 1995. Fibonacci: A programming language for object databases. *VLDB Journal 4*, 403–444.

Apple Computer, Inc. 1994. *Dylan Interim Reference Manual*. Apple Computer, Inc.

Armstrong, J., Virding, R., Wikström, C., and Williams, M. 1996. *Concurrent Programming in Erlang*, 2nd ed. Prentice Hall.

Arnold, K. and Gosling, J. 1996. *The Java Language Specification*, 4th ed. Addison-Wesley. ISBN 0-201-63455-4.

Atkinson, M., Banchilon, F., DeWitt, D., Dittrich, K., Maier, D., and Zdonik, S. 1992. The object-oriented database system manifesto. In *Building an Object-Oriented Database System: The Story of O_2*, F. Banchilon, C. Delobel, and P. Kanellakis, Eds.

Atkinson, M. and Morrison, R. 1995. Orthogonally persistent object systems. *VLDB Journal 4,* 3, 319–401.

Atkinson, M. P. and Buneman, O. P. 1987. Types and persistence in database programming languages. *ACM Computing Surveys 19,* 2 (June), 105–190.

Bal, R., Balsters, H., de By, R. A., Bosschaart, A., Flokstra, J., Keulen, M. V., Skowronek, J., and Termorshuizen, B. 1993. *The TM Manual.* Faculty of Computer Science, University of Twente. Version 2.0 revision C. Available electronically.
URL: `ftp://ftp.cs.utwente.nl/pub/doc/TM`

Baumgartner, G., Läufer, K., and Russo, V. F. 1996. Interaction of object-oriented design patterns and programming languages. Tech. Rep. CSD-TR-96-020, Deptartment of Computer Sciences, Purdue University.

Birtwistle, G. M., Dahl, O.-J., Myhrhaug, B., and Nygaard, K. 1979. *Simula Begin.* Studentlitteratur (Lund, Sweden), Bratt Institute Fuer Neues Lerned (Goch, FRG), Chartwell-Bratt Ltd (Kent, England).

Black, A. and Palsberg, J. 1994. Foundations of object-oriented languages. *ACM SIGPLAN Notices 29,* 3, 3–11. Workshop Report.

Bobrow, D. G., DeMichiel, L. G., Gabriel, R. P., Keene, S. E., Kiczales, G., and Moon, D. A. 1988. Common Lisp Object System specification. X3J13 Document 88-002R.

Bourdoncle, F. and Merz, S. 1996a. Primitive subtyping $\vee$ implicit polymorphism $\vdash$ object-orientation. In *Foundations of Object-Oriented Languages 3.* Extended abstract.

Bourdoncle, F. and Merz, S. 1996b. Type checking higher-order polymorphic multi-methods. In *Proceedings of the 24th ACM Conference on Principles of Programming Languages (POPL'24).*

Boyland, J. and Castagna, G. 1995. Type-safe compilation of covariant specialization: A practical case. Tech. Rep. UCB/CSD-95-890, University of California, Computer Science Division (EECS), Berkeley, California 94720. November.

Boyland, J. and Castagna, G. 1997. Parasitic methods: An implementation of multi-methods in Java. *SIGPLAN Notices 32,* 10, 66–76. Proceedings of OOPSLA'97.
URL: `ftp://ftp.ens.fr/pub/dmi/users/castagna/oopsla97.ps.gz`

Bracha, G. and Griswold, D. 1993. Strongtalk: Typechecking Smalltalk in a production environment. In *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications.*

Bracha, G., Odersky, M., Stoutamire, D., and Wadler, P. 1998a. GJ: Extending the Java programming language with type parameters. Manuscript. Revised August 1998.

Bracha, G., Odersky, M., Stoutamire, D., and Wadler, P. 1998b. GJ specification. Manuscript.

Bracha, G., Odersky, M., Stoutamire, D., and Wadler, P. 1998c. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of the 13th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'98).* 183–200.

Bruce, K., Cardelli, L., Castagna, G., The Hopkins Object Group, Leavens, G., and Pierce, B. 1996. On binary methods. *Theory and Practice of Object Systems 1,* 3, 221–242.

Bruce, K. B., Fiech, A., and Petersen, L. 1996. Subtyping is not a good "match" for object-oriented languages. In *Informal Proceedings of The Fourth Workshop on Foundations of Object-Oriented Languages (FOOL 4).* Contributed talk.

Bruce, K. B., Odersky, M., and Wadler, P. 1998. A statically safe alternative to virtual types. In *Proceedings of the 1998 European Conference on Object-Oriented Programming (ECOOP'98).*

Bruce, K. B., Schuett, A., and Gent, R. V. 1994. A type-safe polymorphic object-oriented language. Accessible by anonymous FTP.
URL: `ftp://cs.williams.edu/pub/kim/PolyTOIL.dvi`

Bruce, K. B., Schuett, A., and Gent, R. V. 1995. PolyTOIL: A type-safe polymorphic object-oriented language. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*, W. Olthoff, Ed. LNCS 952. Springer, Åarhus, Denmark. Extended abstract.

Bruce, K. K. 1996. Typing in object-oriented languages: Achieving expressibility and safety.
URL: `ftp://cs.williams.edu/pub/Kim/Static.ps`

Buneman, P. and Ohori, A. 1996. Polymorphism and type inference in database programming. *ACM Transactions on Database Systems 21,* 1 (March), 30–76.

Buneman, P. and Pierce, B. 1999. Union types for semistructured data. Tech. Rep. MS-CIS-99-09, Department of CIS, University of Pennsylvania.

Cardelli, L. 1986. A polymorphic λ-calculus with **Type:Type**. Tech. Rep. 10, DEC SRC, 130 Lytton Avenue, Palo Alto, CA 94301. May. SRC Research Report.

Cardelli, L. 1988. A semantics of multiple inheritance. *Information and Computation 76*, 138–164.

Cardelli, L. 1989. Typeful programming. In *Formal Description of Programming Concepts*, E. J. Neuhold and M. Paul, Eds. IFIP State of the Art Reports Series. Springer-Verlag.
URL: `http://www.luca.demon.co.uk/Bibliography.html`

Cardelli, L. 1993. An implementation of $F_{<:}$. Tech. Rep. 97, DEC Systems Research Center. February.

Cardelli, L. 1997. Type systems. In *The Computer Science and Engineering Handbook*, A. B. Tucker, Ed. CRC Press, Chapter 103.
URL: `http://www.luca.demon.co.uk/Bibliography.html`

Cardelli, L., Martini, S., Mitchell, J. C., and Scedrov, A. 1991. An extension of system F with subtyping. In *International Conference on Theoretical Aspects of Computer Software*, T. Ito and A. R. Meyer, Eds. 750–770. Lecture Notes in Computer Science 526.

Castagna, G. 1996. *Object-Oriented Programming: A Unified Foundation.* Progress in Theoretical Computer Science. Birkaüzer, Boston, Chapter Type Systems for Object-Oriented Programming.

Castagna, G., Ghelli, G., and Longo, G. 1995. A calculus for overloaded functions with subtyping. *Information and Computation 117,* 1 (February), 115–135.

Cattell, R. G. G., Barry, D., Bartels, D., Berler, M., Eastman, J., Gamerman, S., Jordan, D., Springer, A., Strickland, H., and Wade, D. 1997. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann Publishers, Los Altos (CA), USA.

Chambers, C. 1992. Object-oriented multi-methods in Cecil. In *ECOOP '92, European Conference on Object-Oriented Programming, Utrecht, The Netherlands*, O. L. Madsen, Ed. Lecture Notes in Computer Science, vol. 615. Springer-Verlag, New York, N.Y., 33–56.

Chambers, C. 1993. The Cecil language: Specification and rationale. Tech. Rep. TR 93-03-05, Department of Computer Science and Engineering, FR-35, University of Washington. March.

Chambers, C. and Leavens, G. T. 1996. BeCecil, A core object-oriented language with block structure and multimethods: Semantics and typing. Tech. Rep. 96-17, Department of Computer Science, Iowa State University. December.

Chambers, C. and Leavens, G. T. 1997. BeCecil, A core object-oriented language with block structure and multimethods: Semantics and typing. In *FOOL 4, The Fourth International Workshop on Foundations of Object-Oriented Languages, Paris, France.*

Chen, K. and Odersky, M. 1994. A type system for a lambda calculus with assignment. In *Proc. Theoretical Aspects of Computer Science, Sendai, Japan.* Spinger LNCS.

Connor, R. C. H., McNally, D. J., and Morrison, R. 1991. Subtyping and assignment in database programming languages. In *Proceedings of the 3rd International Workshop on Database Programming Languages.* Napfilon, Greece.

Day, M., Gruber, R., Liskov, B., and Myers, A. C. 1995. Subtypes vs where clauses: Constraining parametric polymorphism. *SIGPLAN Notices 30,* 10 (October), 156–168.

Drossopoulou, S. and Eisenbach, S. 1997. Java is type safe — probably. In *Proceedings of the 11th European Conference on Object Oriented Programming (ECOOP'97)*.

Eifrig, J., Smith, S., and Trifonov, V. 1995a. Sound polymorphic type inference for objects. *SIGPLAN Notices 30,* 10 (October), 169–184.

Eifrig, J., Smith, S., and Trifonov, V. 1995b. Type inference for recursively constrained types and its application to OOP. *Electronic Notes in Theoretical Computer Science 1.*
URL: `http://www.elsevier.nl/locate/entcs/volume1.html`

Eifrig, J., Smith, S., Trifonov, V., and Zwarico, A. 1995. An interpretation of typed OOP in a language with state. *LISP and Symbolic Computation 8,* 4, 357–397.

Fisher, K. and Mitchell, J. C. 1996. The development of type systems for object-oriented languages. *Theory and Practice of Object Systems 1,* 3, 189–220.
URL: `ftp://theory.stanford.edu/pub/jcm/papers/tapos.ps`

Franz, M. 1997. The programming language Lagoona — A fresh look at object-orientation. *Software — Concepts and Tools 18*, 14–26.

Gawecki, A. and Matthes, F. 1996. Integrating subtyping, matching and type quantification: A practical perspective. In *Proceedings of the 10th European Conference on Object-Oriented Programming*. Springer Verlag, Linz, Austria.

Goldberg, A. and Robson, D. 1989. *ST-80, The Language*. Addison-Wesley.

Harbison, S. P. 1992. *Modula-3*. Prentice Hall.

Hauck, F. J. 1993. Towards the implementation of a uniform object model. In *Parallel Computer Architectures: Theory, Hardware, Software, and Applications – SFB Colloquium SFB 182 and SFB 342*, A. Bode and M. D. Cin, Eds. Number 732 in Lecture Notes in Computer Science. Springer, 180–189.

Kempe Software Capital Enterprises 1995. *Ada 95 Reference Manual*. Kempe Software Capital Enterprises. Available electronically.
URL: `http://www.adahome.com/rm95`

Kim, W. 1993. Object-oriented database systems: Promises, reality, and future. In *Proceedings of the 19th VLDB Conference*. 676–687.

Kirby, G. N. C., Connor, R. C. H., Morrison, R., and Stemple, D. 1996. Using reflection to support type-safe evolution in persistent systems. Tech. Rep. CS/96/10, University of St Andrews.

Klas, W. and Turau, V. 1992. Persistence in the object-oriented database programming language VML. Tech. Rep. TR-92-045, International Computer Science Institute, 1947 Center St., Suite 600, Berkeley, CA 94704-1198. July.

LaLonde, W. R. and Pugh, J. 1991. Subclassing $\neq$ subtyping $\neq$ is-a. *Journal of Object-Oriented Programming 3,* 5 (January), 57–62.

Leavens, G. T. and Millstein, T. D. 1998. Multiple dispatch as dispatch on tuples. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*. 374–387.
URL: `http://www.cs.washington.edu/homes/todd/papers/oopsla98.ps`

Lécluse, C., Richard, P., and Vélez, F. 1992. O_2, an object-oriented data model. In *Building an Object-Oriented Database System: The Story of O_2*, F. Banchilon, C. Delobel, and P. Kanellakis, Eds.

Leontiev, Y. 1999. Type system for an object-oriented database programming language. Ph.D. thesis, Department of Computing Science, University of Alberta. Also available as Technical Report TR 99-02.

Leontiev, Y., Özsu, M. T., and Szafron, D. 1998. On separation between interface, implementation, and representation in object DBMSs. In *Proceedings of TOOLS–26'98*. Santa Barbara, California.

Litvinov, V. 1998. Constraint-based polymorphism in Cecil: Towards a practical and static type system. In *Proceedings of the 1998 Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'98)*.

Lutiy, V. G., Merkov, A. B., Leontiev, Y. V., Gawrilow, E. J., Ivanova, N. A., Iofinova, M. E., Paklin, M. L., and Hodataev, A. K. 1994. *DBMS Modula–90K.* RAN Data Processing Center, Moscow. /in Russian: Sistema Programmirovaniya Baz Dannyh Modula-90K/.

Madsen, O. L., Møller-Pedersen, B., and Nygaard, K. 1993. *Object-Oriented Programming in the BETA Programming Language.* Addison-Wesley. ISBN 0-201-62430-3.

Marlow, S. and Wadler, P. 1997. A practical subtyping system for Erlang. In *Proceedings of the Second International Conference on Functional Programming.* Amsterdam.

Matthes, F., Müssig, S., and Schmidt, J. W. 1994. Persistent polymorphic programming in Tycoon: An introduction. FIDE Technical Report Series FIDE/94/106, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ. August.

Matthes, F. and Schmidt, J. 1991. Bulk types: Built-in or add-on? In *Proceedings of the Third International Workshop on Database Programming Languages.* Morgan Kaufmann Publishers.

Matthes, F. and Schmidt, J. 1992. Definition of the Tycoon language — a preliminary report. Tech. Rep. FBI-HH-B-160/92, Universität Hamburg. October.

Meyer, B. 1988. *Eiffel — the language.* Prentice-Hall.

Miller, R., Tofte, M., and Harper, R. 1990. *The Definition of Standard ML.* MIT Press.

Morrison, R., Brown, F., Connor, R., Cutts, Q., Dearle, A., Kirby, G., and Munro, D. 1996. *Napier88 Reference Manual.* University of St. Andrews. Release 2.2.1.

Mössenbök, H. and Wirth, N. 1993. The programming language Oberon-2. Manuscript. Institut für Computersysteme, ETH Zürich.

Myers, A. C., Bank, J. A., and Liskov, B. 1997. Parameterized types for Java. In *Proceedings of the 24th ACM Simposium on Principles of Programming Languages (POPL'97).*

Odersky, M. and Läufer, K. 1996. Putting type annotations to work. In *Proc. 23rd ACM Symposium on Principles of Programming Languages.* 65–67.

Odersky, M. and Wadler, P. 1997. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Simposium on Principles of Programming Languages (POPL'97).*

Ohori, A., Buneman, P., and Breazu-Tannen, V. 1989. Database programming in Machiavelli — a polymorphic language with static type inference. *SIGMOD Record 18,* 2, 46–57.

Pottier, F. 1998. Type inference in the presence of subtyping: from theory to practice. Ph.D. thesis, Université Paris VII.

Qian, Z. and Krieg-Brueckner, B. 1996. Typed OO functional programming with late binding. In *Proceedings of the 10th European Conference on Object-Oriented Programming*, P. Cointe, Ed. LNCS, vol. 1098. Springer, 48–72.

Raj, R. K., Tempero, E., Levy, H. M., Black, A. P., Hutchinson, N. C., and Jul, E. 1991. Emerald: A general-purpose programming language. *Software Practice and Experience 21,* 1 (January), 91–118.

Rehof, J. 1998. The complexity of simple subtyping systems. Ph.D. thesis, DIKU, Department of Computer Science, University of Copenhagen.

Richardson, J. E., Carey, M. J., and Schuh, D. T. 1993. The design of the E programming language. *ACM Transactions on Programming Languages and Systems 15,* 3 (July), 494–534.

Roe, P. and Szyperski, C. 1997. Lightweight parametric polymorphism for Oberon. In *Proceedings of the Joint Modular Languages Conference.*

Saraswat, V. 1997. Java is not type-safe. Available electronically.
URL: `http://www.research.att.com/~vj/bug.html`

Schmidt, J. W. and Matthes, F. 1994. The DBPL project: Advances in modular database programming. *Information Systems 19,* 2, 121–140.

Sequeira, D. 1998. Type inference with bounded quantification. Ph.D. thesis, Department of Computer Science, University of Edinburgh. Also Technical Report ECS–LFCS–98–403.

Shang, D. 1997. *Transframe: The Annotated Reference.* Software Systems Research Laboratory, Motorola, Inc., Schaumburg, Illinois. Draft 1.4.

SQL 1999. Database languages – SQL. American National Standard for Information Technology. ANSI/ISO/IEC 9075-1-1999, 9075-2-1999, 9075-3-1999, 9075-4-1999, 9075-5-1999.

Stoutamire, D. and Omohundro, S. 1996. The Sather 1.1 specification. Tech. Rep. TR-96-012, International Computer Science Institute at Berkeley. August.

Stroustrup, B. 1991. *The C++ Programming Language.* Addison-Wesley.

Taivalsaari, A. 1996. On the notion of inheritance. *ACM Computing Surveys 28,* 3 (September), 439–479.

Thorup, K. K. 1997. Genericity in Java with virtual types. In *Proceedings of the 1997 European Conference on Object-Oriented Programming (ECOOP'97).*

Trifonov, V. and Smith, S. 1996. Subtyping constrained types. In *Proceedings of the 3rd International Static Analysis Symposium.* 349–365. Lecture Notes in Computer Science 1145.

Tsichritzis, D., Nierstrasz, O., and Gibbs, S. 1992. Beyond objects: Objects. *International Journal of Intelligent and Cooperative Information Systems 1,* 1 (March), 43–60.

W3C 2002. *XQuery 1.0 Formal Semantics.* W3C. Working Draft 26.
URL: `http://www.w3.org/TR/2002/WD-query-semantics-20020326/`

Wirth, N. 1983. *Programming in Modula-2*, 2nd ed. Springer-Verlag.

Wright, A. K. 1993. Polymorphism for imperative languages without imperative types. Tech. Rep. TR93-200, Department of Computer Science, Rice University. February.

Received June 2000; revised December 2001; accepted May 2002