HOOD:
A Higher-Order Object-Oriented Database Model and its Implementation

by Michael Brand

A Thesis
Prepared Under the Supervision of
Associate Professor P.T. Wood
In Fulfillment of the Requirements for the
Degree of Master of Science in
Computer Science

University of Cape Town
March 1992

Abstract

There is no accepted standard for the object-oriented database paradigm at present, which has led to different definitions of features and conformance requirements. HOOD is a Higher-Order Object-Oriented Database system which defines a meta-data model for specifying the requirements of an Object-Oriented Database, which provides uniformity and extensibility. From this specification and by making use of a comprehensive structure system, an exemplar or implementation model is defined.

Among the constructs provided by the model are types, instances, objects, values, methods, base types, generic types and metatypes. The mechanisms of instantiation and subtyping allow for relationships between these constructs. Extensibility is provided in the model for types, base types, structures and methods. Uniformity is achieved by defining all constructs as instances and through the use of messages for all operations. There is only one form of object construct which provides persistence and identities. The complex values and extensibility of the model allow it to adapt in order to model the real world instead of adapting the real world to fit the model.

We have implemented a subset of the structures and values defined in the model, provided persistence and identities for object, and included the various constructs mentioned above. The method language allows for the specification of methods, the passing of messages, and the use of complex values. The compiler performs type checking and resolution and generates instructions for an abstract machine which manipulates the database.

Table of Contents

Acknowledgements

I would like to thank my supervisor, Associate Professor Peter Wood, for his guidance and assistance through the various stages of this work and course of my studies. I would also like to thank my parents for their support over the past six and a half years, and the FRD for their support for the past three years. The Department of Computer Science at UCT and my fellow students Graham Wheeler, Andrew Hutchison, Roland Patterson-Jones and Sandi Donno also deserve acknowledgement for their assistance.

Chapter 1 Introduction

Databases have traditionally been used for business applications and, as a result, much of their design has been based on this application domain. There are, however, limitations with relational databases, for example, with regard to their modelling capabilities and computational completeness. The general solution to these problems is to embed a database language within a programming language. This combines the persistence and reliability of data (as found in a database system) with the expressive power of a programming language. This solution is, however, renowned for the impedance mismatch [ABDDMZ90] which occurs between the two languages. On the other hand, programming languages lack the persistence feature found in databases which has led to a new class of persistent programming languages which offer persistence as a basic language construct. This class also includes persistent object-oriented programming languages.

The application domain for databases is diversifying and now includes CAD/CAM, CASE and Office Automation, to name but a few. They bring with them a whole new set of requirements which database systems must satisfy. The applications require the basic database features of persistence, reliability and shared data. But they also require computational completeness and a single system in which both data and operations can be stored in a uniform manner. Object-Oriented systems offer a solution to this problem of uniformity. They also provide an extensible set of constructs which can be used to adapt the system so that it models the real world, as opposed to the all too familiar scenario of adapting the real world to fit the system.

Object-Oriented databases seem to be the result of convergence of a number of systems which are being developed from starting points which are in many different fields, such as: databases, object-oriented programming languages and semantic data models. There are a number of data models, prototypes and commercial systems which have been developed and labeled as object-oriented database systems. There is, as yet, no accepted standard for the field and this explains the proliferation of terms and concepts which have been used. Many of the terms and concepts are ambiguous due to their multiple definitions in the literature.

In Chapter 2, we discuss a number of the features found in those object-oriented database systems which have been published in the literature. The main features of these systems are those of objects, methods, and classes or types. An object is used to model an entity in the real world. Associated with an object is a set of operations, called methods. An object is passed a message and it selects the method it will use to respond to the request. Objects can be grouped together in classes or types, which contain the general definitions and operations of the objects which are members of the group. A class models a concept in the real world. For example, a person by the name of Fred can be modelled in a system by an object. The action of Fred getting married can be defined by a method which may be applied to the object. A class or type can be used to model all people and contains Fred as one of its objects. All

of the definitions and operations for people are defined in the class or type and may be shared by all of the objects which are its members.

Our goal is to define an object-oriented system which operates in a uniform manner, and which can be extended to meet the requirements of the application as they arise. This endows the system with the capability of providing a true reflection of the real world. To achieve this goal within the time constraints, we have defined a data model which specifies an object-oriented database. We have implemented the kernel of the system which is responsible for representing values on disk and in memory, transferring them between disk and memory, and manipulating them. These values are used to define the types and instances which occur in the database. A method language has been defined and a compiler implemented to produce methods which can be used to manipulate the database. The main tasks of the compiler are type checking, type resolution, and code generation.

The data model specifies the overall design for an object-oriented database. In the data model we define the database feature of persistence and a number object-oriented features. Persistence is achieved through the use of objects which are defined in the object system. An object models an entity in the real world and is defined by a type. A type defines the structure of a set of instances and the methods which can be applied to them. Instances are split into values and objects with identities. Identity provides a means of identifying an independent object uniquely. For the user there is an intuitive correspondence between the entities in the real world and the objects in the system.

One type may be specified as a subtype of another type by utilizing the subtyping mechanism. The subtype can make use of methods which are defined in the supertypes. The advantage of this is code reuse for the methods as well as a structuring of the system. All instances are encapsulated and may only be manipulated by methods defined in their types. As a result the system is easy to maintain due to modularity.

To cater for uniformity, we have defined everything in the system as an instance. Thus, types and methods are instances. The model also allows the concept of a metatype which has types as its instances. Methods are treated in the same way as other instances, since there are operations which can be applied to them. By defining everything as an instance, all operations are performed in a uniform way, as a result of which the system is simple.

Primitive values such as integers and strings are provided by a set of extensible base types. New base types may be defined and implemented outside the system and then linked into it.

In order to create the values in the model, an object and a structure system are used to define the basic values and structures. The structure system provides record, set, list, product, array, sum, base value and higher-order structures. These structures define values and the operations which may be performed on them. Structural subtyping is defined for structures and allows types to be specified as subtypes. The object system provides persistence and defines objects and identity.

A generic type allows a more generalized concept to be defined in the system. It defines a type in which values, structures and methods may be abstracted to variables. The generic type is used by specifying actual values for the variables. Generic types allow a single type to be created and reused

where a number of types would otherwise have had to have been created with a large amount of duplication. They add to the extensibility of the system because they allow general types to be defined and also can be used to define new constructors.

The system is called HOOD (Higher-Order Object-Orientated Database), because of the higher-order or meta features that it supports. Structures and methods are treated as values. A type is a special form of an object which is defined with the aid of a metatype. A type is treated exactly the same way as any other object. The method values in the system are stored as objects too. The model is essentially a metamodel since it specifies constructs in terms of which a specific data model may be defined. A set of constraints is specified which a specific data model must adhere to. The definition of a specific system is reflective which also adds to the uniformity, since the structures and operations provided by the system are defined in the system.

Recovery and reliability, concurrency, and storage management are important features of a database system. In this system, however, they have been omitted, since we have concentrated on object-oriented features and modeling capabilities. These features may be added to the system at a later stage, in order to make it a complete object-oriented database system.

The implementation provides representation in memory and on disk for the structures and values defined by the structure system. The primitive operations which are defined for these values are implemented as instructions for an abstract machine which manipulates the database. The object system provides the persistence of objects which are stored in an objects table, by simply reading all objects from disk at the beginning of a session and writing them back at the end of the session.

The system provides type checking by ensuring the constraints for the type system are satisfied. The compiler performs type checking and type resolution for values and expressions specified in the method language. The language allows for the creation of complex values using record, set and list constructors, and for complex operations such as apply-to-all. The compiler checks the validity of operations and ensures the encapsulation of instances, generating code used by the abstract machine.

The next chapter describes the background of the field, in which we describe the current state of the object-oriented database field as is discussed in the literature. Chapter 3 provides an overview of the data model and implementation. A reference list example which is used throughout the thesis is presented here. Chapter 4 discuses the data model in detail, first by presenting the structure and object systems. This is followed by the type system in which the functions and constraints are defined. We end the chapter by presenting an exemplar which specifies a particular implementation for an object-oriented database. In Chapter 5 we discuss the implementation of the system, by presenting the structures used to represent the values in the system. This is followed by a description of the method language compiler, with an emphasis on the type checking and resolution that has been used. We end the chapter by discussing the abstract machine that performs the operations on the database. Our conclusions and suggestions for future work are presented in Chapter 6.

Chapter 2 Background

In this chapter we present the paradigm of Object-Oriented Databases (OODBs) as it is defined and discussed in the literature. We begin with a historical perspective describing the motivation for OODBs and the current state of the paradigm. The features and concepts relating object-oriented systems, database systems and specifically object-oriented database systems are discussed in detail. This provides an insight into the various features found in a system. The virtues of these features is expounded, while their sources of derivation explain some of the conflicts that arise. A number of authors have suggested features deemed essential to an OODB which are discussed in the conformance section. The chapter is concluded with the standardization efforts being made by ANSI.

2.1 Perspective

The Object-Oriented metaphor was first used in the programming language Simula67 and was popularized by the programming language Smalltalk. Of late, a number of programming languages have been developed that make use of Object-Oriented features. Databases have traditionally been used in the business environment and much of their design has been based on the requirements of this application domain.

Object-Oriented Databases are evolving from different paradigms, viz. databases, programming languages, semantic data models, knowledge bases and also in their own right. Conventional databases lack the expressive power found in programming languages and the modelling capabilities of semantic data models and knowledge bases. On the other hand, conventional programming languages lack the efficient handling of persistent data which is provided for by databases.

[Banc88] identifies three phenomena which are appearing in the database field. First there are new non-business-type applications [Masu90] such as Computer Aided Design (CAD), Computer Aided Manufacturing (CAM), Computer Aided Software Engineering (CASE), Office Automation (OA), Factory Automation (FA), and Artificial Intelligence (AI) which require the database features of large amounts of persistent, reliable, shared data. Second, the hardware costs of memory and disk are changing, prompting a re-evaluation of current database system design. Third, the solution used in most commercial systems is to embed a database language into a programming language. This gives rise to the *impedance mismatch* between data manipulation language (DML) and application development languages. There are two problems coupled with this mismatch: first is the difference in programming paradigms such as declarative (SQL) versus imperative (PL/I). The second is the difference in the type systems. OODBs try to solve the problem by extending the DML so that the entire application can be written in it [ZdMa90].

These phenomena have provided motivation for the progression being made towards OODBs, the three main factors being identified in [Banc88]. Firstly, object-oriented systems required database functionality. Secondly, database people are looking for solutions to the impedance mismatch problem and OODBs seem to be a promising solution. They provide the necessary framework for managing both data and operations. Thirdly, there is a general interest by the database community in object-oriented systems for their modelling power.

> *"The overall objective of the field is to integrate database technology and the object-oriented approach in a single system." [Banc88]*

Some programming languages have moved closer to databases by adding persistence as one of their language features, while retaining their expressive power. These are known as persistent programming languages. On the other hand, databases have added some of the features found in programming languages to increase their expressive power, in an attempt to become computationally complete.

There is a convergence towards Object-Oriented Databases because they support the traditional database features of persistence and reliability. Added to this are the modelling capabilities of knowledge bases [ZdMa90] and semantic data models [Osbo89], the expressive power of programming languages and a framework which maintains both data and operations.

In OODB modelling, a real world object corresponds directly with an object in the database and the user can intuitively identify this correspondence. Encapsulation allows for the modelling of complex objects as single entities, which correspond with the complex structures in the real world. A class allows for the description of the generalized concept, which yields implementation efficiency [Masu90].

2.1.1 Present State

There is at present no standard specification for an object-oriented database system. This is quite different from the development of the relational model, where Codd presented his paper [Codd70] which defined it. From this starting point relational database systems were constructed. Consensus is slowly being reached for the definition of an object-oriented system, which clearly influences the formulation of a data model for an object-oriented database system.

Since an accepted model is absent, there is no strong theoretical framework from which to work. The semantics of terms are often ill defined and the resulting conceptual discrepancies create stumbling blocks in the process of determining a standard data model.

There is a large amount of experimentation at present: people are working on theoretical models, prototypes and commercial systems. Experimentation is good because it generates a spectrum of systems and features, from which the best will hopefully emerge. We say "hopefully" because in reality of the first few systems to emerge, the one which gains popularity usually becomes the de facto standard. From then on all emerging systems are overshadowed, even if they are superior. The result is that often the best system is not accepted as the standard for the field [ABDDMZ90]. In [AbKa89],

there is a prediction that OODBs will be the next generation of commercial databases. It certainly seems that OODBs provide solutions to a number of the problems which are currently being encountered.

2.2 Concepts

The concepts and features used to define an OODB are expounded here. There are, however, differences in terminology and concepts in the literature, with which we have attempted to deal. According to [Wegn87], a goal of any system is for its features to be orthogonal (defined independently of the other features) and consistent. This provides the system more flexibility and power.

We discuss the following concepts and features: Objects, Classes and Types, Inheritance and Subtyping, Computational Completeness, Encapsulation, Typing, Metatypes and Metaclasses, Extensibility, Queries, Persistence, Concurrency, Reliability and Recovery, Storage Management, Multiple Users, Distributed Systems, Versions and Histories.

2.2.1 Objects

[Coin87], [LRV88], [Masu90], [Wegn87], [ZhRo88], [ZdMa90] and many others define the concept of an object. Essentially an object models an entity in the real world. There is data associated with the entity which is stored in the *state* of the object (see Section 2.2.1.2 below). The object is manipulated by a set of operations which are defined for it. These operations may be thought of as the capabilities or behaviour of the object, or the interface through which it can be accessed.

For example, if Fred is a person and an entity in the real world, he may be modelled as an object in the database. Fred has a name and a date of birth which are stored in the object's state. A user of the system may or may not have direct access to the data stored in the state of the object, but an operation can be used to obtain the person's name. So, if we apply the name operation to our object, we will get the name "Fred". Since age is a property of people, an operation is used to access the person's date of birth and hence calculate their age.

Associated with objects are the features of object identity, state structures, complex objects, relationships, composite objects, graphical representations and operations. These features are each dealt with in the following subsections.

2.2.1.1 Identity

Assume that Jill is another person being modelled in the database by an object. Furthermore, assume both Fred and Jill have a child by the name of Jack. There are two cases to be considered: either (1) Fred and Jill have the same child Jack, or (2) there are two distinct children by the name of Jack. In a relational system we would store with each person the information regarding their children. If both parents are stored in the database, then the information regarding their children is duplicated and there

is no way of knowing which one of the two cases above is being modelled. If modelling case (1), any update made to the information regarding the child of one parent, a duplicate update will be required for the other parent in order to keep the two sets of information consistent.

[AbKa89], [DaTo88], [KhCo86], [LéRi89], [LRV88], [ZdMa90], [Zdon89] all define the concept of object identity which has also been called a surrogate, i-value, object identifier (OID) and object-oriented pointer (OOP). Object identity is associated with an object for the purpose of uniquely identifying it and is an essential part of an OODB. There is a one-to-one correspondence between the object and its identity. In relational systems a key is used to identify a tuple in a relation but it is dependent on the data and is limited to a specific relation.

According to [KhCo86] there are two dimensions involved in the support of identity. One is the representation dimension which comprises the structures used by a language (system) to represent the identity of an object. The best form of identity is that of a surrogate which is explicitly supported by the system. It is independent of the data in the object and maintains its representation after updates. The system also provides operations to manipulate the identities. The other dimension is the temporal dimension which rates the preservation of identity representation. The best form of identity in this dimension is one that maintains its representation between different sessions and after restructuring of data.

The objects modelling Fred and Jill each have a unique identity which we can use to identify them. The identity does not depend on social security number, name or any of the other data that is stored for them.

The identity of an object can be used to reference it, which allows a number of objects to share the same object as a component by each keeping a reference to it. The benefit is that we can determine if the two objects do in fact share an identical component or if they share components which have the same values. In the case where objects share the same component, update anomalies are eliminated. When the component in one object is updated, the update occurs in the shared object as well. In relational databases, however, the components have the same value and all updates which are made to the one component must be made to all the other components.

If Fred and Jill have the same child, then another object is created in the database to model Jack. Objects Fred and Jill each contain a reference to object Jack, by storing its identity. Any updates made to Jack through object Fred are clearly reflected through object Jill as well. In the other case, where there are two distinct children, each with same name, two objects are created in the database. Object Fred has a reference to the one object, while object Jill has a reference to the other object. By comparing the reference in object Fred to the reference in object Jill, we can determine if they have the same child.

2.2.1.2 Object State

The information associated with an object is stored in its state. The data is stored in a representation of some kind and according to some structure. Databases, programming languages and types provide three different perspectives of the structure which is used.

Attributes are similar to the attributes found in a record structure in programming languages or columns in relations. An attribute has a name and an associated type or value domain. The attribute name is used to identify or address the data stored in the attribute. The state of the object is a record consisting of a number of attributes.

The attributes are classified according to their value domains or types. *Value attributes* are "proper" attributes and are drawn from a value type or system-defined type. Fred's name may be stored in an attribute "name" which is defined over the string domain. These attributes are also known as atomic or simple attributes since they take on system-defined types such as integer, real, string, etc.

A *complex attribute* is made from simple attributes and other complex attributes using constructors, which together form its value domain or type. It allows attributes to be collected into a single logical unit and is closely related to complex objects.

A number of attributes called *aggregate attributes* are grouped or juxtaposed to form a single logical unit. Aggregation is the process of grouping these attributes together. Each of the attributes in a record is an aggregate attribute, while a complex attribute may have a record as its value [Osbo89].

A *reference attribute* is used to link one object to another object. It may be single-valued or set-valued. For example, Fred has a set-valued reference attribute "children" and a single-valued reference attribute "spouse".

Certain data associated with an object can be calculated from that which is stored. To store this data amounts to storing redundant and potentially inconsistent information. For example, to store Fred's age, in terms of years, months and days, is pointless since the day after the age is stored it will become inconsistent. We store the date of birth instead and calculate his exact age as needed. A *derived attribute* is declared in a similar manner to a function declaration. The input parameter is the object and the result is the derived attribute value. The derived attribute "age" calculates the age of a person based on their date of birth. The illusion that "age" is an attribute is created when the operation is passed as a message to the object. Attributes and derived attributes are handled uniformly.

The concepts of *fields* and *instance variables* arise from the object-oriented programming perspective. For each object, each piece of information is stored in a variable called an instance variable, which serves the same purpose as an attribute. A field is a set of instance variables defining the environment of an object. A field might be considered to be similar to a record [Coin87].

From the typing perspective we can define the state of an object by using a data structure which is built with various *constructors*. A record structure is defined by constructing it from a number of attributes and corresponding value domains. There are other constructors such as set, list, array and disjunction [LéRi89]. A disjunction is also known as a sum or union constructor.

According to [ABDDWZ90] set, list and tuple are the minimal set of constructors a system should have. Sets are an intuitive means of representing collections in the real world, a tuple represents the properties of an entity and a list represents ordering which occurs in the real world.

Ideally all constructors are orthogonal, that is, each constructor is independent of all other constructors and any constructor may be applied to any object [ABDDWZ90]. This generates a *complex*

object or structure, which in turn may have a constructor applied to it. The resulting complex objects may be of any form, unlike in the relational model where the tuple constructor may be applied only to simple objects and the set constructor may only be applied to tuple objects.

2.2.1.3 Referencing

In order for an object to perform one of its operations, a message is passed to it. The object to which the message is passed is referenced by a special variable called either "This" or "Self". From this object, one can access its references to other objects. A reference has an implicit direction associated with it: one can get from the referencing object to the referenced object, but not necessarily the other way around. If one object contains a reference to another object, this denotes some form of *relationship* between the two objects. Additional semantics may be attached to the reference depending upon the type of relationship.

> *"Relationships are one of the most fundamental parts of any data model. From one point of view, they are what distinguish databases from file systems." [ZdMa90]*

In Section 2.2.1.1 we discussed the concept of object sharing, where a number of objects each contain a reference to the component (object) which they share. Object sharing is often called aliasing in the programming-language world [ZdMa90].

One relationship is the *inverse* of another relationship if the first object has a reference to the second object and vice versa. For example, Fred has a children relationship which references Jack. If Jack has a parent relationship which references Fred, then the children relationship is the inverse of the parent relationship. *Symmetric relationships* are inherently bidirectional and are maintained by the system as opposed to inverse relationships which are maintained by the user. The spouse relationship between Fred and Jill should be defined as a symmetric one.

When an object has a reference to another object, this second object may be thought of as a property of the first object. Fred may have worked in the design department for the past five years, a fact which can be considered as one of his properties. An object in the database models the design department and a reference to it is specified from Fred.

A system should provide consistency and integrity of references. For example, a reference to an object may only exist if the object exists. If a referenced object is deleted, the reference is termed a *dangling reference* and generates inconsistency. Different systems provide different mechanisms for dealing with this problem.

One way is to delete all the references to an object when it is deleted. This is very costly because it involves finding all of the references in the database. Another way is to replace the deleted object with a *tombstone* which indicates that the object has been deleted. When an object reference is used and a tombstone is encountered, the reference is identified as dangling and is then removed. A *reference count* may be maintained by an object indicating the number of other objects that have references to it or the number of objects that share it. Only when the reference count is zero will the system allow

the object to be deleted. This mechanism transfers the responsibility of finding all of the references from the system to the user.

Another solution has no explicit deletion mechanism, but is based on the references. When all references to an object have been removed, the object becomes unreachable. *Garbage collection* is used by the system to identify and remove all unreachable objects.

The concepts of objects being reachable or unreachable is based on the transitive nature of references. Object *B* is reachable from object *A* if it is possible to navigate from object *A* to object *B*. If Jack has a child Jim, then Jim is reachable from Fred by first referencing Jack using Fred's children relationship and then referencing Jim by using Jack's children relationship. If it is not possible to navigate from object *A* to object *B*, then object *B* is termed *unreachable* from *A*.

2.2.1.4 Object Graph

An *object graph* is not a fundamental property of an object, but is a graphical means of denoting objects and their relationships. Objects are usually denoted by nodes, while relationships are denoted by labelled directed edges. An object graph is defined in [LRV88] which also captures the structure of complex objects. Basic objects are labelled nodes. Tuple objects comprise a node represented by a dot and, for each attribute, a labelled edge to the component object. A set object is denoted by a star with a directed edge to each of its elements.

Navigation is the process by which one moves around the object graph. This is achieved by moving along a reference from one object to another object. It is here that the implicit direction of references is important. One may be able to move from Fred to Jack because Fred has a reference to Jack due to the children relationship. But, if Jack does not have a reference to Fred, then we cannot navigate back from Jack to Fred.

Graphs can provide a visual means of conceptualizing the data in the database. They can be used for a graphical browser, where the user can view objects and navigate between them.

2.2.1.5 Composite Objects

Most of the work done on composite objects has been as part of the ORION system at MCC, and can be found in [BKKK87], [KBCGW87] and [KBG89].

A *composite object* is a collection of objects treated as a single object. The objects in the composition are related to each other by the IS-PART-OF relationship. Special semantics are defined by the system to deal with operations on composite objects. The IS-PART-OF relationship is required by some of the newer applications domains. The relationship models the components and structure of a complex entity in the real world, as well as the operations which are performed on the entity as a whole. The relationship is symmetric: the component references the composite object and clearly the composite object references its components. In [ZdMa90] this relationship is called the *contains relationship*.

In ORION special reference constructors are defined to create composite references with special semantics, viz. exclusive, shared, dependent and independent references. Operations are used to determine the components of a composite object at various levels. The converse operation is also defined to determine parent and ancestor composite objects. Special predicates are used to test if one object is a component or child of another object.

2.2.1.6 Object Operations

An object-oriented system supports various basic or primitive operations to manipulate objects. User-defined operations for objects are covered in the method section (Section 2.2.4). The operations presented here are described in [KhCo86] and [LRV88].

Two objects are *identical* if they have the same identity. Actually the two objects being compared are one and the same. This predicate is called *0-equality* in [LRV88]. To determine if Fred and Jill have the same child, the identical operation can be used to compare the child references. If the result is true, then there is only one person by the name of Jack and he is the child of both Fred and Jill.

Two objects are *shallow equal* if their values and identities are identical. There may be two distinct objects but their values are equal and the objects they reference are identical. This predicate is called *1-equality* in [LRV88]. In the case where Fred and Jill have distinct children, but each with the same values, the identical predicate will return false. The shallow-equal predicate will return true because the children have exactly the same properties.

The *deep equal* predicate is concerned with the values stored in objects and has no regard for their identities. Two simple objects are deep equal if their values are equal. Two complex objects are deep equal if each of their respective component objects are deep equal. Two components are deep equal if the objects which they reference are also deep equal. This operation is recursive and hinges on the comparisons between simple objects. [LRV88] calls this predicate *value equal* or ω-*equality*. The deep-equal predicate can be used to compare two cars and determine if they are the same model, with the same size engine and colour. Each car has its own components which may be the same. None of the components in either of the two cars are shared, thus there are no shared objects. If all of the corresponding components in the two cars are the same, then the two cars are the same.

The following relationship exists between the above three predicates. Two objects which are identical are also shallow equal, and two objects which are shallow equal are also deep equal.

The create or new operation is used on a class and produces a new object whose structure is defined by the class. Once created, the object is assigned a unique identity.

Sets are collections of unique entities. Because of the different forms of equality, as defined above, there are a number of ways in which the system can support sets. The uniqueness of the elements in a set can be based on one of the three predicates. If the identical predicate is used, then when an object is to be inserted into a set and it is identical to another element of the set, it is not added. If, however, the object is shallow or deep equal to another object, it is still added. A set contains unique objects, although some of them may be shallow or deep equal. The *value eliminate* operation produces

a new set in which no two objects are deep equal. There is also a *remove* operation, which is used to remove a given object from a set, according to its identity.

It is possible that two distinct objects may be created and then at a later stage it is discovered that these two objects are in fact the same object. The *merge* operation will take the two objects and produce a new object by merging their identities. The operation needs to take into account all references to the original two objects as well as the structure and values used in the new object.

Identity assignment is a simple operation used to assign a reference from one object to another. It involves storing the identity of the referenced object in one of the referencing object's attributes. This operation is used to specify a relationship between two objects, such as assigning Jack to Fred's children attribute.

Given an object, the *shallow copy* operation will produce a new object which is shallow equal to the given object, but not identical to it. The shallow copy operation is also used to copy the values of one object to another object, in which case no new object is created.

Given an object, the deep copy operation will produce a new object which is deep equal to the given object, but not necessarily shallow equal to it. The operation creates a new object for the given object and new objects for each of the objects which are either directly or indirectly referenced by it. The values from the original objects are then copied to the corresponding new objects. In terms of the object graph, the operation duplicates the subgraph which is rooted at the given object.

2.2.2 Classes and Types

The general purpose of a class or type is to define a group of objects and the operations which can be performed on them. The schema and application programs in a traditional database are replaced by a set of classes and/or types.

A class or type is a generalization of a set of objects. The concept of generalization originated in programming languages as abstract data types and is now being applied to objects. Assume the structures of objects Fred, Jill, Jack, etc. are all the same. The operations which are defined for these objects are also the same. A class or type is used to define the state structure of a person object and all of the operations which may be performed on it. A class or type Person represents the generalized person object. All person objects make use of the definitions and operations which are defined in the class or type.

An *intentional specification* defines all possible objects for a structure. Types and classes may contain an intentional specification or template for defining their objects. Also associated with a class or type is its *extension*, which is the set of objects that are currently its instances.

The distinction between classes and types is based on their difference in origin and use in various systems. A number of definitions for classes and types are provided in the literature. The definitions range from saying that they are the same, to stating that they are totally distinct. Some systems provide only one feature while others provide both. In the next two sections we offer definitions for each.

2.2.2.1 Class

The term class originated in Smalltalk according to [ABDDMZ90]. A class is defined in [MCB89] as a conceptual classification of the real-world; concepts in the real world are modelled by classes in the database. Other definitions appear in [ABDDMZ90], [LéRi89], [Masu90] and [ZdMa90], for example.

> *"A class is a template (cookie cutter) from which objects may be created by 'create' or 'new' operations. Objects of the same class have common operations and therefore uniform behaviour. Classes have one or more 'interfaces' that specify the operations accessible to clients through that interface. A 'class body' specifies code for implementing operations in the class interface." [Wegn87]*

A class factors out the common structure and behaviour (see Section 2.2.4) of a collection of objects. A class is more of a run-time notion, while a type is a compile time notion. A class consists of an object factory and an object warehouse. The object factory is used to produce new objects, while the object warehouse contains the extension of the class. A class is generally a first-class object and can be manipulated at run-time. This provides the system with increased flexibility and uniformity, but renders compile-time type checking impossible. Run-time type checking, however, can still be implemented [ABDDMZ90].

2.2.2.2 Type

The concept of a type has been defined in [ABDDMZ90], [AtBu87], [DaTo88], [LRV88], [MCB89], [Wegn87] and [ZdMa90]. Types stem from programming languages in which they are used to check the correctness of operations.

According to [DaTo88] a type can be viewed as a constraint, which provides a logical approach to types. The algebraic approach to types views them as sets with algebraic operations on their elements.

A type specifies a set of values and the operations for manipulating those values. In this context, the values are generally objects. The objects all have the same characteristics and are defined by the same data structure. A type consists of an interface and an implementation. An *interface* contains the operations which can be used to manipulate the objects and their signatures. A *signature* consists of the type of each input parameter and the type of the result, this information being used by the type checking process. The interface is used to encapsulate the objects by ensuring that only those operations which appear in the interface can be used by the user to manipulate the objects. In other words, the implementation and structure of the objects is hidden from the user of the type. Types defined in this context closely resemble Abstract Data Types used extensively in programming languages.

A type's *implementation* consists of a data part and an operation part. The data part specifies the structure of the object's data (its state), while the operation part contains the implementation of the operations specified in the interface. The implementation is only visible to the type implementer. Simula67 was one of the first languages to use a type in this context [ABDDMZ90].

When a system makes use of both types and classes, types refer to the intentional specifications of objects, whereas classes refer to the extension of the corresponding types. Alternatively, the class

may be the intension (template) and the type merely the interface. In the case where another mechanism exists for defining instances, a type need not include an implementation. In higher-order models types and functions may be viewed as values.

Typing emanates from programming languages where it is used to increase programmer productivity, by ensuring the validity of operations. In an OODB, typing is associated with types, methods, messages and objects. It is used to check the compatibility of data structures and operations, objects and messages and types and subtypes. Typing is defined by a type system consisting of a set of typing rules. These rules are used to compare and infer typing information.

The type checking process makes use of these rules to check the values and expressions for correctness. A run-time error may leave the OODB in an inconsistent state because the operation has only been partially completed. The recovery mechanism is then required to place the OODB back in a consistent state. Typing tries to ensure that during the execution of an operation (method), a run-time error will not occur.

A system may provide various data types (structured types or typing structures) such as records, products, lists, sets, arrays, functions, etc. These structures are defined by constructors which are used to build them from other data types. The system is *data type complete* if any set of data types can be used as a component in the construction of any other data type. The concept of data types also originated in programming languages and is used to group and structure values into a logical unit. In the object state section earlier, we discussed how these structures can be used to define the state structure of an object. These data structures are closely linked to the typing system. There are typing rules defined for each constructor which allow the type checker to reason about the compatibility of data types.

2.2.2.3 Schema Evolution

The set of classes and/or types defines the "schema" for an OODB. The concept of schema evolution corresponds to the modification of the set of classes and types. Typical operations involve adding or removing classes or types, changing their implementations and changing their interfaces. These operations are complex and have far reaching effects on the entire system. The way in which these changes are implemented depends very much on the system (see [BKKK87] and [Osbo89] for two such examples).

Once the schema change has been made, the system should be in a consistent state. This is closely linked to the type checking process and is dealt with in more detail there.

2.2.2.4 Triggers

A *trigger*, also known as an *alert* or a *monitor*, is used by the system to monitor the state of an object. When the state matches the trigger's condition, it fires and executes a set of instructions. As an example, if the stock level drops below a fixed point, new stock must be ordered. A trigger can be set on an object to keep a check on the stock level: when it does drop below a certain level (trigger condition), the user is informed (instructions).

2.2.2.5 Constraints

> *"Integrity constraints are predicates on the state of the database that the database is responsible for enforcing. They typically are defined in the schema relative to an entity or over a collection." [ZdMa90]*

A constraint is used to maintain a higher level of consistency in a system than that imposed by the typing system. A constraint is specified for an object's state which evaluates to a boolean value. It is monitored and maintained by the system. In the case of an update violating a constraint (result evaluates to false), the update is aborted and the object's state is returned to its original value [AgGe89].

An attribute such as Age may be of type integer, but the only acceptable values for the attribute are between 0 and 150. A value not within this range would be unacceptable or impossible, e.g. -4. The system monitors all updates made to the Age attribute by testing the condition specified in the constraint. If the conditions are satisfied, then the update is valid.

2.2.3 Inheritance and Subtyping

The concepts of inheritance and subtyping are discussed in [ABDDMZ90], [LéRi89], [Stei87], [Wegn87], [ZdMa90] and [ZhRo88]. Related issues such as delegation [Wegn87], and contradictions and exceptions [Borg88/89] are not considered below. A class is used by the database to model concepts from the real world. Inheritance is used to model the specialization and generalization relationships that exist between these concepts. Inheritance allows one class to inherit the operations (and definitions) of another class. Inheritance is typically associated with classes, while subtyping is typically associated with types.

A class or type which is being inherited from is called a *superclass* or *supertype* respectively. The class or type which inherits the operations is called the *subclass* or *subtype* respectively. The class from which an object is created is called its *base class*. The terms *superclass* and *supertype* refer to the immediate superclasses and supertypes and also to their ancestors. An object may make use of operations which are defined in its base class or in its superclasses.

[ABDDMZ90] identify four forms of inheritance: substitution, inclusion, constraint and specialization.

In substitution inheritance, we say that a type T inherits from a type S, if we can perform more operations on objects of type T than on objects of type S. This means that an object of type T may be substituted into any context where an object of type S is expected.

Inclusion inheritance corresponds to the notion of classification. It states that type T is a subtype of S, if every object of type T is also an object of type S. This form of inheritance, also known as subtyping, is based on the structure of the objects only. The operations that are defined in the type can be passed to objects defined by the subtypes because the subtyping ensures that the operations are still well defined.

Constraint inheritance is a subcase of inclusion inheritance. A type *T* is a subtype of a type *S*, if it consists of all objects of type *S* which satisfy a given constraint. This form of inheritance is used where facilities provided by inclusion inheritance are insufficient. An integer value may be used in *S*, but in *T* it is required to fall within a specific range of integers. The constraint is used to identify a set of objects which forms a subset of the set of objects defined by the type.

With specialization inheritance, a type *T* is a subtype of a type *S* if objects of type *T* are also objects of type *S* with more specific information. This form of inheritance has the properties of inclusion inheritance, but also defines the specialization of concepts being modelled in the real world.

"Inheritance has two advantages: it is a powerful modelling tool, because it gives a concise and precise description of the world and it helps in factoring out shared specification and implementations in applications." [ABDDMZ90]

Inheritance allows for code sharing and the sharing of some of the class and/or type definitions. This reduces the amount of work required by the implementer, since he/she can reuse the work which has already been completed. It also provides a framework which classifies and relates the different modules in the system. The system is thus simpler to understand and maintain.

Inheritance is an intuitive mechanism for the user to use when modelling concepts in the real world. Some of the more difficult modelling problems in traditional databases have simple elegant solutions as a result of inheritance.

2.2.3.1 Multiple Inheritance

Multiple inheritance occurs when a concept is a specialization of two or more distinct concepts or, conversely, when a concept can be generalized into two or more distinct concepts. In *single inheritance* (the kind dealt with so far) a class has a single superclass. In multiple inheritance a class may have multiple superclasses, from which it may inherit all of the operations.

Associated with multiple inheritance is a potential *name conflict*. This occurs when the same attribute or operation name is used in two or more of the superclasses, but each one models a different idea. There is a conflict because the subclass inherits all of the attributes from each of its superclasses. The resulting record consists of a number of attributes with the same name, which is invalid. A name conflict also arises when two or more superclasses define an operation with the same name. Since an object may make use of operations defined in its superclasses and because they are identified by name, it is not clear which one of the operations is being selected. A process called *conflict resolution* is used to deal with these problems. This usually requires the schema designer to specify which attribute or operation is to be inherited, or to rename those which are inherited.

2.2.3.2 Subtyping

Subtyping is defined by a set of rules specifying when one data type is a subtype of another. These rules are used in inheritance to ensure that the state structure of a subtype is compatible with the state structure of the type. This allows values defined by a subtype to be substituted for values defined by

the type. Subtyping is the necessary condition for inclusion polymorphisms — operations defined in a type which may be applied to instances of a subtype (see Section 2.2.4.3).

Subtyping rules can be specified for a range, record, product, function, variant, enumeration, etc. For example, the subtyping rule for a record specifies that all of the attributes defined in the type must be defined in the subtype. Each attribute type in the subtype must be a subtype of the corresponding attribute type in the type. The subtype may define additional attributes. For example, the state structure of Fred, a Person, is defined by a record structure with attributes name, date of birth and children. The Student type has a state structure defined by a record structure with attributes name, date of birth, children and courses, thereby qualifying as a subtype of Person. Subtyping rules are defined in detail in [CaWe85].

2.2.3.3 Hierarchies

The inheritance between classes may be depicted by a directed acyclic graph (DAG). A class is a node in the graph and a directed edge is drawn from a class to each of its superclasses. The graph indicates from where each class inherits operations. The DAG is drawn as a hierarchy, with the edges pointing upwards. On top are the most general classes and at the bottom are the most specialized classes. With single inheritance the hierarchy forms a tree. With multiple inheritance the tree property does not hold, since a node (class) may have a number of edges radiating from it to other nodes (superclasses). The hierarchy is also called the is-a hierarchy and depicts a classification of concepts.

[ZdMa90] define specification, implementation and classification hierarchies which are basically related to inheritance. The specification hierarchy consists of types and the subtyping relationships between them. The implementation hierarchy provides for code sharing among types. The classification hierarchy describes collections of objects and the containment relationship among these collections.

2.2.4 Methods

An object may be manipulated by a number of operations which are defined in the interface of its class. A *message* is a request to an object, also called the *receiver*, to carry out one of the operations defined for its class. This process is called *message passing*. The message specifies which operation is required, but not how it is implemented. Provided the types of the parameters match the signatures, the object will perform the operation successfully. A *method* describes how the operation is implemented and is specified in a method language. A method is defined for a specific object (or class of objects) and has access to the internal structure of the object. The method forms part of the implementation of a class and is hidden from the user. The checking of parameters against the signature forms part of the type checking process (see Section 2.2.6).

2.2.4.1 Functions and Procedures

Sometimes the operations defined for an object are referred to as functions or procedures. These are the same concepts as found in programming languages. A derived attribute is an example of a function which performs a computation, returning a result. The Age function is specified for class People. It is applied to a person object, performs the calculation and returns the age of the person.

A procedure does not return any results, but has side-effects. A procedure could be thought of as a stored set of commands or as a transaction. The declaration of a procedure consists of a name, a typed argument list and a list of actions.

2.2.4.2 Computational Completeness

In a traditional database the information is queried using a query language, e.g. SQL. More complicated operations require an application programmer to write code in a programming language with embedded database operations. This code is stored in the traditional file system and not in the database. There is a separation between data and operations, in effect requiring two models.

Computational completeness means that one can express any computable function in the language provided by the system. As discussed earlier one of the major problems of databases is the lack of computational completeness and the associated impedance mismatch problem.

OODBs have method languages which resemble programming languages and are usually computationally complete. This solves the impedance mismatch problem. Each method is stored in an object's class, with the result that there is only one model and both data and operations are stored in the database [ABDDMZ90].

2.2.4.3 Morphisms

The following definitions are from [CaWe85]. A *morphism* is used to describe the typing nature of an operation. A *monomorphism* is an operation which operates on operands of only one type. For example, integer addition is a monomorphism, since it only operates on two integer values and returns an integer result. The same operation cannot be used for adding two real numbers. On the other hand, a *polymorphism* is an operation where the operands are not constrained to a specific type. For example, the size operation for a list may be applied to any list and will return the number of elements in the list.

Polymorphisms are split into two categories, universal polymorphisms and ad-hoc polymorphisms. *Universal polymorphisms*, also known as *true polymorphisms*, are operations which execute exactly the same set of instructions regardless of the operand types. Universal polymorphisms are themselves split into parametric polymorphisms and inclusion polymorphisms.

A *parametric polymorphism* is an operation which operates uniformly on a range of types which normally share a common structure. The size operation for a list is an example: the list is the common structure required by the operation, but the values in the list may be of any type. [ZdMa90] call these operations *general polymorphisms*. A *generic operation* is a parametric polymorphism which requires a type parameter.

An *inclusion polymorphism* is an operation that operates uniformly on a set of types which are related by the subtyping relationship. Any data types may be used (records, sets, list, products, functions, etc.), provided they are related by subtyping. This kind of operation is essential to an OODB. It allows an operation such as Age, which is defined in the Person type, to be applied to objects defined by a subtype such as Student. In [ZdMa90] the concept of an *extension polymorphism* is defined. This is similar to an inclusion polymorphism, but only applies to operations which are defined for record structures.

Ad-hoc polymorphisms are operations which operate on a range of types in unrelated ways. For each type, a separate piece of code is written. By examining the types of operands, the correct piece of code can be selected.

Overloading, a form of ad-hoc polymorphism, is where totally distinct operations share the same name. An operation name is said to be overloaded, if there is more that one operation with the same name. By examining the context in which the operation is used, the type checker can determine the types of the operands and select the appropriate operation. For example, a Print operation can be written for people and cars. When the Print operation is applied to Fred, the Print operation defined for people is used, since Fred is a person.

Overriding [ABDDMZ90] is related to overloading but is where a new implementation is defined in a subtype for an operation specified in the type. The implementation in the type is overridden by the implementation in the subtype. When the operation is used for a subtype object, the implementation in the subtype is used.

Coercion, another form of ad-hoc polymorphism, is an implicit translation of operands to the types required by an operation. Since it is implicit, the user is given the impression that the same operation can be applied to values of different types. For example, the addition operation is defined for real numbers and integers. When two integers are added together, the integer operation is used. When two real numbers are added together, the real operation is used. When a real number is added to an integer, the integer is first translated into a real value (such that is still denotes the same numeric value) after which the real operation is used.

A related concept is that of *value sharing*, which is when the same value is used for different types to denote the same semantic concept. For example, the nil value is used for references to denote that no object is being referenced. It appears as though the nil value is defined for each type, but actually it is shared. Value sharing can be thought of as a constant parametric polymorphism.

2.2.4.4 Binding and Dispatching

The process of associating an overloaded name with a specific implementation is called *binding* (or dispatching [ZdMa90]) and normally occurs at compile time. If the association occurs dynamically (during run-time), this is called *late binding*. Although the same number of implementations are written for the different types, the user does not need to be concerned about which implementation to use. When a new subtype is added, new implementations might be written in the subtype for operations

specified in the supertype (overriding). Depending on the form of inheritance being used, even if no new implementation is defined for the new subtype, implementations in the supertypes may be used in its place.

Simple dispatching occurs in a model with single inheritance, where the implementation (method) of the operation is selected from the supertype which is the closest to the receiver's type. In the case of multiple inheritance, there may be a number of candidate methods, so a search criterion is required to select the appropriate one. Dispatching is usually based on the typing of the receiver. Multi-argument dispatching uses the types for a number of arguments in an operation as the search criterion when locating the appropriate implementation of the operation.

2.2.5 Encapsulation

The feature of encapsulation is dealt with by [ABDDMZ90], [DaTo88], [LéRi89], [Wegn87] and [ZdMa90]

Encapsulation allows us to view classes and types as abstract data types (ADTs), where the implementation of the class or type is hidden. Encapsulation is a software engineering technique that provides a sharp distinction between specification and implementation. This concept is new to the database field and believed to be crucial [ZdMa90].

From the database perspective, an object encapsulates both program and data. It is not clear whether the structure of the object is part of the interface or the implementation, unlike programming languages where it is clearly part of the implementation.

Encapsulation also provides modularity for a system. Modularity is required to structure complex applications for both design and implementation purposes. Since encapsulation hides the details of an object, a program may only use the object by using the operations provided in the interface. The implementation and data structures used for an object can be changed, without it having any effect on programs which make use of the object or the rest of the system. Clearly a system with this feature has reduced maintenance costs, because all changes to an object's implementation and the effect of those changes are localized.

According to [ABDDMZ90] there are cases where encapsulation can be violated without reducing the maintainability of the system. In special cases such as ad-hoc queries and browsing, access to the internal structure of an object simplifies the process of obtaining the required data quickly. Since the operation is ad-hoc, it is not stored and hence will not suffer from the maintenance problems associated with modification to the object's implementation.

Models which do not provide a strong view of encapsulation are called *structurally object-orientated*. The structure of the data in the implementation is visible as is the case in semantic data models [ZdMa90].

2.2.6 Type Checking and Inference

When a user declares the type of all variables used in a program, the system can reason about the syntatic correctness of the program based on this information. "Thus types are mainly used at compile time to check the correctness of the programs." [ABDDMZ90]

If type checking occurs during compilation, it is called *static type checking*. If the type checks only occur when the program in running, then it is called *dynamic type checking*. A system is *strongly typed* if it is possible to determine the type of all values and expressions such that it will execute without type errors. *Type inference* is the process whereby the type of an expression is determined from the arguments of the expression, by making use of the typing rules.

Since type checking originated in programming languages, the concepts used there need to be refined to deal with differences found in OODBs. Instead of compiling a program and performing the type checks on its contents, the type checking should be performed over the entire system to determine its correctness. Type checking is required to ensure:

- The operations which are applied to objects in a message pass are defined in its class or one of its superclasses.
- Correctness of operations and expressions in a method.
- Access to the internal representation of an object is only permissible by methods which are defined for that class of object.
- Correct subtyping between types or classes.
- If the schema is modified by adding, deleting, or modifying a type or its implementation, then the system should be in a correct state after these changes.

2.2.7 Metatypes and Metaclasses

The concept of metaclass or metatype is dealt with in [ABDDMZ90], [DaTo88], [ZdMa90] and extensively in [Coin87].

Usually, objects are first class values, while types or classes and operations are considered higher-order values or non-first class values. The treating of classes or types as first class values gives rise to the concept of a metaclass or metatype.

An object is an instance of a class. If we allow a class to be treated as an object, then it too is an instance of a class called a *metaclass*, or in the case of a type, a *metatype*. In a system without metaclasses, there is a clear division between objects and classes. The system provides all of the operations for manipulating classes, while objects are manipulated by both system-defined primitive operations and user-defined methods. In a system with metatypes/metaclasses, although the division between objects and classes does exist, it is not of great importance. We may define messages which we can pass to classes in the same way that we pass messages to objects. The operations provided by

the system to create, update or delete classes can now be replaced by messages which modify the structure of classes dynamically. This provides a uniform treatment of objects and classes.

The mechanism which produces an instance from a class is called *instantiation*. Metaclasses give rise to an instantiation hierarchy. At the bottom level are the objects and above them are their classes. Above each class is its metaclass. The hierarchy displays the relationships between objects and their classes or types.

We have defined a database containing people, with class Person defining a person. We now define a metaclass Person-Occupation, which defines various classes of people with occupations. Each of these classes defines a person with that occupation, such as employees and students. Each of these classes also inherit class Person which defines any person. In Person-Occupation we define a number of attributes relating to the occupation of a person. The Employee class is an instance of Person-Occupation and contains all of the general information relating to the occupation of an employee. All of the Employee objects share the information relating to their occupation, which is stored in their class, but the information which they store is defined by class Person. The advantage is that the information relating to the occupation is stored only once, in the class (variables) and not in each object. Person-Occupation can be thought of as a generic class, where its instances are specific cases of the generic class.

2.2.8 Extensibility

According to [ZdMa90], one of the goals for OODBs is to provide a system with an extensible set of basic modelling primitives. The requirements of a new application are not all known initially, the complex data structures being defined only over time. The system must be able to evolve to meet these requirements. Extensibility provides the system with the facilities to achieve this evolution. It is important that "there is no distinction in usage between system defined and user defined types." [ABDDMZ90]

The system contains a set of predefined types such as integer and string, and it also provides constructors for creating new data structures. These data structures may be used by a user to define new types. Each new type is defined with a set of operations and is encapsulated. Like the system-defined types, the instances of a user-defined type may only be manipulated by the set of operations defined in the type's interface. Since there is uniformity in use between the system types and the user-defined types, a new type shares the same status as any other type in the system.

2.2.9 Other Features

In this section, we briefly describe a number of other features of object-oriented databases. These are queries, persistence, concurrency, consistency, storage management, multiple users, distributed databases, versions and histories.

A *query* facility, which allows a user to ask simple questions regarding the data in the database, should be supplied by the system. It may take the form of a graphical browsing tool or a query language. The query facility is used for data retrieval operations and not for general purpose programming. The means of querying should be independent of the application, that is, no additional operations should be written for any application in order to query it contents.

Persistence — the ability of data to survive from one session to the next — is a fundamental feature of any database. This feature is however quite new in the programming language field, although a number of persistent programming languages exist. In OODBs, this feature naturally occurs and ideally is orthogonal to all other features, i.e. persistence is a property which is associated with objects and not with a class which results in all of its objects being persistent [ZdMa90].

Concurrency is a feature that allows a number of processes to access the database simultaneously. The system ensures that integrity is maintained when data is accessed and manipulated by using transactions and some form of concurrency control.

A database is *consistent* if the data stored therein is accurate and meaningful in terms of its modelling of the real world. A database will at some stage crash and transactions will quite often be forced to abort. The database is *robust* if it contains mechanisms to ensure that, in such an event, it can recover to a consistent state.

Secondary storage management is the way in which a database system efficiently manages the large amount of data on disk. It involves techniques such as indexing, clustering, buffering and optimization. The nature of objects can be exploited for efficient storage management by using their identities to retrieve and index them. When using the system, we often perform queries and operations over a set of objects from the same class. As a result, some systems cluster and buffer objects which are instances of the same class, which improves efficiency where these types of operations are frequent. Other systems cluster objects which form part of the same composite object [KBBCGW88].

A database may support a number of identified users, where the main task of the system is to provide some form of security for their data. This is achieved by issuing each user with an authorization which determines to which data he/she has access. In [KBG89], various forms of authorization are identified.

Networks have allowed databases to become decentralized. A database may be spread over a number of machines which may be locally or widely dispersed. To the user of a distributed database, the network is transparent in that all data is directly available from the machine he/she is working on (there is no distinction between local and remote data). The database systems comprising the network are required to retrieve remote objects efficiently and to cope with concurrent access to objects.

Databases are now being used in the design domain, applications making use of CAD/CAM and CASE being typical examples. The design process is an evolutionary one, where the design undergoes numerous modifications until the best design has been achieved. Some OODBs, such as ORION [KBCGW87] and O++ [AgGe89] provide *version* mechanisms which maintain all of the various designs used in the evolution of the current design.

[CoMa84] discuss the concept of adding history to an object in the GemStone project. A database models the real world, but only provides us with a snap-shot of the real world at a particular instance in time. A *history* is used to store data relative to time. When an object is created a timestamp is assigned to it. Each time the object is modified, the old version is stored and the new version is time stamped. When the object is deleted, a second timestamp is assigned to it, indicating that the object no longer exists at the present time.

2.3 Conformance

A number of authors have proposed sets of features which define an OODB. In this section we discuss some of these proposals. Any system wishing to be termed a database, must provide for the management of large amounts of persistent, reliable and shared data [Banc88].

2.3.1 Manifesto

An OODB manifesto has been defined in [ABDDMZ90] in which mandatory and optional features for an OODB are specified. An OODB should be both a DBMS and an object-oriented system. To be a DBMS it should support persistence, secondary storage management, concurrency, recovery and ad-hoc queries. To be an object-oriented system it should support complex objects, object identity, encapsulation, types or classes, inheritance, overriding and late binding, extensibility and computational completeness. The optional features include: multiple inheritance, type checking, distribution, design transactions and versions.

2.3.2 Zdonik and Maier

Zdonik and Maier have published a collection of readings in Object-Oriented databases. In the first chapter they provide their own perspective on OODBs [ZhMa90].

They identify essential features for a database along with other features which occur more or less frequently. The essential features required for a database (DBMS) are: a model and language, relationships, permanence and arbitrary size. The frequent features include: integrity constraints, authorization, querying, separate schema and views. The less frequent features include: report and form management, data dictionaries and distribution.

The minimum requirement for an object-oriented database are specified in a threshold model. Firstly, it must provide database functionality as described above. Secondly, the object-oriented features of object identity, encapsulation and complex objects must be supported. Inheritance has been omitted because they state that the term has been used in many different ways and it can be simulated with other language features.

The reference model defines the features which a typical commercial OODB should contain. The threshold model forms part of the reference model which adds the following features: structured representations for objects, persistence by reachability, typing of objects and variables, hierarchies, polymorphism, collections, name spaces, queries and indexes, relationships and versions. The following features are identified as being useful additional features: parameterized types, schema updates, active and derived data, integrity constraints, transactions and distributed data.

2.3.3 Wegner

In [Wegn87], Peter Wegner defines the dimensions of object-based languages and provides definitions for a number of the terms. A language is *object-based* if it supports objects as a language feature. An object-based language is *class-based* if every object has a class. A class-based language is *object-oriented* if, in addition to classes, it supports classes and class hierarchies, which may be incrementally defined by an inheritance mechanism.

2.3.4 Various Systems

A number of systems have been developed either as prototypes or as commercial products. In this section we briefly review the features of some of them. The features defined in these systems have been used to define the features discussed in this chapter. There are a number of other systems which we do not cover here, such as Iris [FABC89], CLASSIC [BBMR89] and OZ+ [WeLo89]. We do not cover ObjVLisp [Coin87] and FUN [CaWe85], since these are languages and not database systems. We also do not consider Postgres [StKe91] and Starburst [LLPS91], since these are extended relational systems, rather than OODBs.

2.3.4.1 EXODUS

The EXODUS database, presented in [CDV88], is a toolkit. The EXTRA data model contains instances and types, and a database is a collection of named persistent objects. The type system is based on multiple inheritance. It provides a data type complete set of constructors, viz. tuple, set, fixed length array and variable length array constructors. The "ref", "own" and "own ref" constructors are also defined to provide a limited form of composite object. They however call them complex objects, since they can be treated as either a composite or complex object. An "own" attribute specifies that the attribute contains a value and not the identity of an object. A "ref" attribute specifies a reference to another object. An "own ref" attribute specifies that the referencing object owns the referenced object. Support is also provided for abstract data types (ADTs).

The base types may be extended with ADTs. The system supports types, functions, procedures and generic set operations for any type.

The EXCESS query language is based on QUEL and has features added from POSTGRES and SQL. It has a uniform treatment of sets and arrays, including nested sets. There is a clean, consistent approach to aggregates and aggregate functions. The update syntax supports the construction of complex objects with shared subobjects. According to [CDV88], there is no mention of recovery or concurrency mechanisms.

2.3.4.2 F-Logic

Frame Logic, presented in [KiLa89], has been developed in order to reason about objects, inheritance and schema. F-Logic is a full-fledged logic which appears to be higher order but has first-order semantics and is tractable. It has complete computation power and can be used for database specification.

The syntax consists of basic objects, object constructors, variables and logical connectives. An object is either a class of instances or just an instance, there being no distinction between a class and an object. All objects are typed, although this is not a main feature of the language. Each object also has an identity. The relationships between objects are represented by a lattice. The lattice not only represents the "subclass of" relationship (inheritance), but also represents the "instance of" relationship.

A database is defined as a set of formulae, while a query is an F-term. Inheritance and methods can be specified in the logic. No mention is made of how persistence is achieved and what form of storage management is used [KiLa89]. The system does not appear to support recovery and concurrency.

2.3.4.3 GemStone

The GemStone system is being developed by the Servio Logic Corporation and is based on the language Smalltalk [CoMa84]. This system provides support for objects, classes, methods and messages. A limited form of query is expressible over collections of objects. The system provides concurrency control by means of transactions and the recovery mechanism makes use of shadow copies. The system also supports distribution, multiple users and histories [BOS91].

2.3.4.4 ODE

ODE (Object Database and Environment) is defined in [AgGe89] together with its programming language O++. The language has been derived from C++, hence the name. The system supports objects, classes and multiple inheritance. Each object has an identity and may either be volatile or persistent. The system clusters all objects of the same type. The system supports queries over sets and clusters, persistence, versioning, constraints and triggers. No mention is made in [AgGe89] of any form of recovery mechanism.

2.3.4.5 O_2

The Altaïr group have designed an object-oriented database called O_2, the formal data model being presented in [LRV88] and [LéRi89]. More detail on O_2 is found in [BCD89] and [Deux91]. The main objectives of the system are to increase productivity for application development, to provide better tools, and to improve the quality of the final product. O_2 is based on the ideas of merging programming language, user interface, database and object-oriented technologies. O_2 has a query language called O_2Query, an interface generator O_2Look and an object language O_2C. The data model supports values and objects, types, identity, subtyping, methods and classes. A value has a type, while an object has a value, an identity and a class. The system also supports encapsulation, modularity and extensibility. Persistence is based on reachability from a declared persistent root. Concurrency is implemented with transactions and two phase locking. The recovery mechanism makes use of a write-ahead log.

2.3.4.6 O^2FDL

[MCB89] have developed the Object-Oriented Functional Data Language (O^2FDL). The main goals of this language are to combine object-oriented and functional programming paradigms and to provide a notation for both databases and general purpose programming.

O^2FDL supports objects, classes, inheritance and messages. The expressions of O^2FDL are strongly typed and functional in nature. The type system is based on the FUN type language and the Milner type algorithm [Miln78]. Parametric and inclusion polymorphisms are supported as well as type inference. O^2FDL is a higher order language since functions are denotable values and may be passed like any other value.

To cater for general purpose and database programming, O^2FDL features three different types of functions (user-defined, data-defined and system-defined). A more flexible path expression notation has been used for function compositions compared to the nesting of function calls. The language caters for persistent and temporary objects and allows them to be handled in a uniform manner. In O^2FDL both classes and types are supported.

O^2FDL is more a persistent programming language system than a database system, since no mention of storage management, concurrency, recovery or ad-hoc query features is made in [MCB89/90].

2.3.4.7 ORION

The ORION project was launched at the end of 1985 at MCC and is detailed in [KBCGW88], [KCB88] and [KGBW90]. The object is to integrate a programming language system with a database system. The system supports objects, classes, inheritance, identity, versions, composite objects, transactions, authorization and queries, to name but the main ones. The system takes full advantage of composite objects for the definition and efficiency of versions, concurrency, authorization and schema modifications.

2.3.4.8 ObjectStore

The ObjectStore system [LLOW91] adds persistence to C++, since this is its main interface. The system provides objects, classes and inheritance. ObjectStore provides its own C++ language which allows queries. Collections of objects resemble tables or arrays and the language provides constructs to iterate over them. Type checking for both persistent and transient data is provided by the system. Transactions and read and write locks are provided to protect the integrity of the database.

2.4 ANSI Standard

Object-Oriented Databases or Object Databases which form part of Object Data Management systems (ODM), have become prime candidates for standardization. In this section we deal with the effort being made by ANSI for an ODM standard. We cover the committee which was established to perform this task and their goals and objectives. A number of surveys and workshops were conducted to formulate their recommendations.

The standardization of Object-Oriented Databases (OODBs) has been undertaken by a committee called the Object-Oriented Database Task Group (OODBTG). It operates as part of the Database Systems Study Group (DBSSG), which is an advisory group to Accredited Standards Committee X3 (ASC/X3). It in turn forms part of the Standards Planning and Requirements Committee (SPARC) which operates under the procedures of the American National Standards Institute (ANSI).

The task group was established in January 1989 for a period of two years. Its final task was to draw up a report containing a reference model and glossary, which was published in September 1991. After public comment, the final report was delivered in January 1992.

The goals and objectives of OODBTG were to establish a working definition for the term "Object Database", determine the relationships between Object Database technology and Object-Oriented technology in related fields, and establish a framework for future standards activities in the Object Information Management (OIM) area. The primary deliverables of the committee are an ODM reference model and a prescriptive glossary.

2.4.1 Reference Model

The Reference Model is used to provide prescriptive definitions of ODM terms. It provides a common language for communication and can be used for conformance testing of systems. The model is divided in to three sections: General Characteristics, Data Management Characteristics and ODM System Characteristics.

The general characteristics include the following: objects, bindings and polymorphisms, encapsulation, identity, types and classes, inheritance and delegation, object relationships, attributes, literals, containment, aggregates, extensibility, integrity constraints, and an object language.

The data management characteristics include the following: persistence, concurrency, distribution, ODM object language and queries, data dictionary and name space, change management, reliability and recovery, and security.

The ODM System Characteristics include the following: class libraries, program and user interfaces, and user roles [ANSI91].

2.5 Summary

OODBs seem to hold a number of solutions to the problems which are occurring in the database and programming language fields, since they cater for data and programs in one uniform model. Unfortunately, there is no accepted standard for the paradigm which has lead to the conceptual and semantic discrepancies which are found between current systems. We have discussed the concepts of objects, classes/types, inheritance/subtyping, extensibility and persistence which are essential to any system. We have also covered other features which are found in OODBs, such as: computational completeness, encapsulation, typing, metatypes/metaclasses, queries, concurrency, reliability and recovery, storage management, multiple users, distributed systems, versions and histories.

Chapter 3 System Overview

In this chapter we provide an overview of our data model and its implementation (more detail is given in Chapter 4 and 5 respectively). The background chapter provides a picture of the OODB paradigm. In the light of this we correlate the features defined in the literature with the features found in our system. We begin by identifying the goals of the system. This is followed by a description of an example application, which we will use throughout the remainder of the chapter and the thesis. Next, the main features of the model are illustrated. The chapter is concluded with a description of the implementation of the system.

3.1 Goals

The project was to design and build an object-oriented database, and to this end, there were a number of goals we wished to achieve. The first goal was to have a system which can adapt to model the real world, instead of having to adapt the real world to fit the system. The second goal was uniformity: we should be able to treat everything in the system in the same way and, as a result, the system should be simple to use. The third goal was to be able to change and extend the system to cater for new requirements dynamically as they arise in the application. Modularity was the fourth goal, whereby we can specify modules that define data and the operations to manipulate them. As a result, the application will be structured and easier to maintain. The fifth goal was type safety: the system should perform checks (either static or dynamic) to ensure that the operations are valid for the data on which they are operating. The sixth and final goal was persistence, whereby everything in the system would be able to persist in a uniform structure.

A number of the goals were satisfied by virtue of the system being an object-oriented database. The database aspect provides us with the required persistence and the object-orientation provides us with modularity and the required modelling capabilities. By defining everything as an instance, through the use of complex data structures and metatypes, the system is uniform and allows for dynamic extensibility. The latter is also provided for by the creation of new types, which is a basic part of an object-oriented system, as well as by the use of metatypes, generic types and extensible base types. The type safety of the system is achieved by typing all instances and type checking all operations.

Recovery and reliability, concurrency, and storage management are important features of any database system. However, in our system we have concentrated on the goals above and for this reason, these features and others are omitted. The system may at a later stage be extended, by adding these features, in order to make it a complete object-oriented database system.

3.2 The Data Model

HOOD is a Higher-Order Object Database that makes use of a large number of features from various models and combines them in a uniform manner. The structure system defines various forms of values which may occur in the database. The object system provides the persistence for these values and also a means of uniquely referencing and manipulating them. The structure and object systems provide the apparatus or foundations for defining an implementation model. In this chapter we deal with these two systems in Sections 3.2.2 and 3.2.3.

In the data model chapter the type system is defined in Sections 3.2.2 and 3.2.5, and specifies the concepts which are key to an object-oriented model. A number of constraints are specified for each of the various concepts. Provided the constraints are satisfied, a model will be consistent and exhibit the features of an object-oriented database. The concepts of instance, method, metatype, base type and generic type are also defined. Associated with these concepts are the mechanisms of subtyping and instantiation. The structure, object and type systems form the foundations for defining the data model. They are used in an exemplar to define a realization of the concepts specified in the type system. The exemplar is covered in Section 3.2.8. It provides an implementation data model which can be used to define a user data model.

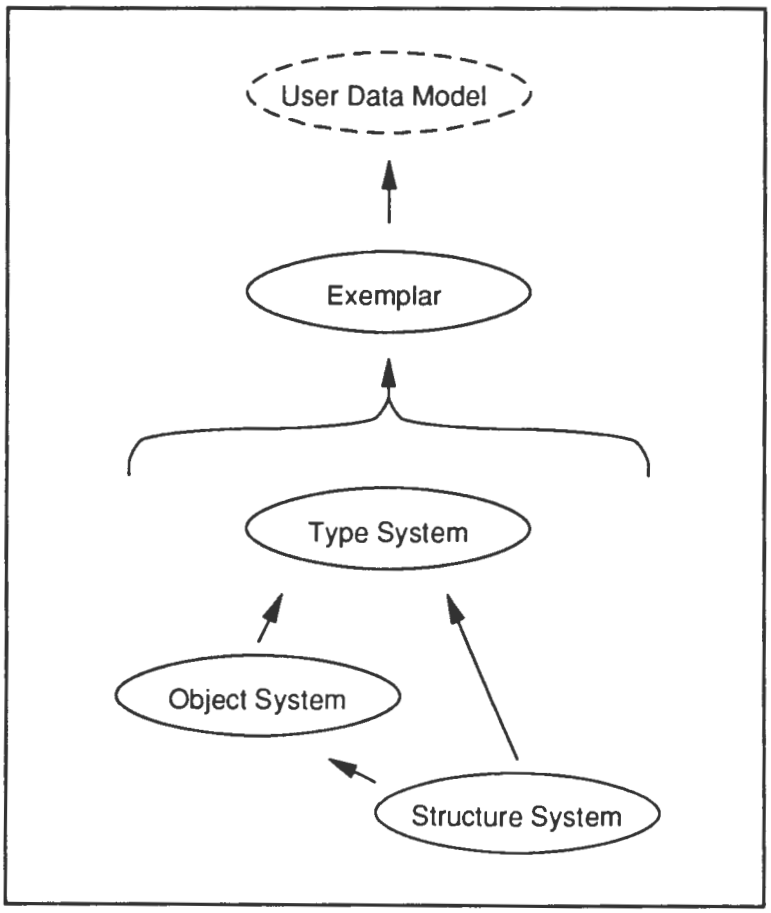

Figure 3.1: Structure of the Model

Figure 3.1 illustrates the structure of the model. At the bottom level are the foundations which specify the general requirements and constraints of an object-oriented database system. The exemplar is defined by making use of the constructs and features which are provided by the lower levels. There are a number of distinct exemplars which may be specified that satisfy the constraints of the type system. The particular one which we have chosen is detailed in Section 4.4. The next level is the user data model and is used to provide a simpler interface for the user. It removes the technicalities found in the implementation model and provides a user friendly means of using the constructs which the model provides. We have, however, not defined a user data model.

The object-oriented features found in this system are essentially the same as the set of mandatory features which are defined in [ABDDMZ90]. This conformance is encouraging because we only obtained this paper after the model had been designed and most of the implementation had been completed.

3.2.1 Reference List Example

We now present an example of modelling bibliographic reference lists which we will use to illustrate the various aspects of the system. A person may write a paper and refer to work done by other people. A reference contains the title and author of work, and information about the publication in which it appears. A reference list simply contains a list of these references.

There are four main concepts in the real world which must be modelled by the system. First there is the concept of a reference, which is the publication or the article in which the work appeared. Second there is the concept of the reference list and is built on the first concept, since it contains a list of such references. Third is the concept of the actual work which is being referenced, such as an article or a paper. A paper is a refereed work that appears in an academic journal. An article is the general concept which defines a piece of work that may appear in a magazine, newspaper, book, etc. Associated with an article is its author and title, and the publication in which it appears. Forth, is the concept of the various publications, such as journals, proceedings, books, reports, magazines, etc. which contain these works. A publication has a name, a date of publication and publisher. These three concepts give rise to the three basic types defined in the database, viz. Reference List, Article and Publication.

Figure 3.2 illustrates some of the relationships which may exist between various entities in the real world. The reference list is depicted with two of the works which it references. The one work is a paper and the other is a book, both of which have been written by the same person. The paper appears in a proceedings that is published by the same publisher who published the book. These are some of the simple requirements for this application. There should be a direct relationship between the concepts and entities in the real world and the structures used to model them in the system.

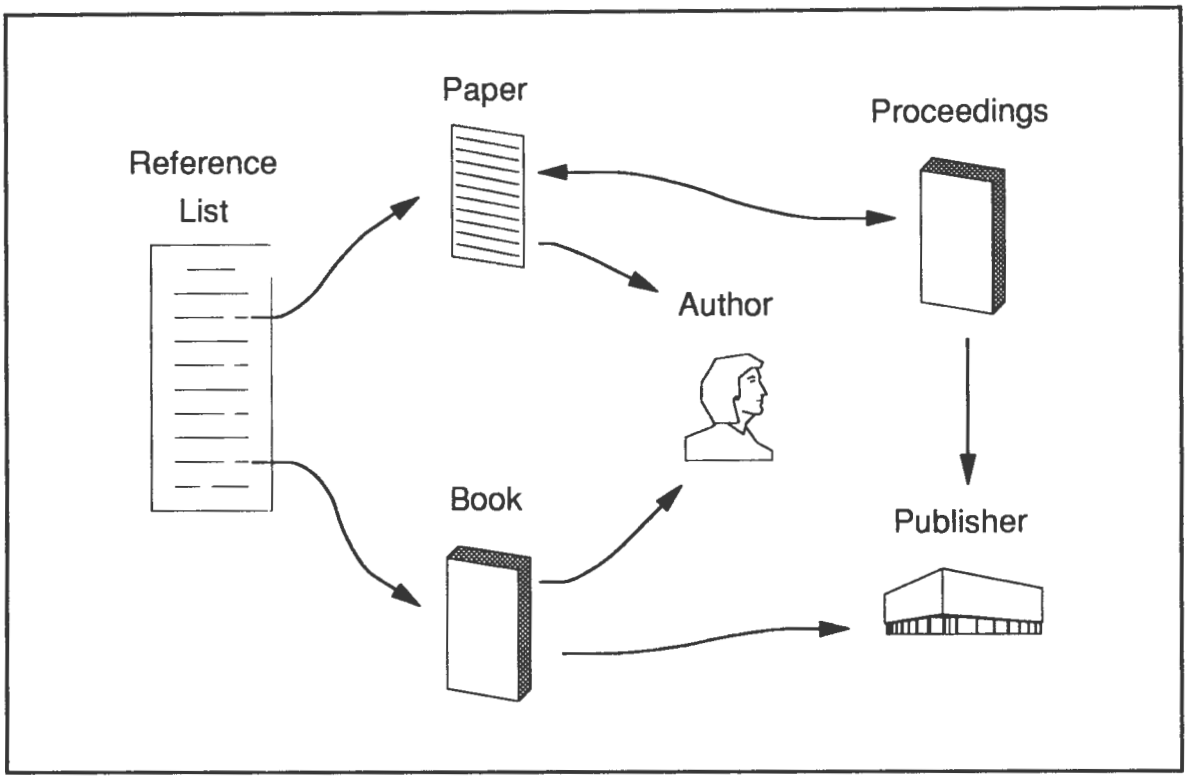

Figure 3.2: Publication Entities in the Real World

3.2.2 Types

The concept of a proceedings is modelled in the system by a type. The type defines the structure for its instances and the operations which may be performed on them. Type *Proceedings* is defined as follows:

```
(Type,
    (name:- 'Proceedings',
    state:- Product(=>Type,
        (title: String, date: Date,
        publisher: =>Organization, editor: =>Person,
        contents: List((work: =>Paper, pages: Product(Integer, Integer)),
        issn: String,   conf-add: String, conf-date: Date))
    behaviour:- {Print, ...},
    objects:- {Proc1},
    subs:- {},
    supers:- {Refereed Publication}))
```

The type has a name, 'Proceedings', which is used to identify it. The state of the type is used to define the structure of its instances. The state makes use of a structure which is defined in the structure system (see Section 4.1). The structure system defines structures quite similar to data types found in programming languages. The symbol => is used to define an object reference. The data structures have

been based on [Schm86], where the record, product, disjoint union and function constructors are defined, along with their assembly and disassembly operations. Some of these constructors have also been defined in [CaWe85], with the addition of subtyping rules. These rules form the basis for the subtyping rules used in the structure system. Ordinal types, ranges and enumerations, which are not dealt with in detail, are based on Ada, Pascal and [CaWe85]. The array and method constructors have been adapted from record and function constructors respectively.

The *Proceedings* type has a set of methods defined for it, which can be used to perform operations on its instances (e.g. *Print*). The methods are stored in the behaviour attribute of the type. The objects attribute stores references to objects which are instances of a proceedings (e.g. *Proc1*). The type also stores references to its supertypes and subtypes.

The type or class concept used in this system is called a type because it captures the properties of an ADT: it is an intentional specification of its instances through the use of a data structure and it is used in the type checking process. Classes are not defined in the system, but the extensional specification of a class is found in the type in that each type stores a reference to each of its objects. A type is used to model a concept in the real world by providing operations to create and manipulate instances of that concept. The system provides extensibility by allowing the user to define new types at will, which have the same status as the system-defined types. The system also allows new base types to be added which also share the same status as the other types.

3.2.3 Instances

Whereas a type models a real world concept in the system, instances are used to model specific cases of those concepts. The following SIGMOD proceedings is an instance of the proceedings concept and is defined as follows:

```
(Proceedings,
    title:- 'Proc. of the 1990 ACM SIGMOD Inter. Conf. on Management of Data',
    date:- 1/6/90,
    publisher:- Pub1,
    editor:- Person2,
    contents:- [(work:- Paper1, pages:- (1, 35)), ...],
    issn:- '2367-34578',
    conf-add:- 'Atlantic City, New Jersy',
    conf-date:- 4/6/90))
```

The instance stores all of the information relating to the publication in the real world which it is modelling. This information is stored in a value which is defined by the structure in the type's state. For example, the title of the proceedings is stored in the attribute 'title' and has the value 'Proc. of the 1990 ACM SIGMOD Inter. Conf. on Management of Data'.

In accordance with the terminology for types, we have termed the concept of an object an *instance* in our model. An instance is defined by the data structure in the type. Instances are split into two disjoint groups: objects and values. An object is an instance of a type's data structure and has the properties of an object as defined in the object system (see Section 4.2). An object may exist as an independent entity and contains information regarding its type. For example, the SIGMOD instance above has its type stored as its first component.

The object system is used to capture the basic properties of an object and its operations. The concept of object identity and object operations which are used in the model are based on [KhCo86]. The concept of an object and its identity are used to capture the notion of a unique entity. An object in the system corresponds directly to the entity being modelled in the real world. The object reference structure is used to reference objects and to create relationships (weak references) between objects and to share components between objects as discussed in [ZdMa90]. An object's identity is used as a means of referencing it. For example, *Pub1* and *Person2* in the above instance are object identities. In addition, the *Proceedings* type stores the identities of all of its objects in the 'objects' attribute. The references must be consistent and the concept of Nil is defined to combat dangling references. The ACM proceedings above is an object and has an identity (*Proc1*) which is used to reference it.

Objects are the only persistent structures. If something is to persist, it must be defined as an object. There are also temporary objects which enjoy the property of identity but do not persist. Once an object has been declared as persistent, the system ensures that the object is loaded from the disk and saved back to the disk.

A value, however, is dependent on the context in which it is used, such as a variable or as a component in a larger structure. Since the information regarding a value's type can always be obtained from its context, this information need not be stored in the value. A value is also defined by the data structure in its type, but omits the portion of information regarding its type.

Not all instances used in the system require the modelling capabilities of an object. The distinction between objects and values is made to allow for values which do not require persistence or an identity, and hence the additional overhead. The one-to-one correspondence between objects and entities in the real world is an important part of the system's modelling capabilities. If the system only provided objects and an arbitrary value is stored as an object, then there should be some entity in the real world with which it corresponds. This is not always the case as values are often required simply to store temporary information.

In the publication example, the entities in Figure 3.2 are modelled in the system by the objects depicted in Figure 3.3. Each object stores the information regarding the entity in the real world which it is modelling and also stores the identities of other objects to which it is related. The *Paper1* object stores the identity of ACM proceedings *Proc1* in its 'appearsIn' attribute, while *Paper1* appears in the 'contents' attribute of *Proc1*. This models the relationship which exists between these two objects. In Appendix A we have defined some of the types and instances used to model the concepts and entities

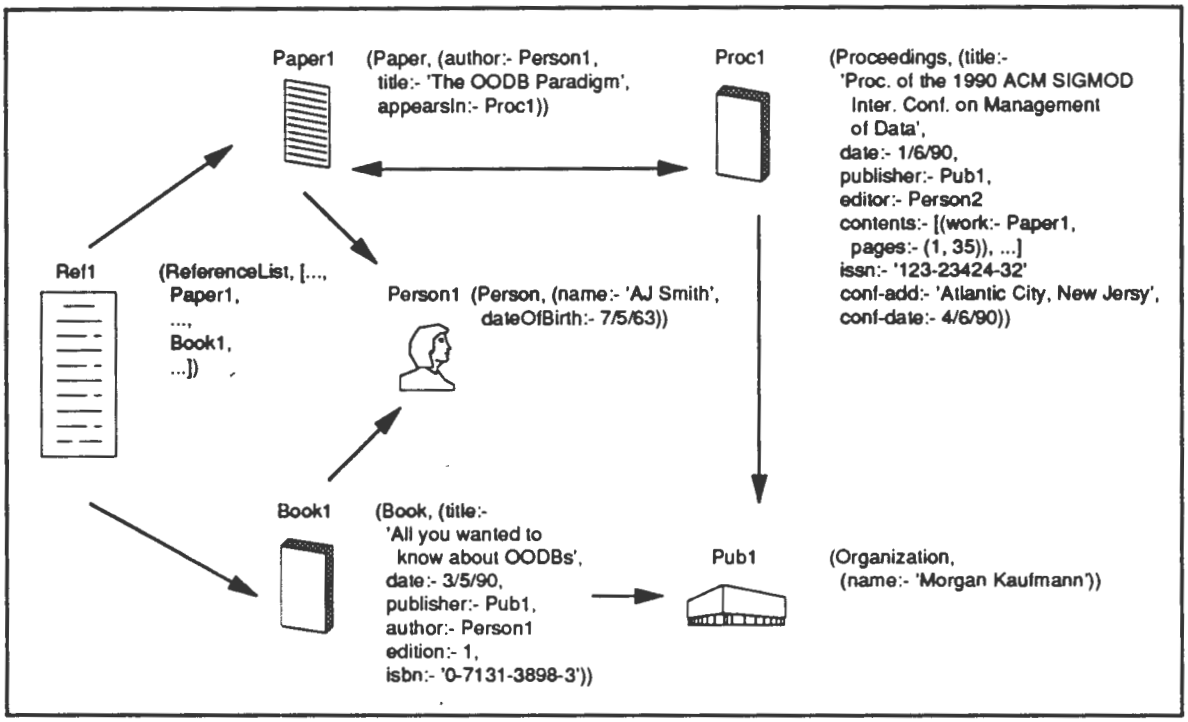

Figure 3.3: Annotated Publication Entities

which are depicted in Figure 3.3. The constructs used to specify these types and instances are defined in the following chapter.

3.2.4 Methods

The operations defined for an instance are specified in its type as methods. To use the method, we pass a message to the object specifying that it must perform the operations defined in the corresponding method. For example, a *Print* method is defined for type *Proceedings* as follows in the method language. It makes use of the conditional *IF* statement and also relies on *Print* methods which have been defined in other types, viz. String, Date, Person, Organization, etc.

```
(Method[Owner:- Proceedings, Param:- Boolean, Result:- Nil],
    (name:- 'Print',
    source:-'
        |/  'Proceedings: '.Print; this.title.Print;  '\nDate of Publication: '.Print; this.date.Print;
            '\nPublished by: '.Print; this.publisher.Print; '\nConference Address: '.Print; this.conf-add.Print;
            '\nConference Date: '.Print; this.conf-date.Print;
            if (param) |/
                '\nEditor: '.Print; this.editor.Print;  '\n: ISSN:'.Print; this.issn.Print; '\n: Papers:'.Print;  /|
            '\n'.Print;  /|'
    code:- ...;))
```

The *Print* method is defined for instances of type *Proceedings* and requires a boolean parameter. The parameter is used to specify either a full printout or a brief printout. This method can be used by passing the *Print* message to the object *Proc1* which performs the operations specified in the method. The following message pass requests the *Proc1* object to perform a full printout according to the actions of the *Print* method which is defined in its type.

```
Proc1.Print(true)
```

A method is defined by a method structure which forms part of the structure system. Since a method is defined by a structure, it is a value and may be treated like all other values. The methods in the system are defined as objects of a special type called *Method*. The reason for doing this is to make use of the same constructs which are defined for instances. The result is that there is a uniform means of dealing with both methods and instances. An object is the only persistent construct in the system and by making the methods objects, we can store them in the system. By using the identity provided by objects, methods can be referenced from the types in which they are defined and do not need to be stored as part of the type structure. Identity also allows us to reference specific methods, even if there are other methods with the same name. Since methods are first-class values, there are also operations which can be performed on them, such as, comparing them to determine if they are identical or using them in parameters. The attribute 'code' contains the compiled version of the method which is defined by the string in attribute 'source'. It contains the machine instructions which are executed by the abstract machine and forms the value which is associated with the method structure.

3.2.5 Subtypes

Subtyping is a mechanism provided by the system that allows us to model the specialization of a concept by another concept. For example, there is the general concept of a refereed publication. This is some form of printed matter which has a title, a date when it was published, an organization that published it, an editor, a list of articles which appear in it and an ISSN number. The concept of a proceedings is a specialization of the publication concept, since it has all of the properties of a refereed publication, and it also has additional properties such as conference address and date. The *Refereed Publication* type is defined below. The *Proceedings* type seen earlier is a specialization of this type and adds the attributes 'conf-add' and 'conf-date' to its instances.

```
(Type,
    (name:- 'Refereed Publication',
    state:- Product(=>Type,
        (title: String,
        date: Date,
        publisher: =>Organization,
        editor: =>Person,
        contents: List((work: =>Paper, pages: Product(Integer, Integer))))
    behaviour:- {...},
    objects:- {...},
    subs:- {Proceedings, Journal},
    supers:- {Edited Publication, Periodical}))
```

The *Refereed Publication* type stores a reference to each of its subtypes in the 'subs' attribute. Likewise, the *Proceedings* type stores a reference to each of its supertypes. As can be seen, *Proceedings* is a subtype of *Refereed Publication*. *Refereed Publication* is a subtype of both *Edited Publication* and *Periodical*; thus, multiple inheritance is supported by the system. The subtyping relationship between types gives rise to the subtyping hierarchy, which depicts the inheritance that occurs between types. In Figure 3.4, we illustrate subtyping between the various types which are used to model the concepts of publications, reference lists and articles.

The term subtyping is in keeping with the use of types and type checking by the system. The various forms of inheritance identified by [ABDDMZ90] are all supported by the subtyping mechan-

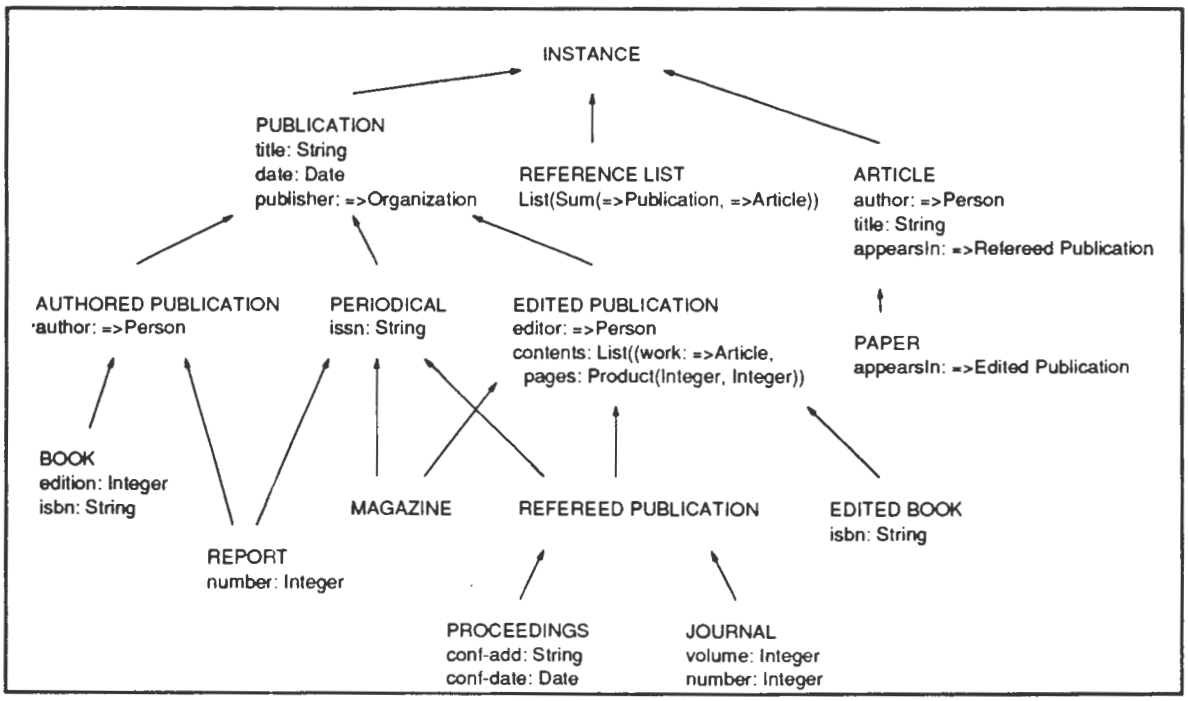

Figure 3.4: Publication Hierarchy

ism. The data structure used in a subtype must be a subtype of the data structure used in the type. This allows the methods defined in the types and supertypes to be inclusion polymorphisms as defined by [CaWe85]. The uniformity of the system is also evident in subtyping, since all types, including base types and generic types, may have subtypes defined for them.

3.2.6 Metatypes

The concept of a metatype and the initial set of system types are based on [Coin87], but have been adapted to fit in with the concepts of a type and an instance. Unlike ObjVLisp, the structures used for types and instances are typed and form part of a database with object identity. The metatype concept provides the system with a reflective means of definition. This gives the system power and flexibility, whereby it can adapt to changes and new requirements as they arise. The metatype concept also allows the system to treat types as objects.

Metatypes allow us to achieve uniformity in another respect. All types in the system are defined as objects. Like methods, they are also stored as objects and may be referenced by an identity. Thus, there is only one persistent construct in the form of an object along with only one form of identity and associated referencing mechanism. An object stores its typing information by storing the identity of its type. A subtyping relationship between types is specified by each type storing the identity of the other type in either its 'subs' or 'supers' attribute. The name of a type is used as a symbolic reference to a type and forms part of the name space. But essentially all types are referenced by their identities.

The type concept gives rise to the instantiation mechanism, whereby an instance is instantiated from its type. The metatype concept allows us to have various levels of instantiation, such as between type and instance, metatype and type, meta-metatype and metatype, etc. The use of the instantiation mechanism creates a hierarchy of types and their instances.

3.2.7 Generic Types

The generic type defined here originated from the generic template defined by Ada, while some of the typing structures are based on the parametric polymorphism structures defined in [CaWe85]. The model provides structures for defining generic parameter lists and hence generic types (see Section 4.1.3).

Generic types can be viewed as a means to extend the set of constructors provided by the system. A constructor may be applied to any set of types and yields a structure with certain basic properties that are independent of the component structures. A list for example has basic properties such as a head and a tail, which are independent of the data stored in the list. A constructor also has a set of operations to manipulate its values, and they too are independent of the component structures. The operations operate uniformly for any value stored in the list. For example, given a list, the *tail* operation will return another list.

A stack can be defined as a generic type by internally using a list. The values stored in the stack (the list) are defined in terms of a generic parameter called *Data*. The generic type *Stack* is defined as follows:

```
(TypeGen,
    (name:- 'Stack',
    state:- Product(=>TypeGen, (value: List(Data)))
    behaviour:- {Pop, Push, Empty, ...},
    objects:- {Stack1, Stack2, Stack3},
    subs:- {},
    supers:- {InstGen},
    generic:- [Data· Struct]))
```

The structure of an instance is defined by a list of values which have the structure *Data* parameter. The operations such as *Pop* and *Push* which are used to manipulate a stack are also defined using the generic parameter. The resulting generic type can be conceptualized in the same manner as the list constructor, and may also be used in the same way. A list of people and a stack of people are denoted as follows:

```
List(=>People)
Stack[Data:- =>People]
```

The component structure in the list can be used as the component structure in the stack, by simply specifying it as the value to be used for the generic parameter. The generic parameter *Data* is replaced by the object reference to people throughout the type. With the use of subtyping and since generic types are treated basically like any other type, it is possible to define a subtype of a generic type. If we view generic types as defining constructors, then it is possible for us to specialize some of the constructors.

3.2.8 Exemplar

The structure and object systems provide the constructs for defining an object-oriented database model. The type system specifies the constraints which must be satisfied in order for the model to be object-oriented. We have defined such a data model and it is called an Exemplar. The exemplar is not unique, since there are many other models which can be defined that satisfy the constraints.

The concept of a type is defined in the system by the metatype *Type*, which defines the structure of a type and the messages which may be passed to them. Since *Type* is a metatype and since we treat all types as instances, there must exist some other type of which *Type* is an instance. Since *Type* defines all types, we have defined it as its own instance. *Type* forms the root of the instantiation hierarchy. The model also requires that everything be an instance, to which end we have defined a type *Instance*. It defines the general structure of an instance and forms the root of the subtyping hierarchy, since every

type is required to produce instances. Because *Instance* is a type, it is an instance of the type *Type*. *Type* is also required to produce instances, therefore it is also a subtype of *Instance*.

From these principal two types, the other types used in the model are defined. These include generic types, method types, base types and the types used specifically for the application, such as type *Proceedings*. The instantiation and subtyping hierarchies are depicted in Figure 3.5. The nodes in the graph are objects and each plane contains different classes of objects. The Metatype plane contains only metatypes. The Types plane contains ordinary types and excluded metatypes. The Instances plane contains all ordinary instances and thus excludes all types. A dotted arrow denotes instantiation and is read "X is an instance of Y", while a solid arrow denotes subtyping and is read "X is a subtype of Y".

To summarize, the system provides uniformity because types and methods are all treated as objects. Thus there is only one form of identity and one mechanism for persistence. Everything is referenced and stored in the same way, which makes the system simple in this respect. Through the use of metatypes, the system is flexible and may be extended by defining new base types and ordinary types. The set of constructors for the system can also be extended by defining new generic types.

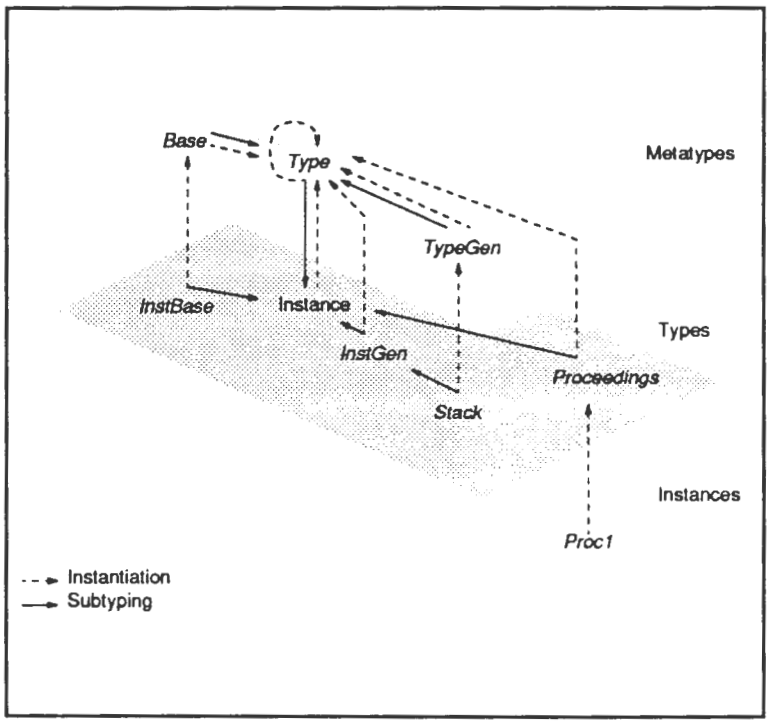

Figure 3.5: Root Hierarchies

3.3 Implementation

We decided to implement only those features which are essential to the system, as trying to implement the entire system in the time available would have been an impossible task. The purpose of implementing part of the system is to demonstrate the features of the model, identify possible problems with the

model which were overlooked in the design and find corresponding solutions, and show that it is possible to implement the system specified in the model.

We have implemented the following data structures and their corresponding values: base, record, product, set, list, method, reference structures and higher-order structures. We have not implemented enumerations, ordinals, sums and arrays. We have defined operations to facilitate the persistence of these values and structures, although we have not implemented the temporary objects. There are also operations for comparing values for equality and structures for equivalence and subtyping. The operations defined in the model for each structure have been implemented. These structures and values, together with their operations, form the kernel of the system.

We have also defined and implemented a method language. The language allows all of the operations defined in the model as well as the formulation of complex data values. The language also defines a set of simple control statements. A compiler type checks the method language and then generates code for an abstract machine. Compilation occurs in the context of a method structure and the database. The method structure is used to type check the recipient of the message, the parameter value and the result of the method. The database is used to check the types used in the method and messages which are passed. Type checking ensures that the operations are correct; when static type checking cannot be performed, dynamic type checks are generated. Type checking also includes a type resolution process, which is used to infer the typing of untyped values. It also ensures the encapsulation of instances and the use of subtypes.

The compiler produces code which is the value associated with a method structure. This code is defined for an abstract machine which operates on the database. The abstract machine defines the primitive operations for data structures, control structures, message passing and special operations which may be linked to the system. Only a portion of the abstract machine has been implemented.

The entire database is stored in an object table. An object is obtained from the object manager by specifying its identity. The object table and manager ensure the persistence of each object and the integrity of references to objects. There is also a name table and manager which implements the name space associated with the database. The name table stores unique identifiers and corresponding object identities. These names may then be used in methods to access the objects. In the present implementation, the name space has been limited to the names which are used for types.

Chapter 4 Model

In this chapter we describe the model for HOOD (Higher-Order Object Database). The structure system is used to define structures and associated values. For each structure a set of primitive operations is defined to manipulate the values. Structural equivalence and structural subtyping are defined to compare structures. Values defined by the structure system are used by the object system to create objects. An object contains an identity and a value as defined in the structure system. The objects are used to provide persistence. These two systems provide the foundations for expressing values in the type system.

The type system defines object-oriented concepts by using functions and constraints. Functions are defined over the values from both the structure and object systems. In order for a value to be classed as an instance of a concept, it must satisfy the conditions of the required constraints. The exemplar is a realization or concretization of values which satisfy the constraints of the type system. It forms an implementation data model that defines a database.

4.1 Structure System

The structure system is the foundation of the model, since it provides the basis for expressing values in the database. The object and type systems which follow are defined in terms of this system. We have defined various operations for comparing the values and structures which are intrinsic to the structure system. This section deals with the constructors which are used to build the structures, the operations defined to compare these structures, the values associated with the structures and the operations defined for these values.

4.1.1 Constructors

A constructor is used to build a structure which in turn defines a set of values. The concept of a structure which we define here is quite similar to a data structure used in programming languages. The following constructors exist in the model: Base, Product, Record, Sum, Set, List, Array, Method, Value Reference, Object Reference, Nil, Value, Gen (generic), GPLS (generic parameter list structure), Struct (structure), StructLim (limited structure) and Ref (reference). In this section, we cover only Base, Record and Reference constructors. Product, Sum, Set, List and Array constructors have similar definitions, the same aspects of which are covered in Appendix B. The Method constructor is covered in Section 4.1.2.2, while *Gen*, GPLS, GPLV (generic parameter list value) and generic references are dealt with in Section 4.1.3.

Associated with each constructor are operations to compare the structures it builds. Structural Equivalence compares two structures built from the same constructor and determines if they are

equivalent — the set of values defined by the one structure is identical to the set of values defined by the other. The operator $\equiv_s$ is used to denote this operation and defines an equivalence relation.

Structural subtyping compares two structures built from the same constructor and determines if the one structure is a structural subtype of the other. (Structural subtyping is a relation defined on data structures, while subtyping is a relation defined on types and will be covered in the type system.) The values defined by the one structure form a 'subset' of the values defined by the other structure. The operator $\leq_s$ is used to denote this operation and defines an order relation. Structural subtyping facilitates inclusion polymorphism — an operation which is well defined for a value of a given structure, will remain well defined for all values defined by structures which are structural subtypes of the given structure.

Inclusion polymorphism refers to the typing nature of an operation which allows for substitution. *Substitution* is a property associated with values, which allows an expected value defined by a structure to be replaced by a value defined by a substructure. Although inclusion polymorphism describes the typing nature of an operation, we also use it to refer to the more general property associated with structures, values and operations. In [ABDDMZ90] these properties are collectively called inheritance.

Each structure defines a set of values. If a value is an element of this set, then we say that the value is an element of the structure. The set of values defined by a structure have assembly, disassembly, relational and translation operations defined for them. There is also a special value defined for all structures called Nil. This encompasses the concepts of empty, void, blank, zero, nothing and null. For set and list structures, it denotes the empty set and empty list. In base values such as integers, it denotes the zero value.

An assembly operation generates a value which is an element of that structure. The disassembly operation, on the other hand, produces from a constructed value one of the component values which was used to assemble it. The relational operations are used to compare values defined by the structure. Equality and inequality are the most common relational operations and are defined for almost all of the structures.

Given two structures, where the one is a structural subtype of the other, the translation operation is defined to convert a value defined by the one structure to a value defined by the other structure. Translation may occur in either direction and, in the case where a value has an undefined component, the Nil value is used.

4.1.1.1 Base Structures

Base types define the primitive values and operations found in the database. More complicated values are assembled by using the base values as components. A base type is implemented externally and then linked into the database. The database requires a specification of the size requirements for the base value — the number of bytes required externally to represent the value, and the operations which implement the messages of the base type. The base constructors build various base structures, defined by the size requirement of the base values. There are two main base structures denoted as follows:

Base[–], *Base*[*X*]

The first structure denotes a dynamic base structure, where the value may have a variable length. The second structure denotes a fixed base value of X bytes, where X is any fixed positive integer. These sizes specified in the structure will correspond to the implementation of, for example, a character string and an integer type, respectively. The system will maintain the values with these requirements in memory and on disk.

The assembly, relational and any translation operations are all defined externally. Disassembly operations may also be defined externally if they are necessary.

4.1.1.2 Record Structures

The record constructor takes a number of structures and unique labels, and constructs a record structure denoted, for example, as follows:

(author: Person, title: String, appears-in: Periodical)

author is a label called the attribute name and *Person* is a structure called the attribute structure; together they form an attribute. The record structure defines a set of values, where each value has the following form:

(author:- P, title:- 'Object....', appears-in:- P)

where P is a *Person* value and P is a *Periodical*. The attribute names used in the structure appear in the value, along with a value from the attribute structure called the attribute value. The colon-dash operator is used in the record value, while the colon operator is used in the structure. The reason for this is explained in Section 5.3.2.1. Together, the attribute names and values form a record value. The record structure defines a set containing all such values:

$$(L_1\colon S_1, \ldots, L_n\colon S_n) = \{\, (L_1\!:\!-V_1, \ldots, L_n\!:\!-V_n) \mid V_i \in S_i,\ 1 \le i \le n \,\}$$

The Nil record value contains, for each attribute value, the Nil value as defined by the attribute structure.

$$Nil = (L_1\!:\!-Nil, \ldots, L_n\!:\!-Nil)$$

$$(L_1\!:\!-Nil, \ldots, L_n\!:\!-Nil) \in (L_1\colon S_1, \ldots, L_n\colon S_n)$$

Two record structures are structurally equivalent if they have the same number of attributes, and the attributes in the two structures can be paired in such a way that the attribute names are equal and the attribute structures are equivalent. That is,

$$(L_1\colon S_1, \ldots, L_n\colon S_n) \equiv_S (K_1\colon T_1, \ldots, K_n\colon T_n)$$

iff

$$\forall L_i \exists K_j (L_i = K_j) \wedge (S_i \equiv_s T_j) \wedge \forall K_j \exists L_i (K_j = L_i) \wedge (T_j \equiv_s S_i)$$

For example, the following two record structures are structurally equivalent:

(author: Person, title: String, appears-in: Periodical)
(title: String, appears-in: Periodical, author: Person)

An attribute value is obtained from a record value by using the disassembly operation, record subscript. This operation makes use of the dot operator which requires a record value and an attribute name. The result of the operation is the corresponding attribute value. For example,

(author:- P, title:- 'Object....', appears-in:- J).title = 'Object....'

Equality and inequality are the two relational operations defined for record values. They require both values to be elements of the same structure and the result is a boolean value. In the case where the values are from different structures, which are not related by subtyping, then the operation is undefined. The equality operation will return *true* if all pairs of attribute values in the two record values are equal, otherwise the result is *false*. That is,

$$=: ((L_1: S_1, ..., L_n: S_n) \times (L_1: S_1, ..., L_n: S_n)) \rightarrow Boolean$$

where

$$(L_1 :- V_1, ..., L_n :- V_n) = (L_1 :- W_1, ..., L_n :- W_n)$$

iff

$$(V_1 = W_1) \wedge ... \wedge (V_n = W_n)$$

Record structure T is a structural subtype of T' provided the following conditions are satisfied:

- Every attribute name that appears in T', appears in T — these attributes are called mandatory attributes.
- The attribute structures of the mandatory attributes in T are structural subtypes of the attribute structures in T'.
- Additional attributes may be defined in T.

In other words, we have that,

$$(K_1: W_1, ..., K_n: W_n, ..., K_m: W_m) \leq_s (L_1: V_1, ..., L_n: V_n), m \geq n \geq 1$$

iff

$$\forall L_i \exists! K_j (L_i = K_j) \wedge (W_j \leq_s V_i)^{\dagger}$$

† The symbol $\exists!$ denotes that there exist unique values which are in one-to-one correspondence with the values quantified by $\forall$.

In the example of a record structure used earlier in this section, we have a record structure consisting of the attributes 'author', 'title' and 'appears-in'. Given that *Journal* is a subtype of *Periodical*, we have the following:

$$(author: Person, title: String, appears\text{–}in: Journal, date:Date) \leq_s (author: Person, title: String, appears\text{–}in:Periodical)$$

Note also that an additional attribute 'date' has been added. It is possible for a record structure to have more than one superstructure. This is closely related to the concept of multiple inheritance. Our newly defined substructure is a structural subtype of the following record structure also:

$$(author: Person, title: String, appears\text{–}in: Journal, date:Date) \leq_s (author: Person, title: String, date: Date)$$

Assume record structures T and T' share attribute name A, but have unrelated structures for A. Then no single record structure can include A and be a structural subtype of both T and T'. This is called a *name conflict.*

The following two structures have no single structural subtype because a name conflict occurs on the attribute 'appears-in'. In the first structure, the attribute denotes the journal in which the article appears, while in the second structure it denotes the date on which the article appears in the journal.

(author: Person, title: String, appears-in: Journal, date: Date)

(appears-in: Date, where: Journal, contents: String)

The only solution to this problem is to rename the conflicting attributes with more appropriate names.

Structural subtyping is defined to facilitate inclusion polymorphism. Any operation defined for a record value is typed by the record structure, specifying the attribute names and structures. A value defined by a substructure contains all of the attributes appearing in the structure; hence, operations defined on the structure are also valid for the substructure.

For example, an operation making use of the 'appears-in' attribute, expects a value that is defined by the *Periodical* structure. But in the substructure, this attribute is defined by the *Journal* structure. Since *Journal* is a structural subtype of *Periodical* any operation which is performed on the value is well defined because of the structural subtyping.

$$(author:\text{–}P, title:\text{–}\ 'Object....', appears\text{–}in:\text{–}J, date:\text{–}1/1/92) \in (author: Person, title: String, appears\text{–}in:Periodical)$$

where P is a *Person* value and J is a *Journal.* In effect, all values defined by substructures are elements of the structure.

A record value may be cast from its structure to a substructure by casting each of the attribute values to the corresponding attribute structure in the substructure. The additional attributes in the substructure are added and their values are set to Nil. To cast a value from a substructure to a structure, the additional attributes are simply discarded and the mandatory attribute values are cast to the corresponding attribute structure in the structure. The cast operation is denoted by the colon, ':' operator.

```
(author:- P, title:- 'Object....', appears-in:- J) :
        (author: Person, title: String, appears-in: Journal, date: Date)

= (author:- (P: Person), title:- ('Object....': String), appears-in:- (J: Journal), date:- Nil)

= (author:- P, title:- 'Object....', appears-in:- J, date:- Nil)
```

In this example a record value is cast to a substructure. The 'date' attribute is added and assigned the Nil value. The remaining attributes (mandatory attributes) are in turn cast to their corresponding structures. In the next example a record value is cast to a superstructure. The *date* value is discarded leaving the mandatory attributes which are cast to their corresponding structures.

```
(author:- P, title:- 'Object....', appears-in:- J, date:- 1/1/92) :

    (author: Person, title: String, appears-in: Periodical)

= (author:- (P: Person), title:- ('Object....': String), appears-in:- (J: Periodical))

= (author:- P, title:- 'Object....', appears-in:- J)
```

4.1.1.3 Reference Structures

The structure system is used to define values and structures, which are used in the type system to define types and their instances. A type defines an instance, but the structure of the instance is encapsulated and hidden. The reference constructors are used to build structures whose values are the instances of a type.

There are two classes of reference structures, object references and value references. An object reference structure is denoted by the object reference constructor, '=>', followed by the name of a type, while a value reference structure is simply denoted by the name of a type. For example, in the pair of structures

```
=>Person, String
```

the first is an object reference to a type called *Person*, while the second is a value reference to the *String* base type. The differences between object and value structures are provided later in Section 4.3.

The only operations associated with these values are the methods defined in their types. The structural equivalence and structural subtyping of these structures is based on the equivalence and

subtyping defined in the type system. Equivalence between types is denoted by '≡' (see Section 4.3.2) and two reference structures are structurally equivalent if their types are equivalent. That is,

$$=>T \equiv_S =>S \ \textit{iff}\ T \equiv S$$

Subtyping between types is denoted by '≤', (see Section 4.3.3). One reference structure is a structural subtype of another reference structure if the one type is a subtype of the other. That is,

$$=>T \leq_S =>S \ \textit{iff}\ T \leq S$$

4.1.2 Higher Order Structures

In this section we deal with the higher-order features found in the structure system. Put simplistically, everything in the structure system is a value. All structures are denotable values and are defined as elements of higher-order structures. In addition, all methods are values which are elements of a method structure.

4.1.2.1 Meta-Structures

There are three higher-order structures which define structures as their values, viz. *Struct*, *StructLim* and *Ref*. The structure *Struct*, short for structure, defines all structures as its set of values. As a result, it also appears as one of its own values. *StructLim* defines a set of values which contains a limited set of structures, forming a proper subset of the values defined by *Struct*. This set excludes structures which contain Base or Generic structures and is used to specify the structures which may be used as components in other structures, such as methods. *Ref* defines a set of reference structures (object and value reference structures), itself a proper subset of the values defined by *StructLim*. The main purpose of the *StructLim* and *Ref* structures is to limit the structures which may be used as components for Method structures and parameters for GPLSs. This prevents base values and GPLVs from being passed as parameters. However, by using the *Struct* structure, structures with base and GPLV components may be defined. As an example, a record structure is a value defined by the *Struct* structure and is assembled as follows:

(author: Person, title: String, appears-in: Periodical)

Associated with the assembly operation is the Nil value. The Nil value in this case is the Nil structure, which as a structure defines an empty set of values.

For each value (structure), various disassembly operation have been defined to extract component values (structures) in much the same way as disassembly operations extract the component values. For example, the attribute structure for 'author' is obtained from the record structure as follows:

(author: Person, title: String, appears-in: Periodical).author = Person

The resulting structure is a value reference to type Person. The relational operations are used to compare values defined by the same structure. In this case, structures are compared using the structural equivalence and structural subtyping operations. The results of these operations are boolean values.

As with all structures, structural equivalence and structural subtyping operations are used to compare them. These operations are also defined for the higher-order structures. Each of these structures is structurally equivalent only to themselves. Structural subtyping is defined according to the set inclusion of the values defined by the structures. Hence, we have

$$Ref \leq_s StructLim, StructLim \leq_s Struct$$

The translation operation casts a value from one higher-order structure to another higher-order structure. The translation of a structure is defined only if it is an element of the higher-order structure to which it is being cast. No conversion occurs within the value during the translation process, as was the case with record translation.

4.1.2.2 Methods

Most of the operations performed by the database take the form of messages. A message is passed to an instance of a type with an optional parameter. The instance (also called the recipient) then reacts by performing the sequence of instructions specified in the message's method and may yield a result. A method may be treated as a value and therefore has a structure and various operations associated with it.

The method structure types a method (value) and consists of three components. The first component is the owner and specifies a *Ref* structure. Only instances of the reference structure's type may be passed messages which are values of this method structure. Thus, the owner component defines the typing of the recipient of the message. The second component is the parameter and specifies the structure of the parameter passed with the message. The third component is the result which specifies the structure of the value which results from performing the method. The second and third components may only be specified by structures which are defined by *StructLim* structures. The following method structure defines a set of methods (values) that may be passed to instances of type *Paper*, with a parameter defined by an instance of type *Journal* and yields a result which is a *Person*.

Method(Paper, Journal, Person)

The associated methods are owned by type *Paper*. Method structures are structurally equivalent if the owner, parameter and result structures are structurally equivalent respectively. That is,

$$Method(O_1, P_1, R_1) \equiv_s Method(O_2, P_2, R_2)$$

$$iff$$

$$O_1 \equiv_s O_2, P_1 \equiv_s P_2, R_1 \equiv_s R_2$$

The method assembly operation is compilation: given a method specified in the method language, compilation type checks and converts it into an executable (assembled) form — the method value. The Nil value is the Nil method which performs no operation. If this method yields a result, then this value is the Nil value as defined by the result component of the method structure.

The disassembly operation for method values is message passing. Given an instance of the owner's type and a value defined by the parameter, the message pass "extracts" the result value by disassembling (executing) the method value.

Two methods are equal if for all instances and parameter values they yield the same results. The implementation of such an operation is generally undecidable, thus only the identical operation has been defined for method values. Two methods are identical if they are defined by exactly the same code. Since the code is the same, the effect of the method will be the same, thus the two methods are equal. Note that methods may be equal, but not necessarily identical.

Method structures may also be related via structural subtyping. Method structure M is a structural subtype of method structure M' if the owner and parameter of M are superstructures of those of M', while the result of M is a substructure of the result of M'. That is,

$$Method(O_1, P_1, R_1) \leq_s Method(O_2, P_2, R_2)$$
$$iff$$
$$O_2 \leq_s O_1, P_2 \leq_s P_1, R_1 \leq_s R_2$$

Given that *Paper* is a subtype of *Article*, *Journal* is a subtype of *Periodical* and *Student* is a subtype of *Person*, the following structural subtyping relationship exists for two method structures:

$$Method(Article, Periodical, Student) \leq_s Method(Paper, Journal, Person)$$

A message may be passed to a value where the value's structure is either equivalent to or a substructure of the owner structure. If one looks at the converse, a value may be passed a message defined by an owner structure which is either a superstructure of or structurally equivalent to the value's structure. If we replace a method with a substructure method, then from the method's perspective it is being passed to a value which is a substructure. The same principle holds for the parameter structure. For example, an instance of type *Paper* is passed a message defined for type *Article*, since the instances for type *Paper* are included in the instances for type *Article*, the message is well defined.

In the context in which the message is passed, a result with a specific structure is expected and is defined by the result component. A substructure method produces a result which is a substructure of the value expected by the context. Since we can replace a value with a substructure value, the result of the substructure method is well defined for the context.

4.1.2.3 Value

The *Value* structure is used to store a particular instance of a type within the type. A type defining a range of values is an example of where this structure is used. The type defines the instances, but is also

required to store the upper and lower bounds of the range. The *Value* structure is used to define the structure of these two values. The instances of the range are defined by the state structure of the range type. The *Value* structure specifies that the upper and lower bounds are defined by the structure in the type's state. An example showing how this structure works is provided in Section 4.4.4.

4.1.3 Generic Structures

The generic structures defined in the structure system are not themselves generic but are used in the type system to define generic types. There are three generic structures. The primary generic structure is the Generic Parameter List Structure (GPLS) which is used in a generic type to define the generic parameters. A higher-order structure called *Gen* defines a set of values consisting of all GPLSs. Each GPLS defines a set of Generic Parameter List Values (GPLVs) which are lists containing values for the generic parameters. A GPLV defines what is called an *implied type* from a generic type.

4.1.3.1 Generic Parameter List Structure

The GPLS defines and types parameters which are used in generic types. It is defined relative to a generic type and can not be used independently. It consists of a list of parameter names and structures. The structures type the values which the parameters represent.

For example, a generic type can be defined for a binary tree which stores sorted data. The logic of the tree is the same regardless of the type of data being stored. The tree requires a data type and two operations to sort and compare the data. The data stored in the tree is abstracted to a parameter called 'data'. The GPLS for the generic tree type is as follows:

```
[data: Struct, equals: Method(data, data, Boolean), precedes: Method(data, data, Boolean)]
```

The parameter called 'data' has the structure *Struct*, since the tree may store data which is defined by any structure. The parameter called 'equals' is defined by a method structure and defines a method which is used to determine if two data values are equal. The other parameter called 'precedes' is also defined by a method structure and determines if one data value precedes another.

The actual structure and operations used in a specific binary tree are stored in the GPLV. A GPLV for the example above is assembled as follows:

```
[data:- Journal, equals:- Equals, precedes:- Older]
```

and defines for the generic tree type, an implied *Journal* tree type sorted by the age of the journal.

The GPLS is defined in the generic type (see Section 4.3.4) and the GPLV is stored in the instance. The disassembly of a GPLV occurs implicitly, due to the relationship between the GPLS and the generic type. When the instance makes use of information or methods defined in the generic type, the values in the GPLV are used to replace the parameters which occur in the generic type.

Two GPLVs are equal if they are elements of the same GPLS and, for each parameter, their values are equal. Two GPLSs are structurally equivalent if they contain the same number of parameters and the parameters can be paired such that the parameter names are equal and the parameter structures are structurally equivalent. That is,

$$[p_1: S_1, ..., p_n: S_n] \equiv_S [q_1: T_1, ..., q_m: T_m]$$
$$iff$$
$$(m = n) \wedge \forall p_i \exists! q_j ((p_i = q_j) \wedge (S_i \equiv_S T_j))$$

Structural Subtyping is defined for GPLSs to facilitate the subtyping of generic types. The subtyping principle employed here is similar to that of record subtyping. That is,

$$[q_1:T_1, ..., q_m:T_m] \leq_S [p_1:S_1, ..., p_n:S_n]$$
$$iff$$
$$\forall p_i \exists! q_j ((p_i = q_j) \wedge (T_j \leq_S S_i))$$

For example, given the following two GPLSs, the one is a structural subtype of the other:

$$[data: StructLim, equals: Method(data, data, Boolean), precedes: Method(data, data, Boolean)]$$
$$\leq_S$$
$$[data: Struct, equals: Method(data, data, Boolean)]$$

This is because both parameters 'data' and 'equals' appear in the subtype, while *StructLim* is a substructure of *Struct* and the method structures are equivalent.

Given GPLSs L and L' such that

$$L \leq_S L'$$

the values defined by L are included in the set of values defined by L'. All parameters which appear in L' appear in L. Due to inclusion polymorphism each of the parameter values in L is an element of the corresponding parameter structure in L'.

Translation is defined for GPLSs and GPLVs so that generic instances may be cast. Casting from a substructure to a superstructure and vice versa is identical to that for a record structure.

4.1.3.2 Generic Parameter List Value

A GPLV is an element of a GPLS, which is used in an instance of a generic type to define the actual values for the generic parameters which appear in the generic type. A generic type, with its parameters replaced by actual values, is called an *implied type*, which resembles an ordinary type. Subtyping is defined between implied types and requires, as part of its definition, the structural subtyping of GPLVs. In this context a GPLV is used as a structure and thus requires the various operations associated with structures.

There are no explicit values associated with a GPLV as is the case with other structures, instead the values are defined by the implied type. The assembly, disassembly, relational and translation operations are not defined for a GPLV since they are defined for each implied type.

A GPLV is structurally equivalent to another GPLV if they are equal as defined in the previous section. For structural subtyping there are two cases to consider. In the first case both GPLVs are elements of the same GPLS. GPLV V_1 is a substructure of GPLV V_2, if the value of every structure parameter in V_1 is a substructure of the corresponding value in V_2 and all corresponding value parameters in V_1 and V_2 are equal. For example,

$$[data{:-}\ Journal,\ equals{:-}\ Equals] \leq_s [data{:-}Article,\ equals{:-}\ Equals]$$

$$iff$$

$$Journal \leq_s Article,\ Equals = Equals$$

In the second case, where 'data' is a structure parameter and 'equals' is a value parameter, GPLS S_1 is a substructure of GPLS S_2. The GPLV V_1 defined by S_1 is a structural subtype of the GPLV V_2 defined by S_2 if, for all parameters in V_2, the corresponding value parameters in V_1 are equal and the corresponding structure parameters in V_1 are structural subtypes. All additional parameters in V_1 are ignored. In the following example we have two GPLVs defined by two GPLSs respectively:

$S_1 = [data{:}\ StructLim,\ equals{:}\ Method(data, data, Boolean), precedes{:}\ Method(data, data, Boolean)]$
$V_1 = [data{:-}\ Journal,\ equals{:-}\ Equals,\ precedes{:-}\ Older]$
$S_2 = [data{:}\ Struct, equals{:}\ Method(data, data, Boolean)]$
$V_2 = [data{:-}\ Article,\ equals{:-}\ Equals]$

Then $S_1 \leq_s S_2$ and $V_1 \leq_s V_2$ since the 'data' parameter's (structure parameter) value *Journal* is a substructure of *Article*, while the 'equals' parameters (value parameters) have the same value *Equals*. The 'precedes' parameter occurs only in S_1 and is ignored.

This second case need not have been explicitly defined, since it is implicitly defined by the first case with the aid of inclusion polymorphism. This case demonstrates how inclusion polymorphism works for the structural subtyping operation. The value defined by S_1 is included in the set of values defined by S_2.

4.1.3.3 The Structure *Gen*

Since all structures in the structure system are values, so too are the GPLSs. As values they are defined by a higher-order structure called *Gen* — short for Generic. *Gen* in turn is an element of the higher-order structure *Struct*. The assembly and relational operations for a value are defined in Section 4.1.3.1. In a metatype that defines generic types, *Gen* is used to specify the GPLSs which occur in them. The disassembly of a value (GPLS) is implicit, occurring through the use of the parameters in the specification of a generic type.

Gen is structurally equivalent only to itself and is its own structural subtype. Since there is only one structure, *Gen*, the translation operation is trivial: it is the identity translation. Examples of the concepts defined here are presented in Sections 4.3.4 and 4.4.5.

4.1.3.4 Generic Reference Structures

With the introduction of generic structures, a reference structure may be specified to a generic type and may also include a GPLV. A reference to a generic type specifies a set of values containing all instances from the generic type. For example, the structure

```
=>BinaryTree
```

is an object reference to the generic type *BinaryTree*. What follows is a reference to an implied type for a binary tree of journals sorted by age:

```
=>BinaryTree[data:- Journal, equals:- Equals, precedes:- Older]
```

Here, the reference to a generic type with a GPLV specifies a subset of the generic type's instances. Only those instances which have GPLVs that are subtypes of the GPLV used in the reference are elements of this structure. This structure allows us to be restrictive as to which instances of a generic type may be referenced.

Two references to generic types with GPLVs are structurally equivalent if they are defined by the same reference constructor, the types are equivalent and the GPLVs are equivalent. Structural subtyping is also defined for this structure. A reference to a generic type with a GPLV is a structural subtype of a reference to the generic type without the GPLV. For example,

$$=>BinaryTree\,[data{:-}\,Journal,\ equals{:-}\,Equals,\ precedes{:-}Older] \leq_s =>BinaryTree$$

since the first's type is a subtype of the second's type and the first's GPLV is a structural subtype of the second's.

4.2 Object System

The purpose of the object system is to provide persistent, independent values which are referenced uniquely and consistently, irrespective of time and medium. The object system defines the concept of an object, along with various operations for manipulating objects. We have defined the object system by building on the concepts defined in the structure system.

4.2.1 Objects

The values defined in the structure system are used to form objects. Bound to a value is a unique identity and together they form an object. An object is explicitly created by making use of the *New* operation.

New requires the structure of a value, it creates an object with a *Nil* value defined by that structure and assigns a unique identity to it, which is its result. The only way to access the value within the object is by specifying its identity. An object continues to exist until it is explicitly removed by using the *Delete* operation.

The *New* operation is also used to create objects at various different levels of persistence. A persistent object is defined as a level 0 object. Temporary objects are defined at positive integer value levels and only exist for the duration of the session. Operations are defined to convert objects from one level to another and to remove all objects from a specific level as well as the objects at higher levels. Levels are used for experimentation and give the user a mechanism for manipulating a set of objects, without it affecting the rest of the database.

The object system, as expected, is responsible for the consistency of the identities. Each time an object is created the identity associated with it must be unique. An identity is used to reference an object, and to this end, the object system must ensure that for every reference a valid object exists. A special identity called *Nil* is defined which represents the concept of a reference to no object. When an object is deleted, all references to it are set to Nil. The process of setting these references to Nil is called *annulling*. The context in which the reference is used will then know that no object exists for that reference. If the reference were maintained, then a dangling reference would occur and the database would be in an inconsistent state. An object O_1 may only hold a reference to another object O_2 provided the level of O_2 is equal or less than the level of O_1, so that when a level and all levels above it are cleared, there will be no dangling references.

When the *New* operation is used to create a level 0 object in the database, the object system is responsible for ensuring the persistence of the object. The object system transfers the object from disk to memory when it is required and updates the object on disk when the object is modified. The *Delete* operation is used to remove both temporary and persistent objects.

4.2.2 Object Reference Structures

In the structure system two reference structures were defined: object references and value references. An object reference structure is used to define a value which references an object. This value consists of the identity of the object being referenced. By using this structure, we can define values which reference objects and, hence, use it to set up relationships between objects. The set of values defined by the object reference structure consists of a set of object identities. An object reference structure is specified by the object reference constructor and a type. Only the identities of objects which are instances of this type or its subtypes are valid values for the structure. If a reference meets this requirement and the object exists or the Nil identity is used, then the reference is *correctly typed.*

The following value along with the identity *I* forms an object, which is an instance of type *Article*:

```
(Article, (author:- A, title:- 'Object...', periodical:- P))
```

An object reference to *Article* is denoted as follows:

```
=>Article
```

This declares a structure whose values are the identities of objects which are instances of type *Article*. *I* is a valid value for this structure. Through the identity *I*, the value within the object can be manipulated by various operations which are described next.

4.2.3 Operations

A set of operations for handling objects is defined in the object system. The *New* and *Delete* operations which allow for the creation and termination of objects have already been covered. The remaining operations deal with manipulating objects, their values and their identities. All operations on objects make use of the object identity and through it, access the object's value.

4.2.3.1 Equality Operations

The relational operations identical, shallow equality and deep equality, as described in Section 2.2.1.6, are defined in the model. *Identical* is denoted by the equal-dash-equal operator '=-='. As an example,

$$A =\!-\!= B \text{ iff } I = J$$

where *I* is the identity of *A* and *J* is the identity of *B*. The *shallow equality* operation is denoted by the equal-plus-equal operator '=+='. As an example, two distinct objects, *X* and *Y*, which are instances of *Article*, are shallow equal if their 'title' values are equal and the objects referenced by 'author' and 'periodical' are identical respectively.

$$X = (Article, (author:\!- A, title:\!- \text{'}Object...\text{'}, periodical:\!-P))$$
$$Y = (Article, (author:\!- B, title:\!- \text{'}Object...\text{'}, periodical:\!-Q))$$
$$X =\!+\!= Y \text{ iff } A =\!-\!= B, \text{'}Object...\text{'} = \text{'}Object...\text{'}, P =\!-\!= Q$$

Deep equality is denoted by the equal-star-equal operator '=*='. The two distinct objects, *X* and *Y*, which are instances of *Article*, are deep equal if the title values are equal and the objects referenced by 'author' and 'periodical' are deep equal respectively.

$$X = (Article, (author:\!- A, title:\!- \text{'}Object...\text{'}, periodical:\!-P))$$
$$Y = (Article, (author:\!- B, title:\!- \text{'}Object...\text{'}, periodical:\!-Q))$$
$$X =\!*\!= Y \text{ iff } A =\!*\!= B, \text{'}Object...\text{'} = \text{'}Object...\text{'}, P =\!*\!= Q$$

4.2.3.2 Operations

Identity, shallow and deep copy operations, as described in Section 2.2.1.6, and denoted by '<--', '<-+' and '<-*', respectively, are supported. After any of the above copy operations, the associated identical

or equality operation will return *true* when applied to the two operands of the copy operation. For example, after X <-+ Y, X =+= Y will yield *true*, while X =-= Y will yield *false* provided *X* and *Y* were not identical before the copy operation.

The identity copy operation requires the identity of an object and assigns it as the value of an object reference structure. The encapsulation of an object is maintained with this operation since only its identity is accessed. The shallow and deep copy operations are applied to objects by using their identities, which is normally obtained from an object reference. These two operations violate encapsulation since they access the values stored within the objects and they are not defined as methods in the object's type. The operations are however only defined if the type of the object on left (*X*) is a supertype of the object on the right (*Y*). Thus the resulting values of these operations will be correctly typed for their contexts.

4.2.3.3 Translation

In the type system (Section 4.3.3.1) the concept of translating an instance from a type to either the supertype or subtype is defined. Object translation is based on the translation defined in the structure system. When an object is translated, it affects all references to the object. These references must be checked to ensure that they are correctly typed and do not compromise the consistency of the object system. Since object translation of the object's value is covered by the structure system (Section 4.1.1.2), the remainder of this section deals with ensuring that the references to the translated object are correctly typed.

Assume that an object is translated to an instance of a subtype. Then all references to this object remain correctly typed due to inclusion polymorphism. All references to the translated object are typed either with the object's type or one of its supertypes. By translating the object to a subtype, the object is still an element of the type.

Object translation may also be used to convert an instance of a type to an instance of a supertype. There are three cases regarding the references to such an object to be considered. In the first case, an object of type T is translated to a supertype T'. All references to this object are checked, if the reference is typed by the supertype T', or perhaps one of its supertypes, then the reference is correctly typed and remains. Otherwise, the reference is invalid and is annulled. For example, assume an object of type *Proceedings* is to be converted to an object of type *Publication*. All references to the object which are typed by the following structures

```
=>Proceedings, =>Periodical, =>Edited Publication
```

are annulled, while all references to the object which are typed by the following structure

```
=>Publication
```

are valid.

In the second case, an object which is an instance of a generic type is translated to a generic supertype *G*. A reference to this object remains if it is defined by a generic type which is either a supertype of or equivalent to *G*. All other references to this object are annulled.

In the third case, the reference to the generic object is defined with a GPLV. Each instance of a generic type contains a GPLV which defines the implied type. If the GPLV in the object is a subtype of the GPLV in the reference, then the reference is correctly typed and remains. Otherwise the generic object is not an instance of the implied type, which is specified in the reference, so it is annulled.

4.2.3.4 Merge

The *Merge* operation is defined to take two objects and coalesce their values to form the merged object. The identities of the two objects are replaced by a new identity for the merged object. The references made to the original two objects are type checked and either set to the new identity or annulled. The operation returns the identity of the merged object.

When Merging two objects, the coalescing of the object values and the references to them are dealt with as follows. There are three cases to be considered. In the first case, both objects are instances of the same type. Thus the object values have the same structure and the first object is maintained while the second object is discarded. Before the Merge operation was performed the references were consistent. Since these two objects are instances of the same type all references will still be correctly typed. All references to the these two objects are replaced by references to the merged object and the database remains consistent.

In the second case, one object is an instance of type *T*, while the second object is an instance of a subtype of *T*. The object which is an instance of the subtype is maintained, while its identity becomes the identity for the merged object and forms the result of the operation. The object which is the instance of type *T* is discarded. All references to the subtype object are maintained, while all references to the discarded object are replaced by the identity of the subtype object. Since the references are being converted to a subtype they will be correctly typed.

The third case is where the two objects are instances of arbitrary types which are not related by the subtype relation. For this case there are two possibilities. In the first, the two types share a common subtype, through multiple inheritance. All structures defined in both types appear in the subtype, so values from the types are assigned appropriately to the subtype which becomes the merged object.

The second possibility is where no subtype exists for the two objects, in which case it is not possible to define a meaningful merge of the two objects. In this situation no merging occurs and the Nil identity is returned. The user may then explicitly cast the individual objects to a common supertype and then merge them, or may create a subtype and then merge the objects again.

4.3 Type System

The type system defines the concepts of instances, types, methods, subtyping and instantiation through the use of functions and constraints. The functions are defined over the domains and codomains of the structure and object systems. A set of constraints is defined for each concept, specifying the requirements of values which are classed under that concept. By using the apparatus defined in the structure and object systems, a specific model or exemplar is created which satisfies the constraints of the type system. The exemplar exhibits the features of an Object-Oriented Database. In an exemplar, each function maps to a specific set of operations which yield the results that are required by the constraints.

A type is an object and has a unique name. The behaviour of a type is defined by method objects, where each of their method structures has the type as its owner. The constraints specified in the following sections must be satisfied in order for the system to be consistent.

4.3.1 Instances

An entity in the real world is modelled in the database by an instance. The database consists of instances which are values as defined by structures in the structure system. An instance may also be an object, since it also consists of a value defined by a structure in the structure system (see Section 4.2.1). An *instance* is defined by a type, where *instantiation* is the process used to generate an instance from a type.

The instance-of relation denoted by the arrow symbol '⇨' exists between types and their instances. If A is an instance of type T, then this is denoted as follows:

$$A \Rightarrow T$$

The type function denoted by τ, determines from an instance, the type which instantiated it.

$$(A \Rightarrow T) \Rightarrow (\tau(A) = T)$$

The type constraint specifies that every instance must have a type.

4.3.2 Types

A type is used to abstract a concept in the real world and model it in the database. A type encapsulates and hides the structure and behaviour of a group of related entities. A *type* defines the structure and behaviour of its instances and its relationship to other types. Instances are encapsulated and may only be manipulated through the interface provided by the type.

For example, the concept of a conference proceedings may be modelled by a type. The proceedings type contains the structure for defining values that are specific proceedings. The *Proceed-*

ings type contains the behaviour of all proceedings, which consists of messages that may be passed to any proceedings instance. The *Proceedings* type may be defined as follows:

```
(Type,
   (name:- 'Proceedings',
   state:- Product(=>Type,
      (title: String, date: Date,
      publisher: =>Organization, editor: =>Person,
      contents: List((work: =>Paper, pages: Product(Integer, Integer)),
      issn: String,   conf-add: String, conf-date: Date))
   behaviour:- {Print, ...},
   objects:- {Proc1},
   subs:- {},
   supers:- {Refereed Publication}))
```

A type is also an object since it is required to persist and be referenced uniquely. In addition it must conform to the following constraints. Each type has a unique name used to reference it symbolically. (In a number of the examples, we use the name of the type to represent its identity.) Given a type, the name function, denoted by ν, returns a string which is the name of the type. The name constraint specifies that no two types may have names which are equal. For example, given the proceedings type *T*, ν will return the name of the type.

$$\nu(T) = \text{'}Proceedings\text{'}$$

The names of the types form part of name space which is used to identify objects.

Each type has a state which specifies the structure of its instances. A structure, as defined in the structure system, is stored in the type and its values form the instances of the type. Given a type, the state function, denoted by σ, returns the structure which defines the instances. The state constraint specifies that every type must have a state. For example, the journal type's state is defined by the following structure:

$$\sigma(Proceedings) = Product(=>Type, \left(\begin{array}{c} title: String, \\ date: Date, \\ publisher: =>Organization, \\ editor: =>Person, \\ contents: List\left(\left(\begin{array}{c} work: =>Paper, \\ pages: Product(Integer, Integer) \end{array} \right) \right) \\ issn: String, \\ conf\text{–}add: String \\ conf\text{–}date: Date \end{array} \right))$$

All proceedings have a title, date of publication, publisher, editor, issn and a list of papers that appear therein. A specific instance of a proceedings might have the following values:

```
(Proceedings, (title:- 'Proc. of the 1990 ACM SIGMOD Inter. Conf. on Management of Data',
    date:- 1/6/90,
    publisher:- Pub1,
    editor:- Person2,
    contents:- [(work:- Paper1, pages:- (1, 35)), ...],
    issn:- '2367-34578',
    conf-add:- 'Atlantic City, New Jersy',
    conf-date:- 4/6/90))
```

where *Pub1*, *Person2* and *Paper1* are object identities and *Proceedings* is the identity of the *Proceedings* type. Each type has behaviour associated with it. The behaviour consists of methods which define the actions of the entities being modelled. Since the structure of the instances is encapsulated and hidden by the type, the only way to manipulate the values is through the behaviour. Given a type, the behaviour function, denoted by μ, returns a set of method objects. The behaviour constraint specifies that every type has a behaviour. These objects are owned by the type. For example, the journal type has among its methods one called *Volume*.

$$\mu(Journal) = \{ Volume, \ldots \}$$

The method *Date* returns the date of publication. A message may be passed to any instance of type *Proceedings*, requesting it to perform the operations specified in the method *Date*. The variable *this* used in the method denotes the instance (receiver) to which the message is passed. The method obtains the second component of the given proceedings value by using the product subscript operator '!', which yields a record value. The attribute name 'date' is used to subscript the record and this value is returned by the method.

```
return(this!2.date)
```

Given an instance *Proc1* of type *Proceedings* with the value above, the message *Date* may be passed to it, yielding the following result:

```
Proc1.Date = 1/6/90
```

A type may be a direct subtype of another type (see Section 4.3.3). For each type, its subtyping relationship with other types is stored. Given a type, the supertype function denoted by π, determines its set of direct supertypes. Given a type , the subtype function, denoted by β, determines its set of direct subtypes. For example, type *Proceedings* is a direct subtype of type *Edited Publication*, thus the following relationship exists:

$$\beta(EditedPublication) = \{ Proceedings, \ldots \}$$
$$\pi(Proceedings) = \{ Edited\ Publication, \ldots \}$$

The subtyping constraint for a type specifies that each type must store its direct supertypes and subtypes. There is also the consistency constraint that specifies for any type and its direct subtype, that the following relationship must hold:

$$T \in \beta(U) \textit{ iff } U \in \pi(T)$$

The object constraint specifies that each type must also store a reference to all of its object instances. Given a type, the object function, denoted by θ, returns its set of objects. There is also an associated consistency constraint: consistency is to be maintained between this set and the set of instances which have this type as the result of their type function τ.

$$\tau(A) = T \textit{ iff } A \in \theta(T)$$

One type is equivalent to another type only if both types are identical, that is they are the same type.

$$S \equiv T \textit{ iff } S = T$$

4.3.3 Subtyping

While a type is used to model a concept, a subtype is used to model a specialization of that concept. An instance of a specialized concept is still an instance of the concept. For example, type *Periodical* models the concept of a publication which is published at regular intervals, say once a month. Associated with a periodical is its title, date of publication, publisher and issn. A journal is also a periodical since it is a publication which is published at regular intervals. The same information is associated with a journal, along with additional information such as the editor of the journal, its contents which is a list of papers, the volume and the issue number. A journal is a specialization of the periodical concept, this being modelled by defining type *Journal* as a subtype of type *Periodical*. The instances of type *Journal* are also instances of type *Periodical*.

Although a journal contains specialized data (volume and number), it still can be used as a periodical, since all of the information required for a periodical appears in it. As a result, any journal may also be manipulated by an operation defined for a periodical.

Subtyping as used in this model is similar to inheritance and the is-a relationship which is expressed between classes (types). With inheritance a subtype/subclass is defined by specifying that it inherits other classes. The system then generates the subtype/subclass accordingly. Subtyping, on the other hand, is a relationship that may exist between two types in the system. A type is defined on its own and if it satisfies the subtyping constraints, then it may be used as a subtype and may make use of the methods defined in the supertypes. The state structure of the subtype can be defined by extracting the type's state structure and adding components to it.

For one type S to be defined as a *direct subtype* of another type T, the following three conditions must be satisfied. First, the state of S is required to be a structural subtype of the state of T, that is,

$$\sigma(S) \leq_S \sigma(T)$$

The state structure specifies the structure of the instances. By requiring structural subtyping, the instances of the subtype and the methods of the type will enjoy inclusion polymorphism. This condition allows instances of subtype S to be classed as instances of type T. Second, the subtype function β applied to T must result in a set containing S. Third, the converse applies for the supertype function π. When π is applied to the subtype S, the resulting set must contain T . That is,

$$S \in \beta(T), T \in \pi(S)$$

If these three conditions are satisfied, then the direct subtyping relationship exists between these two types and is written as follows:

$$S < T$$

From the direct subtype relation, the more general subtyping order relation is defined and is denoted by '$\leq$'. Any type is clearly its own subtype, since a structure is its own structural subtype, and inclusion polymorphism of its operations is trivial. That is,

$$S \leq S$$

For two distinct types, one is the subtype of the other if a sequence of direct subtypes exists between them. This allows for the transitivity of the subtyping relation. Thus, a non-direct subtype will also be able to enjoy the property of inclusion polymorphism. We have that

$$S < T \Rightarrow S \leq T$$
$$S < T, T < U \Rightarrow S \leq U$$
$$S < T_1, ..., T_n < U \Rightarrow S \leq U$$

The messages defined in type U may be passed to the instance of type S. Type S inherits the messages defined in its supertype — the messages in type U are inclusion polymorphisms for instances defined by type S.

Type *Journal* is a direct subtype of type *Refereed Publication* and the following conditions exist:

$$\sigma(Journal) \leq_S \sigma(RefereedPublication), Journal \in \beta(RefereedPublication),$$
$$RefereedPublication \in \pi(Journal)$$
$$\Rightarrow$$
$$Journal < RefereedPublication$$
$$\Rightarrow$$
$$Journal \leq RefereedPublication$$

The subtyping relation gives rise to a directed acyclic graph (DAG) called the subtyping hierarchy. The nodes in the graph are the types and a directed edge is drawn from subtype to type. Only the edges

between direct subtypes are drawn as the other edges are redundant. This hierarchy is also called the is-a hierarchy or an inheritance hierarchy.

4.3.3.1 Instance Translation

An instance of type *T* may be translated to an instance of type *S*, provided the types are related by subtyping. The instance of *T* is cast to the state structure of *S* by using the operations defined in Sections 4.1.1.2 and 4.2.3.3. The consistency constraint for translation specifies that the identity of the object must be removed from the 'objects' attribute of type *T* and added to that of type *S*. The type constraint for the object must also reflect this change.

4.3.4 Generic Types

A type is used to model a concept, but requires a complete specification of the values and methods it uses. A generic type is used to model a more general concept, one that defines a generalized instance and behaviour. The instance contains information pertaining to the concept and may also contain information which is used by the concept, but which is arbitrary. The behaviour and structure of the concept is the same regardless of this arbitrary information.

A generic type allows for the abstraction of the values used in the type — the arbitrary information. The abstraction takes the form of generic parameters. A generic type requires a GPLS (generic parameter list structure), which defines its generic parameters and their typing. These parameters may be used freely within the type or its methods, instead of specific values. By specifying a GPLV (generic parameter list value), an implied type is specified for the generic type. An *implied type* is the generic type with all parameters replaced by the values in the GPLV. The implied type resembles an ordinary type.

The concept of an Integer binary tree sorted in ascending order may be modelled as a type, where specific binary trees are instances of this type. A binary tree contains sorted data, but regardless of the data being stored in the tree, its structure and logic remains the same. A generic type is used to model the generalized concept of a binary tree, where the type of data and the operations used to sort and compare the data is abstracted to generic parameters.

If we defined a binary tree as an ordinary type, we would be required to specify the structure of the data that is being stored in the trees. For each different data structure, we would be required to specify a different type, which is inefficient. By using a generic type, we only specify the type once.

An implied type conceptually defines a type, but no physical type is created. The generic type makes use of the GPLV and whenever a generic parameter occurs, the corresponding value in the GPLV is substituted for it. For example, a GPLV is specified with the *Integer* structure, equality and ascending operations, which together with the generic Binary tree type form an implied type. The implied type is the same as the Integer binary tree sorted in ascending order which would otherwise have been defined. The *Binary Tree* type may be defined as follows:

```
(TypeGen,
    (name:- 'BinaryTree',
    state:- Product(=>TypeGen,
        (value: data,
        left: =>BinaryTree[data:- data, Equals:- Equals, Precedes:- Precedes],
        right: =>BinaryTree[data:- data, Equals:- Equals, Precedes:- Precedes])),
    behaviour:- {Add, First, Last, Next, Previous, ...},
    objects:- {Tree1, Tree2, Tree3},
    subs:- {},
    supers:- {InstGen},
    generic:- [data: Struct, Equals: Method(data, data, Boolean),
        Precedes: Method(data, data, Boolean)]))
```

The parameters in the GPLS (which appear in the 'generic' attribute) are typed by structures. When a parameter is used in a generic type, its context is type checked against its GPLS structure. When a GPLV is specified, the values associated with the parameters are also type checked against the GPLS parameter structures.

The methods defined for a generic type may make use of these generic parameters. The logic of the method remains the same, but the value which is manipulated will depend upon the implied type. These methods are called parametric polymorphisms, because they are defined for a number of different types and require a GPLV to parameterize the type, and hence the method, when the message is used.

A generic type has all of the constraints that an ordinary type has, plus one additional constraint. Given a generic type, the generic function, denoted by γ, returns the GPLS. Each generic type must have a GPLS, which is the result of γ. For example,

$$\gamma(BinaryTree) = \begin{bmatrix} data\text{: } Struct, \\ equals\text{: } Method(data,data, Boolean), \\ precedes\text{: } Method(data, data, Boolean) \end{bmatrix}$$

An implied binary tree type of integers, sorted in ascending order is specified as follows:

```
BinaryTree[data:- Integer, equals:- Equals, precedes:- LessThan]
```

This is a generic type followed by a GPLV. The data parameter in the state is replaced in the generic type by a value reference to type *Integer*. The equals and precedes parameters used in the methods to manipulate the tree are replaced by the *Integer Equals* operation and the *Less-Than* operation, respectively.

The following object, *Tree1* is an instance of the implied binary tree above:

```
(BinaryTree[data:- Integer, equals:- Equals, precedes:- LessThan],
    (value: 5, left: Tree2, right: Tree3))
```

The instances of a generic type may only be object instances. An object stores its instantiating type, which includes the GPLV for defining the implied type. This is the first component in the object above. For value instances, the context is required to be typed by a reference containing a GPLV, since the value instance relies on its context for its typing (see Section 4.3.8 for the differences between objects and values). The value for this instance is as follows:

```
(value: 5, left: Tree2, right: Tree3)
```

As can be seen, the information for the implied type is not stored in the value and the context is required to store this information. The context must be typed with the following value reference structure:

```
BinaryTree[data:- Integer, equals:- Equals, precedes:- LessThan]
```

If the value was typed with the following value reference structure, there would be no way of determining the implied type.

```
BinaryTree
```

4.3.5 Metatypes

A concept is modelled by a type and a specific case of that concept is modelled by an instance. If we consider each specific type to be an instance of the concept called a *type*, then we model the concept of a *type* by means of a type called a *metatype*. A type defines the structure and behaviour of its instances. If we allow a type to be an instance in the model, then since a metatype is a type and thus an instance, it is an instance of another type called a meta-metatype. This process can continue ad infinitum. A type which produces an instance which is not a type is called an *ordinary type*.

The type function τ may be applied to any instance and will return the instantiating type. If the instance is a type, then it returns a metatype. For any type, the instance constraint and the type constraints must be satisfied. The system is reflective because it consists of instances which are defined by types, but types are themselves instances. This allows the structure of the instances and types to be specified within the model, provided they adhere to the constraints.

As described in Section 4.3.1, the process called instantiation generates an instance from a type. Given an instance A of type T, this is denoted as follows:

$$A \Rightarrow T$$

If T itself is an instance of a metatype M and M is an instance of a meta-metatype MM, we have

$$T \Rightarrow M \Rightarrow MM$$

This instantiation relation '⇨' generates an instantiation hierarchy. The leaves of the hierarchy consist of ordinary instances, above which are the ordinary types, followed by the metatypes, meta-metatypes,

etc. This hierarchy can also be depicted by a graph where the nodes are instances (this includes types) and a directed edge is drawn from an instance to its instantiating type.

4.3.6 Methods

The behaviour of a type is defined by a persistent set of methods. Each method has a name, source code and a compiled version which corresponds to a method value (as defined in Section 4.1.2.2). A method is defined as an object, which is an instance of a type called a *method type* and may be defined as follows:

```
(TypeGen,
    (name:- 'Method',
    state:- Product(=>TypeGen,
        (name: String,
        source: String,
        code: Method(Owner, Param, Result))),
    behaviour:- {link, compile, edit, load, write},
    objects:- {Equals, New, NewType, ...},
    subs:- {},
    supers:- {InstGen},
    generic:- [Owner: Ref, Param: StructLim, Result: StructLim]))
```

A method being an object facilitates the persistence and referencing of unique methods, even if the names of some methods are the same. The behaviour of a type consists of a set of references to these method objects. The state structure of a method type defines an instance which contains a name and source code defined by strings and the method value defined by a method structure. The *Equals* message for journals is an instance of the state structure above and is defined as follows:

```
X = (Method[Owner:- Journal, Param:- Journal, Result: Boolean],
    (name:- 'Equals', source:- 'return(this = param);', code:- C))
```

where *C* is the executable code for the method. *X* is an object which is an instance of method type *Method*. This is a generic type which will be explained in more detail in Section 4.4.6. A method type is the same as any other type, but is specialized to define instances which contain method values and may be passed as messages in the system.

A message pass is denoted by an instance of a type, followed by the name of a message and perhaps a parameter. Each method object in the system is symbolically referenced through its name and each method is owned by a type. A method defined in a type may be redefined in a subtype with the same name (overriding), but with specialized operations. Given a message pass, a look-up procedure (dispatching) is required to bind the symbolic reference with a particular method, see Section

5.4.3.2 for an example. The messages of methods defined in supertypes may be passed to instances of types, due to inclusion polymorphism.

We have defined the look-up procedure to bind that method which is 'most' specialized to the message pass. Given the subtyping hierarchy, with multiple supertypes, a method may be defined and redefined at various levels. The look-up procedure finds the method which is the nearest (in terms of nodes in the hierarchy) to the instance's type. This amounts to searching the hierarchy in a breadth-first manner. The process continues until either the correct method is found or all types including the root of the subtyping hierarchy have been searched. In the latter case, a method does not exist for the message pass which is thus invalid.

Once the method object with the correct name is found, its structure is obtained from the state structure of its method type. The method structure is type checked against the message pass. Type checking involves checking the instance to which the method is passed and the structure of the parameter value. If the type check is satisfied, then the message may be passed.

There are three functions associated with method types and their instances. The method name function, denoted by ν_m, given a method object returns the name of that method. For example, the name function applied to the method object above will return the string *Equals*. That is,

$$\nu_m(X) = Equals$$

This function is used to obtain the name which identifies the method. The name constraint specifies that every method has a name and within the set of methods owned by a single type, all method names are unique.

Given a method type, the method structure function, denoted by σ_m, returns the method structure. The method structure constraint specifies that every method type must define a method structure. This constraint is used to perform type checking. Given a message pass, all methods in the instance's type are searched. If one is found with the correct name, then its type is located and the method structure function is used to obtain the structure of the method value. This structure is then compared to the message pass in order to validate it. In the following example, by using this function on the method type *Method*[...], we obtain the structure defining the method *Equals*. That is,

$$\sigma_m\,(Method[...]) = Method(Journal, Journal, Boolean)$$

Given a method object, the code function, denoted by κ, returns the method value stored in the object. Provided the message pass is valid, this value (message) may then be passed to an instance defined by the owner type. This function is similar to the method structure function which determines the structure of a method, whereas this function determines the value for that structure. In the following message pass, where *J1* and *J2* are journals, the code function is used to obtain the code.

```
J1.Equals(J2)
```

$$\kappa(Equals) = C$$

This code can then execute to perform the action of the method.

4.3.7 Base types

There exists in the model a set of ordinary types called base types. These are used to define the primitive values and operations from which more complicated structures, values and operations may be defined. *Integers* and *Strings* are examples of base types. There are two parts to a base type: first is the representation for the values, and second are the operations. Base structures are used to specify which class of representation the instances of the base type requires. A base value may either be fixed or dynamic and there are corresponding memory representations for each.

All of the operations for a base type are implemented externally and then linked into the system. The operations manipulate the values as required, but maintain the memory representation required for that particular base structure. Each operation for a base type is stored in the same way as methods are stored. A method object contains the name of the operation and a link to the external code. The method structure which types this operation has an owner structure which is a value reference to the base type. The operation is then used as a message in exactly the same way as any other method.

The *Integer* base type requires a fixed value of four bytes. The state structure of the *Integer* type is the following:

```
Product(=>Type, Base[4])
```

An *Integer* object contains a reference to its type and a base value of four bytes. An *Integer* value instance is defined only by the second component of the above structure. That is,

```
Base[4]
```

All operations defined for *Integers* are implemented for a representation making use of four bytes. The addition operation is defined to accept two integer values of four bytes each, add them together and return a result of four bytes. The method structure for *Addition* is as follows:

```
Method(Integer, Integer, Integer)
```

As far as the system is concerned, a four byte value is being used. No extra information regarding the value is assumed by the system. All base type methods are specified by value references because only the value defined by the base structure may be used by the external implementation. The structures which define the complex values in the system and the structure of objects is hidden. The interface for base values are the values defined by the base structures.

The set of base types in the model is totally extensible and shares the same status as any other type. Base types may have subtypes which are also base types. Additional operations may be linked

to the subtype, while methods may also be defined for the subtype. The correctness of a base subtype is the responsibility of the base type implementer. The system provides the facility to perform base type subtyping, but there is no way in which it can check its validity.

4.3.8 Objects and Values

A type produces instances which are classified into value instances and object instances. An object instance is an object defined by the state structure of its type. Being an object makes the instance an independent entity referenced through its identity. In order to manipulate an object we need to know its structure. Since the object is independent, there is no typed context from which to obtain this information. As a result, each object stores within it a reference to its instantiating type, from which the structure of the object is obtained. This reference to the instantiating type forms part of the state structure in the type and thus also forms an integral part of the object instance. The object reference structure defines a value consisting of the identity of an object instance, which is an instance of the type used in the reference structure. For example, state structure of the *Proceedings* type is defined as follows:

$$\sigma(Proceedings) = Product(=>Type, \left(\begin{array}{c} title: String, \\ date: Date, \\ publisher: =>Organization, \\ editor: =>Person, \\ contents: List\left(\left(\begin{array}{c} work: =>Paper, \\ pages: Product(Integer, Integer) \end{array} \right)\right) \\ issn: String, \\ conf\text{–}add: String \\ conf\text{–}date: Date \end{array} \right))$$

A proceedings object is defined as

```
(Proceedings, (title:- 'Proc. of the 1990 ACM SIGMOD Inter. Conf. on Management of Data',
    date:- 1/6/90,
    publisher:- Pub1,
    editor:- Person2,
    contents:- [(work:- Paper1, pages:- (1, 35)), ...],
    issn:- '2367-34578',
    conf-add:- 'Atlantic City, New Jersy',
    conf-date:- 4/6/90))
```

where the first component contains a reference to the instantiating type — *Proceedings*. The proceedings object above has the identity *I* which may be used as the value for an object reference to type *Proceedings*, that is,

$$I \in =>Proceedings$$

A value instance on the other hand is a value defined by a portion of its type's state structure. A value instance is always used in a typed context, such as a component in another instance. From this context, the structure of the instance is obtained and hence the value instance may be manipulated. As stated above, part of the state structure for any type defines a reference to the instantiating type. This reference is redundant in a value instance because this information is always available from the context in which the value instance is used. For this reason, a value instance is defined by a portion of its type's state structure. The portion which is excluded is that which defines the reference to the instantiating type. The value reference structure defines a value which is a value instance of the type used in the reference structure.

Given the state structure of a type, the value function, denoted by λ, yields the portion of it which defines a value instance. The state structure for type *Proceedings* is defined above. This structure consists of a product with two components, the first component defines the reference to the instantiating type and the second component defines the remainder of the instance. The value constraint removes the product structure and yields the second component.

$$\lambda(\sigma(Proceedings)) = \left(\begin{array}{c} title\text{: } String, \\ date\text{:}Date, \\ publisher\text{: } =\!>\!Organization, \\ editor\text{: } =\!>\!Person, \\ contents\text{: } List\left(\left(\begin{array}{c} work\text{: } =\!>\!Paper, \\ pages\text{: } Product(Integer, Integer) \end{array} \right) \right) \\ issn\text{: } String, \\ conf\text{–}add\text{: } String \\ conf\text{–}date\text{: } Date \end{array} \right)$$

A value instance of type *Proceedings* might be specified as follows:

(title:- 'Proc. of the 1990 ACM SIGMOD Inter. Conf. on Management of Data',
date:- 1/6/90,
publisher:- Pub1,
editor:- Person2,
contents:- [(work:- Paper1, pages:- (1, 35)), ...],
issn:- '2367-34578',
conf-add:- 'Atlantic City, New Jersy',
conf-date:- 4/6/90)

which is an element of the value reference structure to type *Proceedings*.

The value function may also be used on an object instance to obtain a value instance. This effectively converts an object instance into a value instance. The owner of a method structure requires a reference structure. This determines the structure of the value to which the message is passed. If a method is defined by an object reference structure, then the message may only be passed to an object. The method for this structure is defined over the type's entire state structure. But for a value reference

structure, the message may be passed to both object and value instances. In the case of an object instance, it is converted into a value instance by using the value function.

All methods which are defined externally and linked into the system are defined only for base structure values. The base type constraint specifies that the state structure of base types is defined so that their value portion is a base structure. The state structure of the *Integer* base type has the following form:

```
Product(=>Type, Base[4])
```

The first component is a reference to the instantiating type and the second component is the base value. The value portion of this value is defined by:

$$\lambda(Product(=>Type, Base[4])) = Base[4]$$

A value instance of this type is defined by the base structure. When a message is passed to an *Integer* instance, if it is a value instance, then it only consists of the base value which is used by the linked operation. If the instance is an object, then it is referenced through its identity, which is used to obtain the object value. The value function is used to obtain the value portion which is defined by the base structure. The base value is then used by the linked operation.

The state of the *Journal* type contains a value reference to type *Integer* for attribute 'volume'. In an instance, a value instance will be stored in the 'volume' attribute. This corresponds with just the integer value defined by the base structure. The object reference to its instantiating type, *Integer*, is not included, this information being obtained from the structure of the *Journal* instance.

4.3.9 Practical Considerations

Ideally, we would like the system to be totally reflective: the constraints and functions which define the type system are defined by values and operations within the system, and hence the system defines itself. The state function σ for example can be defined as a method which will locate the structure in a type. This method would be defined in the metatype that defines types.

This ideal can not be attained, because the methods used in the system are treated as instances on the same level as all other instances. In order to manipulate an instance, a message must be passed to it. Thus, in order to manipulate a method, a message must be passed to it.

When evaluating a message pass, the look-up procedure is used to obtain the correct method. Having obtained the correct method, it is disassembled to yield the result of the message pass. The look-up process relies on a number of functions. The type function τ is used to obtain the type of the instance and also the type of a method object in the behaviour. The behaviour function μ is used to obtain the set of methods defined for a type. The various functions associated with methods are used to obtain a method's name, structure and value. These functions cannot be implemented as methods because in order to pass a message, these methods would have to be evaluated first. But in order to

evaluate each of them, the look-up procedure would be required to find them. A recursive situation arises with the effect that no method will ever get evaluated. For one method to be evaluated, another method needs to be executed, which in turn requires a method to be evaluated.

For this reason, the constraints and functions need to be implemented on top of the system and not from within it. The message passing process is external to the system, defined by special operations (not methods) which implement the functions. The system still requires the structure specification of where the values associated with each function are stored. This is a reflective specification defined within the types, but the operations used to retrieve this information are specified as special operations and not as methods. All of the functions are implemented in the system in this manner. By using these functions, there is an external view to the type system which can check its integrity independently.

4.4 Exemplar

The structure and object systems provide the apparatus and the type system provides the constraints for creating an exemplary model. An exemplar is an interpretation or a realization of the type system, where a collection of values and objects satisfy the constraints and, hence, define an Object-Oriented Database. We present values and operations which are used to define instances, types, base types, generic types, methods and the reference list example. The information required by the functions in the type system is stored in an instance and is specified by the state structure in the type. For each function, an operation is specified which will access this information. As stated above, these functions are not implemented as methods, but as special operations.

4.4.1 Instance

Type *Instance* defines all instances in the database. Every type in the database is its subtype since all types are required to produce instances; consequently it forms the root of the subtyping hierarchy. The data required by the type function τ, which is associated with every instance, is defined for type *Instance*. The state consists of a product structure containing an object reference to type *Type* in the first component and a Nil structure in the second component. The first component is defined to store in the instance, its instantiating type. In all other types the second component is used for the values stored by the instance. In this type, there are no other values to be stored, so the Nil structure is used. There is no specific behaviour defined for this type as its primary use is to define the structure of instances.

```
(Type,
    (name:- 'Instance',state:- Product(=>Type, Nil),
    behaviour:- {}, objects:- {},
    subs:- {Type, InstGen, InstBase, Publication, Article, RefList}, supers:- {}))
```

The type function τ requires an object reference to an instance and returns the first component. That is it returns the object referenced by *Type*.

$$\tau = this!1$$

4.4.2 Type

The concept of a type is defined by the metatype called *Type*, which defines all of the structures required by the type functions in order to satisfy the type constraints. Because a type is required to be an instance, type *Type* is a subtype of type *Instance*, and hence it appears in the 'subs' field of the instance above. Thus, it produces types and they are instances. A type is a product value where the first component is a reference to its instantiating type (a metatype), and the second component contains the information pertaining to the type. This product structure is "inherited" from type *Instance*, as indicated by *Instance* appearing in the 'supers' field below. Type *Instance* is an instance of type *Type*, and this is reflected in the first component of type *Instance* above.

The type *Type* defines all types and since it is a type, it also defines itself. Thus, *Type* is its own instance and forms the root of the instantiation hierarchy. All of its instances are types and all of its subtypes are metatypes. The first component of type *Type* contains a reference to itself because it is its own instance, which means that *Type* also appears in its own objects field.

```
(Type,
    (name:- 'Type',
    state:- Product(=>Type,
        (name: String,
        state: Struct,
        behaviour: Set(=>Method),
        objects: Set(=>Instance),
        subs: Set(=>Type),
        supers: Set(=>Type))),
    behaviour:- {NewInstance, NewType, ...},
    objects:- {Type, Instance, InstGen, TypeGen, Base, Ordinal, Range,
                Publication, Article, RefList, ...},
    subs:- {TypeGen, Base},
    supers:- {Instance}))
```

The data associated with the name function which is used to reference the type, is defined by the name attribute, that is,

$$\nu = this!2.name$$

The data associated with the state function, which is used to define its instances, is defined by the state attribute. Given a type, σ returns the structure which is found in the type and is used for the instances of the type, that is,

$$\sigma = this!2.state$$

The value function specifies which portion of a state structure constitutes the value component. It is used to obtain the second component of the product structure in a type, namely,

$$\lambda = this!2$$

This function operates on types to obtain the structure for a value instance. A corresponding value function for instances is used to convert object instances into value instances, by obtaining the second component of the product value. The data associated with the remaining functions corresponds to the attributes in the record.

The methods defined for this type are passed to its instances (types). These methods are used to create object instances (*NewInstance*) and new types (*NewType*), as well as to ensure the consistency of the system by storing a reference to the type in the instance and adding a reference to the instance to the type's set of objects. The method *NewInstance* is defined below:

```
(Method[Owner:- =>Type, Param:- Nil, Result:- =>Instance],
   (name:- 'NewInstance',
   source:-'
      =>Instance: Obj;
      |/
         Obj <-- this!2.state.new;
         Obj!1 <-- this;
         this!2.objects.ins(Obj);
         return Obj;
         /|'
   code:- ...))
```

The method declares a variable 'Obj', which is an object reference to type *Instance*. An object is created with the *new* primitive message defined for structures, by passing it to the structure of *Type*. The identity of the new object is stored in 'Obj'. The reference to the instantiating type is set in the next line, by storing the identity of the type in the first component. This is followed by the insertion of the object's identity into the set of objects held by the type. The method then returns the identity of the object.

The *NewType* method makes use of this method to create a new type as an instance. It then obtains from the user a name, state structure and subtyping requirements. Provided the name is unique and the subtyping constraints are satisfied, the type is created. The method then updates the corresponding sets of subtypes and supertypes to ensure the consistency of the types.

4.4.3 Base Types

It may be helpful to refer to Figure 4.1, which depicts the subtyping and instantiation hierarchies for base types, while reading this section. (See Section 3.2.8 for an explanation on how to read the figure).

The database contains various base types, including strings and integers. The instances of these types are base values such as integers and strings. The metatype *Base*, which is a subtype and an instance of *Type*, is used to define all base types. It defines the structure of base types and contains the methods which are passed to them.

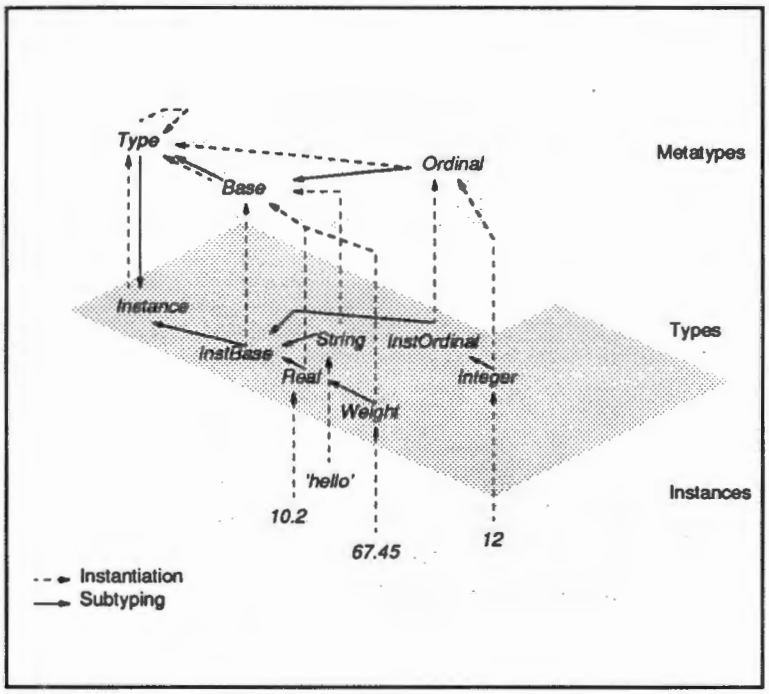

Figure 4.1: Hierarchy of Base Types

```
(Type,
    (name:- 'Base',
    state:- Product(=>Base,
        (name: String,
        state: Struct,
        behaviour: Set(=>Method),
        objects: Set(=>InstBase),
        subs: Set(=>Type),
        supers: Set(=>Type))),
    behaviour:- {LinkBase},
    objects:- {InstBase, String, Real, Weight},
    subs:- {Ordinal},
    supers:- {Type}))
```

The type *InstBase* is a subtype of *Instance* and an instance of *Base*. It defines the general structure of base type instances and the methods which may be passed to them. The *Real* and *String* types are among its subtypes.

```
(Base,
    (name:- 'InstBase',
    state:- Product(=>Base, Nil),
    behaviour:- {...}
    objects:- {}
    subs:- {InstOrdinal, String, Real},
    supers:- {Instance}))
```

The type *String* is defined as an instance of *Base* and as a subtype of *InstBase*.

```
(Base,
    (name:- 'String',
    state:- Product(=>Type, Base[-]),
    behaviour:- {Concat, Len, Copy, Ins, Pos, Del, ...},
    objects:- {...},
    subs:- {...},
    supers:- {InstBase}))
```

The type's state is defined by a product which has been "inherited" from *Instance* via *InstBase*. The second component is the dynamic base value used for strings. The dynamic base structure is a specialization of the Nil structure used in *InstBase*. All methods written for *String* are defined over this dynamic base value. For example, the concatenate method takes two string values and appends the one to the end of the other. Concatenate is implemented externally and linked into the system through the *Concat* method. The method requires two string values and returns a string value.

An ordinal type is a base type for which the following operations and values are defined: first, last, predecessor, successor, value and position. The *Integer* type is an ordinal type since it has these properties. Its first and last values are the smallest and largest integers which can be represented in the implementation. The predecessor is defined by decrementing the value by one, while the successor operation is defined by incrementing the value by one. The value and position operations are the identity functions in the case of integers. The metatype *Ordinal* is a subtype of the metatype *Base* and all ordinal types are instances of it. All operations which are performed on ordinal types are defined in this type.

```
(Type,
    (name:- 'Ordinal',
    state:- Product(=>Ordinal,
        (name: String, state: Struct, behaviour: Set(=>Method),
        objects: Set(=>InstOrdinal), subs: Set(=>Type), supers: Set(=>Type))),
    behaviour:- {First, Last, Value}, objects:- {InstOrdinal, Integer},
    subs:- {}, supers:- {Base}))
```

The type *InstOrdinal* is an instance of *Ordinal* and defines all ordinal instances. It is a subtype of *InstBase*, since it produces base instances. The general operations which are defined for ordinal instances are defined here.

```
(Ordinal,
    (name:- 'InstOrdinal',
    state:- Product(=>InstOrdinal, Base),
    behaviour:- {Succ, Pred, Pos},
    subs:- {Integer}, supers:- {InstBase}))
```

The *Integer* type is defined as an ordinal type, is a subtype of *InstOrdinal* and is an instance of *Ordinal*. All of the operations which are defined for it are linked into the system. Each operation is defined as a method object which is owned by type *Integer*.

```
(Ordinal,
    (name:- 'Integer',
    state:- Product(=>Type, Base[4]),
    behaviour:- {Negate, Add, Sub, Mult, Expo, Mod, Rem, Real, EqualLessThan,
            EqualGreaterThan, Succ, Pred, Pos, ...},
    objects:- {}, subs:- {Age}, supers:- {InstOrdinal}))
```

4.4.4 Range

A type defines a set of values which, in some cases, might be ordered. It then makes sense to talk about a subset of values, which form a range as specified by a first and last value. A value is contained in the range if its position in the ordering lies between the first and last values.

The metatype *Range* is used to define types which store ranges of values. The state structure defines the structure of a type, but with two additional attributes. The 'first' and 'last' attributes are used to store the first and last values of the range. These values are however defined by the state structure of the type. The attributes are typed with the *Value* structure, which specifies their values are defined by the state structure of an instance of type *Range*. Thus the values for these two attributes are typed by the value for the 'state' attribute.

```
(Type,
    (name:- 'Range',
    state:- Product(=>Range,
        (name: String, state: Struct, behaviour: Set(=>Method),
        objects: Set(=>Instance), subs: Set(=>Type), supers: Set(=>Type),
        first: Value,
        last: Value)),
    behaviour:- {Contains}, objects:- {Age},
    subs:- {}, supers:- {Type}))
```

The *Contains* method checks whether or not a value is in the range. The method is owned by *Range* since it is passed to its instances, which are types that define ranges of values. The parameter of the method consists of the value whose containment is being determined. The result of the method is a Boolean value. The method relies on the fact that for any type which has ordered values, the methods *EqualLessThan* and *EqualGreaterThan* are defined.

```
(Method[Owner:- =>Range, Param:- Value, Result:- Boolean],
    (name:- 'Contains',
    source:-'
        |/
            return(this!2.first.EqualLessThan(param).and(this!2.last .EqualGreaterThan(param)));
            /|'
    code:- ...;))
```

As an example of a range, the age of a person might be defined by an integer value between 0 and 200. Type *Age* is defined to model the ages of people and is an instance of *Range*. It is a subtype of *Integer*, since it defines a range of integer values. The value instances are typed by a base value of four bytes. This structure is used to type the values stored in the 'first' and 'last' attributes. In the metatype *Range*, these values are typed by the *Value* structure.

```
(Range,
    (name:- 'Age',
    state:- Product(=>Age, Base[4]),
    behaviour:- {young},
    objects:- {A},
    subs:- {},
    supers:- {Integer},
    first:- 0,
    last:- 200))
```

The *Contains* message may be passed to type *Age* with the parameter 5, to determine if 5 is contained in the *Age* range. Because 5 is between 0 and 200, the result is true.

```
Age.contains(5) = true
```

The hierarchy associated with range types is illustrated in Figure 4.2.

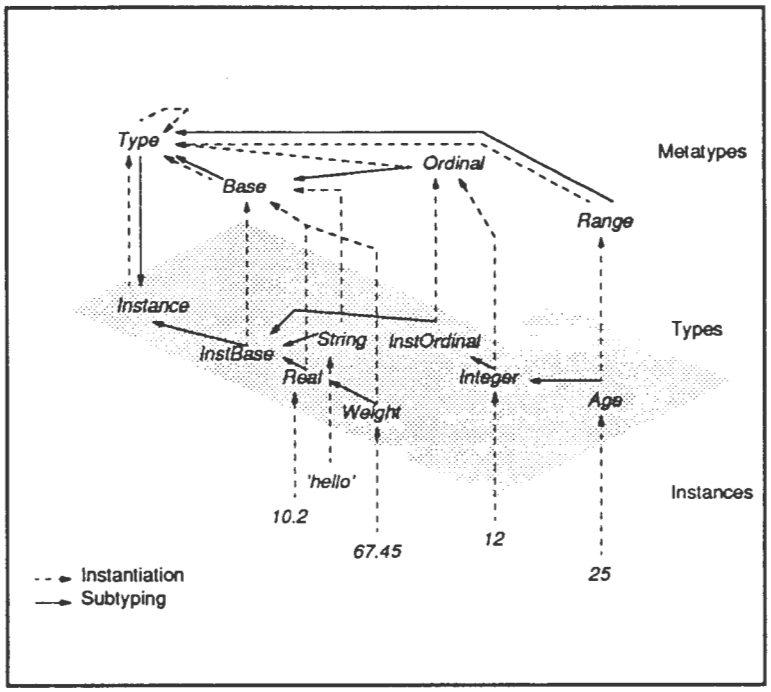

Figure 4.2: Hierarchy of Range Types

4.4.5 Generic Types

A generic type requires the generic constraint in addition to all of the type constraints. A type called *TypeGen* is used to define all generic types in the same way that *Type* defined all types. *TypeGen* is a subtype of *Type* and adds to the record structure the attribute 'generic', typed with the *Gen* structure. All instances of *TypeGen* are types and contain a GPLS in the 'generic' attribute. This attribute stores the data required by the generic function γ. The parameters defined in the GPLS may be used in the state structure or in the methods which appear in the behaviour.

```
(Type,
    (name:- 'TypeGen',
    state:- Product(
        (name: String, state: Struct, behaviour: Set(=>Method), objects: Set(=>Instance),
        subs: Set(=>Type), supers: Set(=>Type), generic: Gen)),
    behaviour:- {},
    objects:- {Method}, subs:- {},
    supers:- {Type}))
```

The type *InstGen* defines all generic instances and contains all of the methods passed to generic instances. It is a subtype of *Instance* and an instance of *Type* (see Figure 4.3). *InstGen* specializes the instantiating type to *TypeGen*.

```
(Type,
    (name:- 'InstGen',
    state:- Product(=>TypeGen, Nil),
    behaviour:- {},
    objects:- {},
    subs:- {Method, BinaryTree},
    supers:- {Instance}))
```

A generic type is defined as a subtype of InstGen and is an instance of the metatype TypeGen. The binary tree example used earlier is defined as follows:

```
(TypeGen,
    (name:- 'BinaryTree',
    state:- Product(=>TypeGen,
        (value: Data,
        left: =>BinaryTree[Data:- Data, Equals:- Equals, Precedes:- Precedes],
        right: =>BinaryTree[Data:- Data, Equals:- Equals, Precedes:- Precedes])),
    behaviour:- {Add, First, Last, Next, Previous, ...},
    objects:- {Tree1, Tree2, Tree3},
    subs:- {},
    supers:- {InstGen},
    generic:- [data: Struct, Equals: Method(data, data, Boolean), Precedes: Method(data, data, Boolean)]))
```

Notice that the state of the type specifies a value whose structure is defined by the 'Data' parameter. The 'Equals' and 'Precedes' parameters store methods and are used in the methods owned by *BinaryTree* to sort and compare the data values. The left and right subtrees are defined as object references to BinaryTrees. The GPLV used in the reference specifies that the reference is to a binary tree which has the same parameter values as this binary tree. This prevents a binary tree of *Integers* from having a binary tree of *Publications* as a subtree.

For example, *Tree1*, *Tree2* and *Tree3* are *Integer* binary trees, sorted by *LessThan* and compared by *Equals*. *Tree1* has the following value:

```
(BinaryTree[data:- Integer, equals:- Equals, precedes: LessThan],
    (value:- 5, left:- Tree2, right:- Tree3))
```

Tree2:

(BinaryTree[data:- Integer, equals:- Equals, precedes: LessThan],
(value:- 1, left:- Nil, right:- Nil))

Tree3:

(BinaryTree[data:- Integer, equals:- Equals, precedes: LessThan],
(value:- 8, left:- Nil, right:- Nil))

All of the generic types and their relationships are illustrated in Figure 4.3.

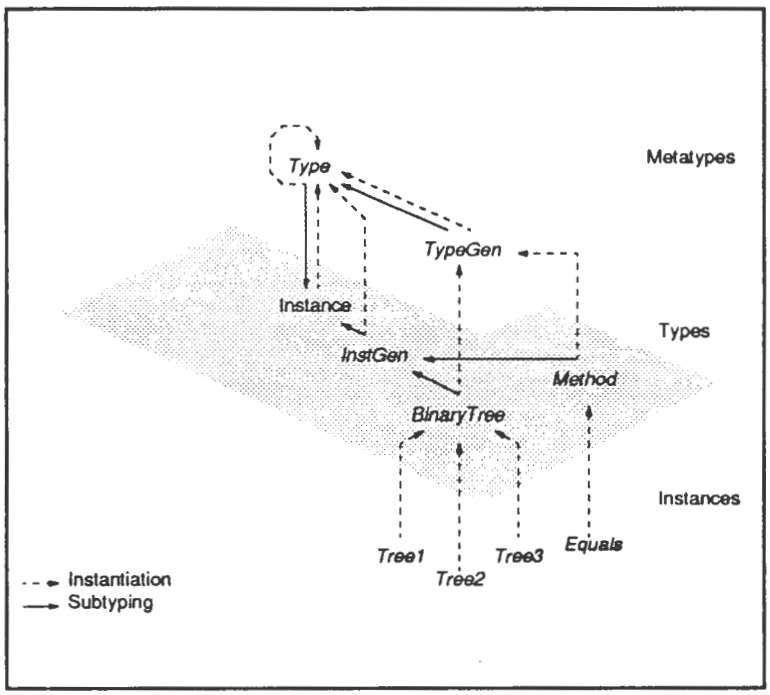

Figure 4.3: Hierarchy of Generic Types

4.4.6 Method

Each method in the database requires a method type. In the method type is a specification of the method structure which types the method value in the instance. The general structure of a method instance remains the same; the only parts that vary are the structures used in the components of the method structure. A generic method type called *Method* is defined and the owner, parameter and result components are abstracted to corresponding generic structure parameters. *Method* is a subtype of *InstGen* and is an instance of *TypeGen* (see Figure 4.3).

```
(TypeGen,
    (name:- 'Method',
    state:- Product(=>TypeGen,
        (name: String,
        source: String,
        code: Method(Owner, Param, Result))),
    behaviour:- {link, compile, edit, load, write},
    objects:- {Equals, New, NewType, ...},
    subs:- {},
    supers:- {InstGen},
    generic:- [Owner: Ref, Param: StructLim, Result: StructLim]))
```

Each method is an instance of type *Method* and the typing of the method value is stored in the instance by means of the GPLV, which is part of the reference to the instantiating type. The *New* method defined for type *Type* used earlier is an instance of *Method*. The advantage of this approach is that all methods are instances of one type. There is no need for the creation of a new method type for each new method structure that is required.

4.5 Summary

The model defines a structure system which specifies all the different data structures and values which may be used in the system. The object system defines object identity and facilitates persistence. The type system defines the constraints required for a set of values to be classed as an object-oriented database, the exemplar being such a set of values.

The system has higher-order features, since structures, methods and types may be treated as values. This results in the uniform way in which all operations and structures are defined in the system. The system is extensible by allowing the user to define new types, at both the ordinary and meta level. The set of base types can be extended by writing implementation for new base types and linking the operations to the system. The set of data constructors defined by the system may also be extended through the use of generic types.

The reference list example demonstrates how a real world situation can be intuitively modelled in the system. There is a direct relationship between the entities in the real world and the objects in the system.

Chapter 5 Implementation

Clearly it was impossible to consider implementing the entire system. So we chose to implement those features of the model which we felt were the most important and which were in keeping with our goals. The data structures and techniques used are not the most efficient, but then that is not their purpose. The implementation is used to demonstrate the model and identify any possible problems which are inherent in its design.

The structure system forms the basis for the representation of values in the system and hence modelling the real world. We implemented some of the structures in the structure system (as described in Section 5.1) along with various operations to manipulate the structures and their instances, and to transfer them to and from disk. The object system provides the facility to identify values in the system uniquely and to map them to entities in the real world. The object system makes use of an object table and an interface. With these two systems implemented, we can create the persistent objects required by the type system.

The operations defined in the model and the behaviour of objects require some form of method language to specify actions. We designed a simple syntax for expressing the operations which can be applied to objects and values. A compiler has been implemented primarily to check the correctness of the operations, and to ensure the encapsulation of objects, while allowing inclusion polymorphisms. The compiler also generates instructions for an abstract machine which manipulates the database. The abstract machine has been defined, but only partially implemented. The system as described in this chapter has been implemented in approximately 40 000 lines of *C* code.

5.1 Structure System

For each constructor in the structure system, we define a memory and a disk representation for its structures and corresponding instances. The structure system is data type complete, which means that any structure can be used as a component in any other structure. For this reason, the representations and operations for the structures and instances are defined in a recursive manner.

5.1.1 Structures

Each structure has a label which is used to identify its constructor, such as a record or product. The label dictates the representation used for the structure: in the case of a record it is a tree, and in the case of a product it is a list. The components of a structure are defined by pointers to other structures, which in turn are identified by their labels. For example, the following structure is used to define instances for type *Paper*:

Product(=>Type, (author: =>Person, title: String, appearsIn: =>Refereed Publication))

The memory representation for this structure is depicted in Figure 5.1. The product structure is represented by a product structure node, which has a pointer to a list of component structures. The ordering of the nodes in the list determines the ordering of the components in the instance. The first component is represented by an object reference structure node, which contains the identity of type *Type*. The second component has a pointer to a record structure. The record structure is represented by a binary tree sorted on attribute name. Each attribute is represented by a node in the tree, and stores a pointer to the structure associated with it, the position of the attribute value in the instance and the left and right subtrees.

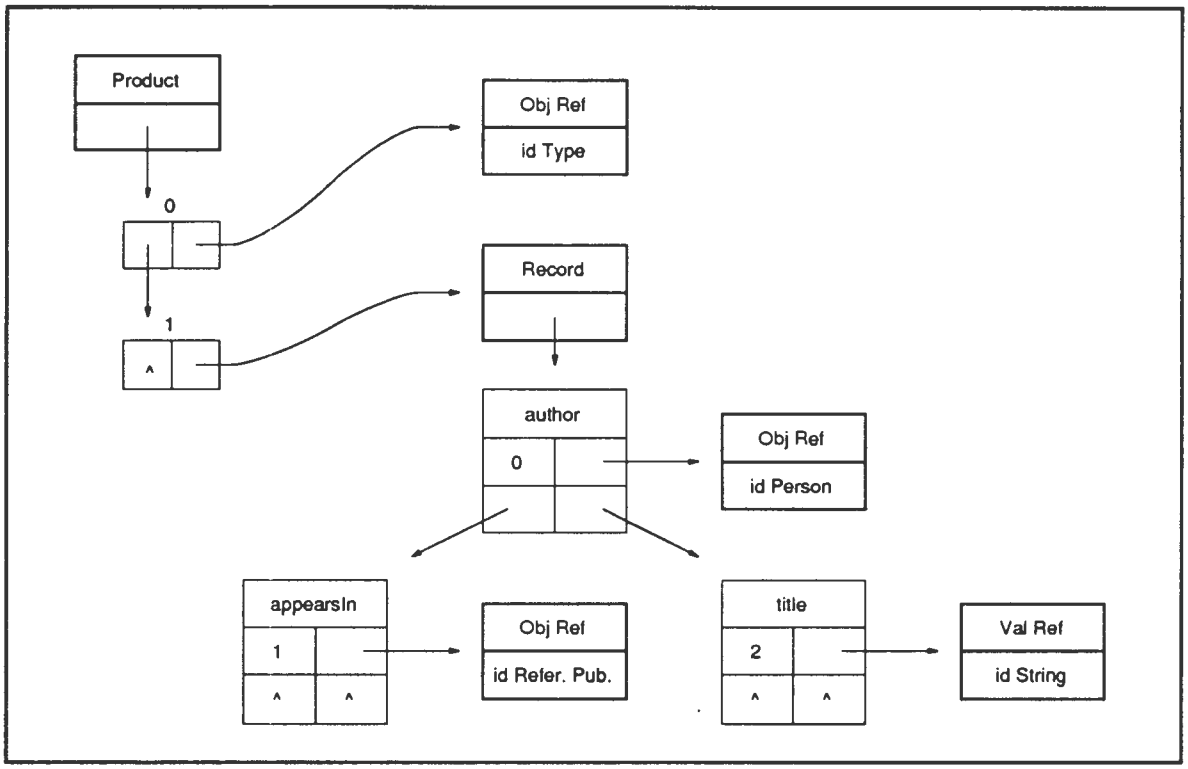

Figure 5.1: Memory Representation for a Structure

There is also a disk representation of the structures in the Structure System. Unlike the memory representation, the structure is stored in a sequential format, which corresponds to a depth-first traversal of the memory representation. Each structure is identified by a label which dictates the structure of its values and components. If the structure has a dynamic number of components, as is the case for products and records, the label is followed by a number indicating the number of components. The disk representation for the structure in Figure 5.1 is illustrated in Figure 5.2. The sequence begins with the product label, followed by the number two, indicating there are two product components. The first component is an object reference to *Type* and is stored as an object reference label followed by the identity of *Type*. The second component directly follows the first component and is identified by the record label and the number three, indicating that for this structure there are three components and

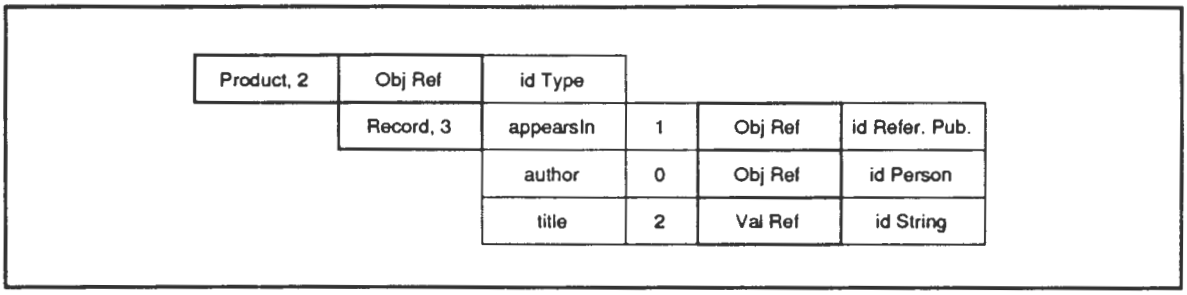

Figure 5.2: Disk Representation of a Structure

attributes. The attributes are stored in order and each one contains the attribute name, position and the attribute structure.

5.1.2 Instances

A structure defines the representation for an instance. When an instance is manipulated, the structure is required in order to determine the representation of the instance. The structure used above may have the following value:

(Paper, (author:- Person1, title:- 'The OODB Paradigm', appearsIn:- Proc1))

Each product and record instance is represented in memory by an array of pointers. In a product, the position in the array is determined by the position of the component structure in the list of product components. For the record value, the position is stored with each attribute in the structure. The memory representation for the instance above is illustrated in Figure 5.3. The product value is an array of two pointers, where the first points to the identity used for the object reference, and the second points to the record value. The value stored for the 'title' attribute is defined by a value reference to type String. Thus the representation for this value is defined by the value portion of the structure stored in type

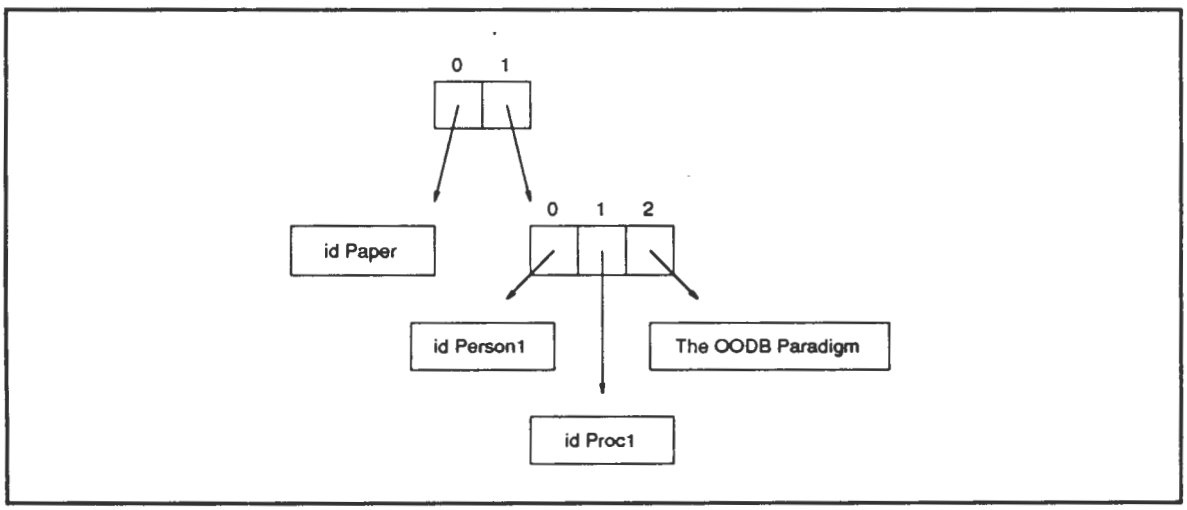

Figure 5.3: Memory Representation of an Instance

String, which is a dynamic base value. The pointer in position two points to a dynamic base value, which stores the string 'The OODB Paradigm'.

The disk representation for an instance is similar to that for a structure. It is stored sequentially, based on a depth-first traversal of the instance representation. However, unlike the memory representation, labels are stored in the sequence in order to define the structure of the representation. This allows the memory representation to be generated independently of the structure when it is read from disk. This independence is required because, when loading the database, the structures which define the instances are stored within some of the instances and thus the structure is not always generated before the instance. The disk representation for the instance above is illustrated in Figure 5.4 and starts with the product label specifying two components. The first component is labelled as an identity and stores the identity of type Paper. The second component is identified by a record label specifying three components. The third component for the record value is specified by a label for a dynamic base value, which is followed by the value. This allows the value defined by a value reference to type String to be read without making use of type String.

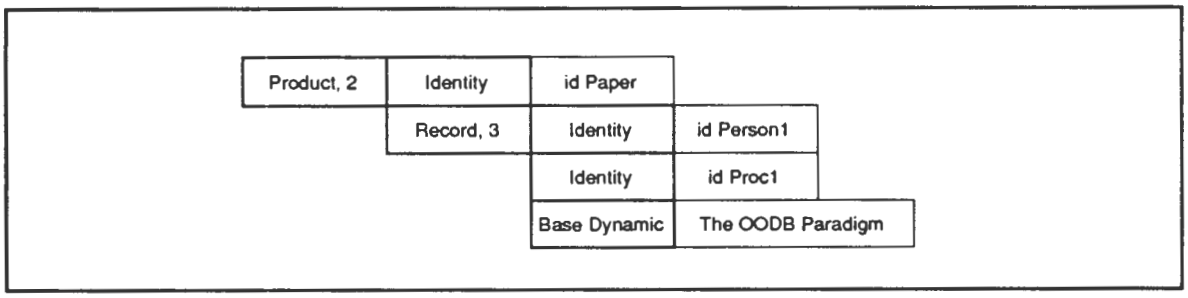

Figure 5.4: Disk Representation of an Instance

These forms of representations are used for all of the structures and instances in the system. They are used to create the objects, types and instances which are used in the rest of the model.

5.2 Object System

The object system implements the identity and persistence of objects. It consists of an object table structure for storing the objects and an interface for manipulating them. The object table consists of a hash table onto which balanced binary trees are chained. The *modulo* function is used as the hash function to split the identities into disjoint sets. Each set of identities is represented by a balanced binary tree and is accessed from the hash table. Each node in the tree contains the identity of an object and a pointer to the value for the object. Thus, the object in the model is implemented by this identity and the value to which is pointed. Figure 5.5 depicts the object table structure. By having both of these structures, the number of comparisons required to find an object are greatly reduced.

The interface consists of a number of functions which can be used to perform the following operations. A new object is created by generating a unique identity and placing a new node in the tree

with a pointer to the object. The existence of an object can be determined by finding its identity in the object table. An object is accessed by specifying its identity. If the identity appears in the object table, then the pointer which is associated with it is returned. Objects may also be removed from the table.

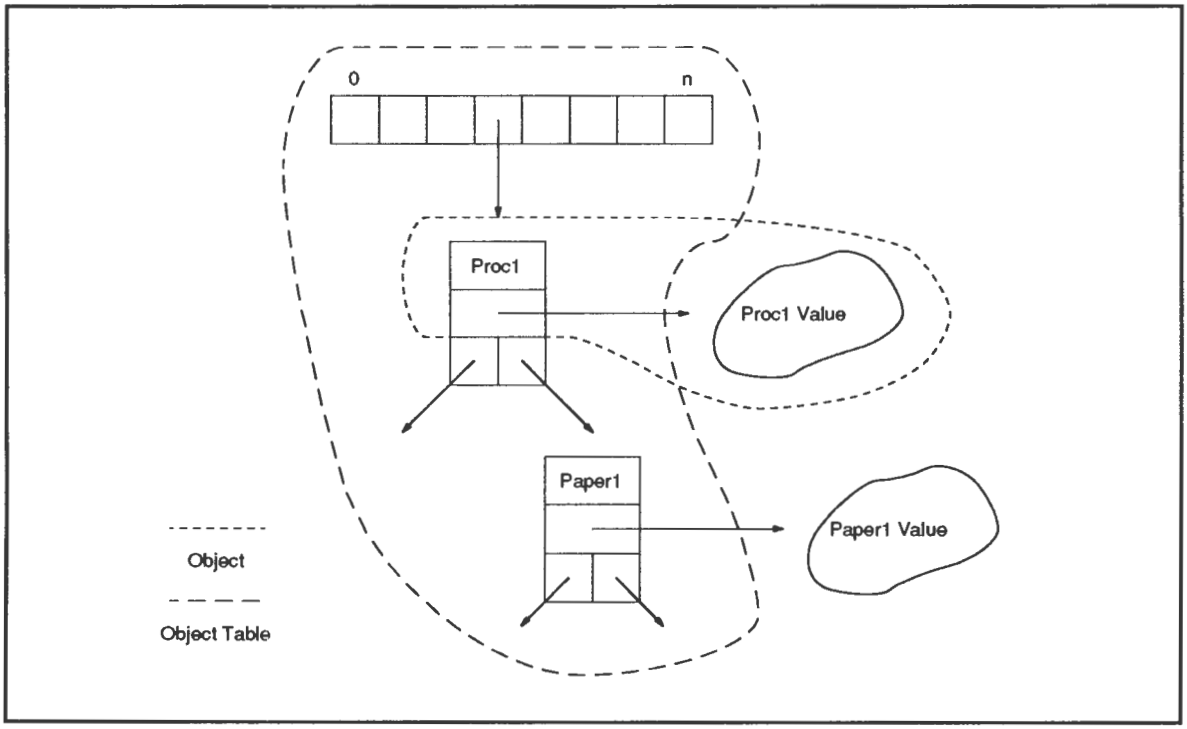

Figure 5.5: Object Table

When a session begins, a particular database is selected and is loaded into memory. This process involves reading the identities of the objects and their values, and generating the object values as discussed in Section 5.1.2. As each object is generated, so the associated node in the binary tree is also generated. After the database has been read, all objects are represented in memory. At the end of the session all objects are transferred back to disk with their identities.

Temporary objects are specified for a specific level. Figure 5.5 depicts the object table for level 0 objects. For each different level a new object table is created. When a persistent object is created, it is inserted into the level 0 table. When a temporary object is created, it is inserted into the table that corresponds with its level number. The operation of clearing a level simply involves deleting the object tables for that level and the ones above it. Due to the reference constraints on temporary objects, no dangling references will occur.

5.3 Method Language

We have defined a model for HOOD and this section poses some of the requirements for the system's method language. The main requirements are to support complex values, the use of structures as values, and the syntax and typing of messages, apply-to-all and insertion operations. We present a portion of

the syntax, which allows ordinary operations, messages, apply-to-all and insertion operations to share the same grammar production for all operations.

5.3.1 Requirements of the Method Language

In the model for HOOD we specify operations, values and structures which must be expressible in the method language. This section deals with these requirements.

5.3.1.1 Typing of Methods

Each method is typed by three structures as indicated below.

```
Method(Owner, Parameter, Result)
```

First is the owner structure, which specifies the type of the instance to which the message is passed. *This* is a predefined variable in all methods and is typed by the owner structure. It represents the instance to which the message is passed. Each method is defined for a specific type and belongs to that type. Hence we say that the method is *owned* by the type. The owner structure is a reference to the owner type.

Second is the parameter structure which specifies the type of value which is used as the parameter in the message pass. If the parameter structure is defined, then it can be accessed via the predefined variable *Param*. The third structure is the result structure, which types the instance returned by the method. These three structures form the method structure, or signature, which types a method.

As an example, a method can be written for the *Integer* type to determine the maximum of two integers. The method structure has *Integer* as the owner and parameter, because these are the two values which are being compared. The result is also an *Integer* since the larger of the two is returned.

```
Method(Integer, Integer, Integer): max;
```

Given two *Integer* variables *X* and *Y*, we can pass the message *max* to *X* with *Y* as the parameter.

```
X.max(Y)
```

The result will be either *X* or *Y* depending upon which is larger. In our model we also support generic types, which make use of generic variables (parameters). If the owner is a generic type, then the method may make use of the generic variables which are specified and typed by the generic type.

5.3.1.2 Database

A database consists of types and their object instances. Associated with each type are the following:

- a unique name, used to reference it symbolically;
- a state structure which defines its instances;
- a set of methods which forms its behaviour;

- its supertypes and subtypes;
- the objects which are its instances.

Each method is stored in the database as an object and is associated with the behaviour of its owner. In a method, an instance may be passed a message which is defined in its type or in one of its supertypes.

A type may be used in a method as an instance, simply by specifying the name of the type. The name of a type can also be used to define a reference structure, viz. a value or object reference.

The database consists of all the types which are defined in the system. The base types, such as *Integer*, *String*, *Real* and *Boolean*, are defined in exactly the same way as user-defined types, and thus are stored in the same manner in the database. These specific base types are required by the method language, since they are used by the primitive operations and are also required for the control structures.

The compilation of a method occurs in the context of the database. The types and methods which are stored in the database may be used by the method and are accessed by the compilation process.

5.3.1.3 Typing

Expressions in the method language are strongly typed statically, that is, their typing structures are determined at compile time. There are however some primitive operations whose results can only be typed at run-time. For these operations dynamic type checks are required (see Appendix C). The compiler at present statically binds all type names and method names to their respective objects. This characteristic can however be relaxed to dynamic binding.

An instance of a type may only be manipulated by messages which are owned by the type. A method can manipulate the internal structure of an instance which is defined by the owner. Since the method is compiled in the context of the owner (as part of its behaviour), it has access to the state structure which defines the typing of the instance. Typically the access to the internal structure is performed via the predefined variable *This*.

The model defines subtyping for instances which are defined by types and data structures. We may use an instance in any context where an instance of a superstructure or supertype is expected. In other words, the method language supports substitutability. This, in turn, implies that the compiler must be able to deal with the subtyping of structures and types.

5.3.1.4 Denotable Values

The model specifies various complex data structures and corresponding values. These include the following structures: base, set, list, product, record, reference, etc. The method language is required to facilitate the denotation of these complex values, in a manner similar to that which we use in the model.

A base structure is used to define base types such as Integer, String, etc. A base structure has the form:

```
Base[4] or Base[-]
```

which defines a static base value of four bytes and a dynamic base value, respectively. Associated with these structures are base values which are denoted either as character strings or as byte strings. The set of base types is totally extendible. They make use of the base structure for their definition, primarily for the specification of size requirements, either fixed or dynamic. The system provides persistence and integrity for a static or dynamic string of bytes. The base type is implemented externally to the system by defining a number of *C* functions. These functions are then linked to method objects in the system, and the methods are linked to the base type which is created internally. These methods form the interface to the base type. When one of these methods is specified in a message, the method is located and it is determined that the method is defined externally. The external function is located and passed the corresponding arguments. Its result is then used as the result of the message pass. The structure of the base value is defined in the implementation and the linked operations manipulate it accordingly. The system has no knowledge of its structure except for its size requirements.

The base types *Boolean*, *Integer* and *String* are predefined because they are required by some of the primitive operations in the method language. Thus, the external implementation of these types must match the internal structure which we have used. But for all other base types it is impossible to define specific base type values for the method language. This would require a knowledge of the external implementation in order to translate the lexical value into its correct representation. Each base type should supply various methods for the creation of base values, such as floats or bitmaps. Creation operations could also make use of integer values which they convert into their own values.

The model defines the concept of a constructor which combines structures to form a new structure. Associated with each constructor are a number of operations and a notation for specifying both the structure and the value. The record constructor produces a record structure:

(title: String, date: Date, ...)

which has a record value:

(title:- S, date:- D, ...),

where *S* and *D* are variables containing *String* and *Date* values. Similar constructors, structures, and values exist for sets, lists, products, etc.

One of the higher order features of the model is that structures may be used as values. There is a constructor called *Struct* which produces a structure whose values are structures. The implication of this is that the method language must treat structures and values in the same way. We have defined various operations which may be performed on structures, such as *Multiple-Inheritance* which determines from a list of structures, a structure which is their subtype. Thus the structures behave like values in the context of these operations. For example, each type stores the structure which defines its instances, so that, in the context of the type, this structure is treated as a value.

In the model we define the concept of a Nil value for each of the structures. There is also a Nil structure, which is the Nil value for Struct. Nil also denotes the null object, which is used with references

to objects. Nil is a parametric polymorphic constant and can be used anywhere. The typing or structure of the Nil value is determined from the context in which it is used.

The name of a type as described earlier can be used to denote either a type (object) or a value reference to that type (structure). The type's name is overloaded since it denotes two distinct concepts. By examining the context, we can determine how the type is being used. Another case of overloading is that of parentheses. The product value is denoted by a left parenthesis, a list of values separated by commas and a right parenthesis. An expression may also be placed in parentheses to denote precedence. A conflict occurs between a parenthesized expression and a product value with only one component. This conflict is resolved in Section 5.4.2.

5.3.1.5 Instances

Each type has a state structure which defines its instances. In the model we have split instances into objects and values. An object is an independent persistent entity which has an identity, and a reference to the type that instantiated it. From this reference, the structure of the object can be determined. A type's state structure contains the specification of the reference to the instantiating type.

A value on the other hand is dependent on the context in which it is used. A value may be stored in a variable or as a component in a larger value. The typing of the value is determined from the typing of the variable, or from the typing structure of the component in the larger value. Since the typing of a value may always be determined from its context, it is redundant to store the instantiating type in the value. The structure of a value is defined by the type's state structure excluding the reference to the instantiating type.

In the model we support two forms of reference structures which make use of these different forms of instances. The object reference structure:

```
=>Integer
```

defines that the identity of the object is to be stored, while the value reference structure:

```
Integer
```

defines that an actual value defined by a portion of the type's state structure is to be stored. The state structure for the *Integer* type is defined as follows:

```
Product(=>Type, Base[2])
```

An object of type *Integer* has an identity and consists of a reference to the instantiating type (=>Type), and a base value of two bytes (Base[2]), for example:

```
(Integer, 1)
```

A value reference to *Integer* would contain only the base value:

1

Since an object is defined by the entire state structure and a value is only defined by a portion of it, it is possible to obtain a value from an object. The implication of this is that wherever a value is expected in a method, an object may also be used. The value must be implicitly extracted from the object (a form of coercion). For message passing, we have specified that all values are passed by value and that all objects are passed by identity.

5.3.1.6 Operations

The method language requires operations to perform the actions of the system. These operations are either specified by an operator symbol or by the name of an operation.

All named operations which are defined for structure values as well as messages defined for instances make use of the message passing syntax, i.e. the instance or value, a dot, the name of the operation or method and an optional parenthesized parameter. The dot operator is also used for subscripting a record value. In the case of the variable *This*, type resolution must determine if a message is being passed or if the record value is being subscripted.

We have overloaded a number of other operators and named operations. Once again, type resolution must determine from the context which specific operation is being used. For example, the operator '=< 'has been overloaded for the subset, sublist, substructure and subtype operations. The user defined operations are specified by methods and identified by their names. Names used for methods need not be unique in different types, since the method is obtained by a look-up which allows for overloading.

For list and set structures, we have defined apply-to-all and insertion operations, denoted by '<=' and '/' respectively. The apply-to-all operation applies an operation to every value in a set or list, yielding a set or list containing the results. For example, given a list of journals one can apply the *Date* message to every journal in the list.

```
[J1, J2, J3, J4, ...] <= .Date
= [J1.Date, J2.Date, J3.Date, J4.Date, ...]
= [1/1/91, 2/3/91, ...]
```

The result is a list containing the publication dates of the journals in the list.

The insertion operation applies an operation to successive pairs of values in a set or list. For example, given a list of journals, the newest journal can be obtained by inserting the *Newer* message, which is defined in the *Journal* type, into the list.

```
[J1, J2, J3, J4, ...] / .Newer()
= (...((J1.Newer(J2)).Newer(J3)).Newer(J4) ...)
= (...(J2.Newer(J3)).Newer(J4) ...)
= J2
```

We have outlined the main requirements for a method language as specified by the model. In the following two sections we cover the grammar of the method language and its type resolution.

5.3.2 Syntax

```
exp  : ID
     | ID '[' NUM ']'
     | ID '[' '-' ']'
     | ID '(' explist ')'
     | '(' record ')'
     | '=>' ID
     | '=>' ID gplv
     | VALUE
     | ID gplv
     | '[' ']'
     | '[' explist ']'
     | '[' gpls '[' '
     | '(' explist ')'
     | '(' recordval ')'
     | '{' '}'
     | '{' explist '}'
     | exp opexp

explist  : exp
         | explist ',' exp

record  : ID ':' exp
        | record ',' ID ':' exp

recordval  : ID ':-' exp
           | recordval ',' ID ':-' exp

opexp  : '=' exp
       | '=-=' exp
       | '=+=' exp
       | '=*=' exp
       | '=/=' exp
       | '=<' exp
       | '>=' exp
       | '>' exp
       | '<' exp
       | '*' exp
       | '+' exp
       | '-' exp
       | ':' exp
       | '!' NUM
       | '.' ID
       | '.' ID '(' exp ')'
       | '<=' opexp
       | '/' op

op  : '='
    | '=+='
    | '=*='
    | '=/='
    | '*'
    | '+'
    | '-'
    | '.' ID '(' ')'
```

The syntax of the method language has been designed to facilitate message passing, the use of structures and values at the same level, the denotation of complex values and the apply-to-all and insertion operations. The method syntax has a declaration section for variables and a body consisting of a sequence of statements. The statements provide for control structures and expressions. The expressions are used to perform the actions of the system through constructs such as message passing.

Above is a portion of the syntax which relates to expressions and operations. The only reserved words are those used for the control structures *IF*, *THEN*, *ELSE*, etc. which appear in statements. Only these identifiers are converted to tokens. The token *ID* denotes all other identifiers. As explained below, the production in which an identifier is used has its own reserved word table associated with it. Once the reserved word has been identified, more specific requirements regarding the production can be checked.

If we convert an identifier into a token, then whenever that identifier occurs in a method, it is converted to the associated token. This would define a large reserved word table and words such as *Method*, *List*, etc. could only be used in the grammar rule that defines the structure. We would be unable to use these identifiers to denote the names of types. We have decided to allow overloading of identifiers; thus, identifiers such as *Method* can be used for the specification of a structure and for the name of a type. From the context in which the identifier is used, it can be determined which concept is being referred to. Once a grammar rule has been determined, only then is the identifier which is used in it compared to the identifier which is expected.

5.3.2.1 Expressions

Expressions are used to specify values, structures and operations. The first nine productions for expressions above are used to parse structures. For example, the production:

exp : ID '(' explist ')'

is used to produce product, method, list and set structures. Once the parser has established that this production is to be used, the identifier is checked against a set of reserved words including: *Product*, *Method*, etc. Once the identifier has been identified, the number of expressions required in the expression list is determined and checked.

The last expression production is used to parse operations, the other productions being used to parse complex values. Because record structure and record value are defined as productions of the same non-terminal *exp*, we require some means of distinguishing between a value and a structure. The record value makes use of a colon followed by a dash ':-', while the record structure makes use of just a colon ':'.

5.3.2.2 Message Passing Syntax

The syntax for a message pass is defined by the following three productions:

```
exp   : exp opexp
opexp : '.' ID
      | '.' ID '(' exp ')'
```

The first production specifies the recipient of the message pass, *exp*, and the operation. The second production specifies a message pass operation with no parameter. This production is also used to specify

record subscripting. Type resolution is used determine which operation is being used. The third production specifies a message pass with a parameter.

These productions are used both for primitive messages (defined by the system) and user-defined messages (messages owned by a type), the correctness of their use being determined by type resolution.

5.3.2.3 Apply-to-all

The syntax for operations is defined by the productions of the non-terminal *opexp*, which stands for operation expression. We have not defined the productions for operations in the standard way:

exp : exp op exp

because of the apply-to-all operation.

The apply-to-all operation has the following components: an expression to which an operation is applied, the apply-to-all operator, an operation which is to be applied and an optional expression (the existence of which depends on the operation being applied). The production above does not satisfy the requirements of the apply-to-all operation:

exp : exp <= op exp

We also wanted to use the same productions for both *ordinary* operations and apply-to-all operations. As a result, we have defined all operations by the production:

exp : exp opexp

which specifies an expression followed by an operation-expression. The operation-expression productions specify an operator followed by an expression, as seen, for instance, in the message passing example above. This syntax allows for "ordinary" operations, although is does require more complicated type checking since the left expression, operation and right expression do not all appear in the same production.

The apply-to-all operation is specified as an operation expression, defined by the '<=' operator and an operation expression:

opexp : '<=' opexp

This production allows the apply-to-all operation to make use of all operations defined in the syntax, including itself, The example presented earlier is parsed as follows:

[...] <= . Date

exp '<=' '.' ID
exp '<=' opexp
exp opexp
exp

Nested list and set values can make use of multiple occurrences of the apply-to-all operation due to its recursive definition. Given a list containing lists of journals, the date of each journal may be obtained while preserving the nested list structure. The following expression applies to the outer list an operation which will yield a list of dates from a list of journals. This operation is specified by applying the date message to a list of journals.

```
[[J1, J2], [J3, J4], ...] <= <= .Date
[[J1, J2] <= .Date, [J3, J4] <= .Date, ...]
[[J1.Date, J2.Date], [J3.Date, J4.Date], ...]
[[1/1/91, 2/3/91], ...]
```

This expression is parsed as follows:

```
[[J1, J2], [J3, J4], ...] <= <= .Date

exp '<=' '<=' '.' ID
exp '<=' '<=' opexp
exp '<=' opexp
exp opexp
exp
```

5.3.2.4 Insertion

The insertion operation has the following components: an expression, the insertion operator '/' and the operation which is to be inserted. The insertion operation has also been defined by the operation-expression production, since it is an operation and may be used wherever any other operation is used. It consists of the insertion operator and an operation:

```
opexp : '/' op
```

The operation production is required for the various operations which may be inserted. The newest journal example provided earlier is parsed as follows:

```
[J1, J2, J3, J4, ...] / .Newer()

exp '/' '.' ID '(' ')'
exp '/' op
exp opexp
exp
```

The next section deals with resolving and typing checking methods which are specified according to the above syntax.

5.4 Type Resolution

We have specified typing structures which are used by the type resolution process to type check the operations and expressions. The resolution process is also responsible for the inference of typing structures where these structures are undetermined.

The compiler for HOOD has been written with the aid of Lex and Yacc. Approximately 12 000 lines of *C* code have been used to perform the process of compilation and type resolution.

The purpose of type resolution is twofold: first is the inference of typing structures, and the second is the checking of expressions for type correctness. Due to the relative simplicity of the method language syntax, the resolution of typing is far more complex. The typing process makes use of typing structures to store the typing information generated by the expressions as they are parsed. These structures are then used by the type checking and resolution processes to determine the correct typing of values. For example the following operation must be resolved to determine the structure of the components in the list. Once this structure has been determined, the appropriate *Date* message can be located if it exists. From this, the structure of the resulting list can be determined

```
[J1, J2, J3, J4, ...] <= .Date
```

5.4.1 Typing Structures

The typing of expressions and operations is deferred until sufficient information is available, due to the syntax design for the productions *exp*, *opexp*, and *op*. These structures are used to store the typing information which is obtained as the expressions and operations are parsed. These structures are called typing structures and their representation is closely related to the productions in which they are generated. The remainder of this section describes these structures.

Each structure has a tag to identify it, while some have additional information stored within them. Listed below are the typing structures used for the expression productions, which we call expression structures:

(Error) — the expression has a typing error.

(Nil) — a nil identifier has been used and must be resolved.

(This) — the predefined variable *This* is being used. It must be determined if it is being used as an instance or if it is being used as the value defined by the type's state structure.

(Structure, structure value) — a structure of some kind has been denoted (first 9 productions for *exp*).

(Type, identifier) — the name of a type has been used and it must be determined if it is being used to reference the type or if it is being used as a value reference structure.

(Base Value, value) — a base value has been denoted. The base value is stored in the structure for possible conversion later.

(List Value, (...)), (Set Value, (...)), (Product Value, (...)) — these complex values share the same structure, except for the tag which distinguishes them. The structure consists of a list of typing structures, which were generated by the component expressions in the complex value (the production *explist*).

(Record Value, (...)) — this structure is similar to the one above, except in addition to each component's typing structure, the name of the attribute is also stored.

(Determined, structure) — In the case where the typing structure for an expression has been determined, this structure is used. It contains a structure which types the value denoted in the language.

The operation-expression productions have the following typing structure associated with them, which is called an *opexp structure*. A tag, which is defined by the operation, is also used to distinguish between the different operations.

(Operation, (...)) — For productions of the form:

opexp : operator exp

the operator is stored as the tag along with the expression structure.

(Message, identifier), (Message param, identifier, (...)) — For the two message passing productions, the identifier and optional parameter expression structure are stored.

(Apply-to-all, (...)) — The apply-to-all production has a structure tagged with the apply-to-all operator and contains an *opexp* structure.

(Insertion, (...)) — The insertion production has a structure tagged with the insertion operator and contains an *op* structure (which is defined below).

The operation productions also require a typing structure, which is used to store the operation and is called an *op structure*. It has a tag to identify the operator, and in the case of a message pass operation, the identifier used in the operation is stored. The insertion of the *Newer* operation into a list of journals is represented as follows:

exp: (Determined, List(=>Journal))
opexp: (Insertion, (Message param, Newer)

5.4.2 Resolution Process

The typing of an expression may be in one of three states, namely, there is a typing error, it is determined, or it is undetermined and must be resolved. The type resolution process involves inferring the correct typing for an undetermined expression, type checking the correctness of components in an expression and the generation of the result's typing structure.

As a method is parsed, so the typing structures are generated. The resolution process is performed either by attempting to resolve these structures on their own against a determined structure, or against another typing structure. For each operation, the associated typing structures are generated and resolved against the specific requirements for that operation.

5.4.2.1 Typing Structures

The type checking process determines if a determined typing structure is correctly typed for the context in which it is used. This is done by comparing the determined structure to an expected structure. The resolution process attempts to infer the correct typing structure for an expression containing an undetermined typing structure. These processes may only occur if the component typings are either undetermined or determined. If a component's typing is erroneous, then the entire expression is erroneous and the Error typing structure is generated as the result. The resolution process determines the correct typing for Nil, *This*, Types, the use of parentheses and the typing structures for complex values.

Each of these processes requires two structures. These structures are modified by each process which returns a result indicating the status of the structures. The structures may be in five different states:

Substructure — the first structure is a substructure of the second structure and both structures have been determined.

Equivalent — the two structures are equivalent and both structures have been determined.

Superstructure — the first structure is a superstructure of the second structure and both structures have been determined.

Unresolved — the typing structures contain components which cannot be resolved and, apart from the unresolved components, there are no errors.

Different — the two typing structures have been determined and are different or erroneous.

When resolving the components of a typing structure, the type checking and resolution processes are used. The result of these processes is a state as defined above. To determine the status of the two structures, the state of each pair of components of the structures is used. A transition matrix, shown in Table 5.1 is defined to determine the status of the structures. The left hand side indicates the current

	sub	*equiv*	*super*	*unres*	*diff*
sub	*sub*	*sub*	*diff*	*unres*	*diff*
equiv	*sub*	*equiv*	*super*	*unres*	*diff*
super	*diff*	*super*	*super*	*unres*	*diff*
unres	*unres*	*unres*	*unres*	*unres*	*diff*

Table 5.1: Transition Matrix

state of the pair of structures being compared and the top indicates the status of the current pair of components.

As an example, consider comparing the following two product structures:

$$P_1\text{: } Product(Integer, Journal, Publication, Publication)$$
$$?$$
$$P_2\text{: } Product(Integer, Publication, Publication, Journal)$$

$$Integer \equiv Integer,\ equiv\text{–}equiv \rightarrow P_1\ equiv\ P_2$$
$$Journal \leq Publication,\ equiv\text{–}sub \rightarrow P_1\ sub\ P_2$$
$$Publication \equiv Publication,\ sub\text{–}equiv \rightarrow P_1\ sub\ P_2$$
$$Publication \geq Journal,\ sub\text{–}super \rightarrow P_1\ diff\ P_2$$

One begins by assuming that the two structures are equivalent. The first components (*Integer*) are equivalent, thus the structures remain equivalent. The second component of P_1 (*Journal*) is a substructure of the second component of P_2 (*Publication*), thus P_1 is a substructure of P_2. The third components are equivalent, thus P_1 remains a substructure of P_2. The fourth component of P_1 is a superstructure of that of P_2, thus the status of the two structures is different. The resolution process terminates at this point, even if there are more components to analyze; hence, there is no row labelled *diff* in Table 5.1.

5.4.2.2 Individual Structures

An undetermined typing structure may be resolved on it own by performing the following default conversions to a determined typing structure:

This — is converted into an instance with the structure of the method's owner.

Structure — defines a structure value and has determined structure *Struct*.

Base Value — is converted to a determined structure defined by a dynamic base structure.

List, Set, Record, Product Values — are only converted into determined structures if all of their component values can be converted into determined structures.

If the typing structure cannot be determined, then it remains unchanged, while if there is an error, it is converted to the *Error* structure. This process can be used to determine the structure of a parameter in a message pass.

5.4.2.3 Determined and Undetermined Structures

When resolving an undetermined typing structure against a determined typing structure, there are specific determined structures against which the undetermined structure may be resolved.

Nil — resolves against any structure since it is defined for all structures.

This — is resolved by comparing the structure of the method's owner structure to the determined structure.

Structure — may only be resolved against a structure which is defined by *Struct*.

Type — denotes a value which is either a reference to the type (object) or it is a structure, called a value reference structure. If the determined structure is defined by an object or value reference, then the type is being used as an object. If the determined structure is defined by *Struct*, then the type is being used as a structure.

Base Value — the structure may only be resolved against a base structure.

Set Value, List Values — the determined structure must be either a set or list structure respectively. Each component in the set or list value is then resolved against the component in the determined structure by recursively using this process.

Product Value — if the determined structure is a product, then the components are compared using the resolution process with the aid of the transition matrix (Table 5.1). If the above test fails and the product value contains only one value, then that component is compared with the structure. In this case the parentheses which generate a product value are being used to parenthesize the expression.

Record Value — requires a record structure and compares attributes and their typings with the aid of the transition matrix.

For example, assume that a variable *X* is declared and contains an object reference to *Type*. The value held in *X* is compared to type *Person*, which is parenthesized:

```
=>Type: X;
X = (Person);
```

The associated typing structures are:

(Determined, =>Type) — (Product Value, (Type, Person))

The *Product Value* structure is generated by the 13th *exp* production. Because the equality operation requires two values which have the same structure, and since the one structure is determined, the above process is used. We first compare the undetermined product value structure to the determined structure. Since the determined structure is not a product structure, this test fails. Since the product value has only one component, we now compare the following two typing structures:

(Determined, =>Type) — (Type, Person)

The undetermined structure is now tagged by *Type*. Since the determined structure is defined by an object reference structure to *Type*, the type value is being used to reference the type *Person* and not to denote a value reference structure. Thus, the type value *Person* is also a value of the object reference to *Type*, so the values have the same structure and the operation is valid.

5.4.2.4 Undetermined Structures

Two undetermined typing structures are resolved by comparing their tags. The process modifies the two typing structures and returns their status. Listed below are the different cases for each pair of tags in the expression structures along with the associated actions:

Nil - Nil — both remain unresolved.

Nil - This — both are converted to the structure of the method's owner.

Nil - Structure — both are converted to *Struct* values.

Nil - Base, List, Set, Record, Product — provided that these values can be resolved on their own, Nil is converted to their structure. Otherwise they remain unresolved.

Nil - Type — both remain unresolved since they can be used as structures or objects.

This - This — both are used as values.

This - Type — both are used as object instances.

This - X — the remaining structures are check against the structure of the method's owner.

Structure - X — the structure is converted to the determined structure *Struct* and the undetermined typing structure is compared to it.

Base Value - Base Value — when compared with another base value, both are converted to determined base structures. The other cases are defined above.

Set Value, List Values - Set Value, List Values — for set or list values, both values are resolved on their own by resolving their components. If the resulting structures are determined, then we can compare the two structures.

Record Value - Record Value — matching attributes are found and then the component values are compared. Using the transition matrix for the attributes, the status of the two values is calculated.

Product Value - X — the product value is compared against another structure by first checking if the other value is a product value and comparing the components, with the aid of the transition matrix. If this fails and the product value only has one component, then this component is resolved against the other typing structure, since the value denotes a parenthesized expression.

Type - Type — both structures remain unresolved.

Type - Structure — both structures are converted to determined *Struct* structures.

For example, if we have an object reference to type *Paper* and we wish to know if it is a structural subtype of a value reference to type *Article*, then we would specify the following expression:

```
=>Paper = Article
```

(Structure, =>Paper) — (Type, Article)

This is the last case in the resolution process above. The structure value is converted into a determined value with structure Struct. The type value denotes a value reference structure which is also a determined value with structure *Struct*, that is,

(Determined, Struct) — (Determined, Struct)

The operation is valid since it requires two values which are structures. The result of the expression is *false* since an object reference structure is not related to a value reference structure.

If the above processes have been unsuccessful in converting an undetermined typing structure to a determined structure, then all occurrences of *Type* in the structures are converted to objects as the default, in order to resolve the typing structures. The processes above are used again to determine the typings. If they are still unsuccessful, then there is insufficient typing information in the expression and it cannot be resolved.

5.4.3 Resolution of Operations

The resolution of typing occurs at the production:

exp : exp opexp

since at this point all typing information for an operation has been gathered. Resolution also occurs in the statements where expressions are used for assignment operations, boolean conditions and return values. Each operation has specific requirements which its arguments must satisfy. In this section we deal with these requirements and their validation. The resolution process described above and the resulting status of the typing structures is used for the resolution of operations.

5.4.3.1 Primitive Operations

The method language provides complex data structures and associated operations. These operations are called primitive operations and make use of operators and messages. A number of these operations are overloaded and it is the responsibility of the type resolution process to determine which operation is required. For example, in Section 4.1.1.2, the primitive operations for record values are covered, which include assembly, attribute subscripting, casting and relational operations.

The resolution process first determines the operation and then the structures of its arguments. The operation is determined by examining the tag in the *opexp* structure. For each primitive operation, the typing requirements have been determined from the model, viz. the left and right arguments or the recipient and parameter. From these requirements we determine which resolution process to use. The equality operations, for example, require the left and right hand sides to have the same typing. In this case, the typing of the left argument is resolved against the typing of the right argument. If the resulting status of the typing structures is either *sub*, *equiv* or *super*, then the equality operation is correctly typed and is thus valid.

The operator '>=' is used for the superset and superlist operations. In both cases, the left and right arguments are required to have the same typing, but restricted to a set or list value, respectively. The typing of the left argument is resolved against the typing of the right argument and then the

determined typing structure is checked for a set or list value. If the status of the typing structures is either *sub*, *equiv* or *super*, then the operation is correctly typed and is thus valid.

In each of the examples above, if an operation is valid, then its result is a determined typing structure for a boolean value. This result is passed up to the next production which makes use of it as a subexpression.

The *Multiple-Inheritance* primitive operation is defined for structures. It requires a list of structures and attempts to produce a new structure which is a substructure of all the structures in the list. The message is passed to a list of structures and the result is a structure. The resolution process resolves the typing of the recipient against a determined structure for a list of structures. If the typing is valid, then the result of the operation is a determined typing for a structure.

The next three sections deal with specific operations, viz. message passing, apply-to-all and insertion.

5.4.3.2 Methods

Primitive operations and messages share the same productions. By examining the typing structures of the arguments, we can determine which form of operation is being used. If the recipient's typing structure is a record structure and the operation does not have a parameter, then the operation is record subscripting. If the typing structure is defined by a reference to a type, the operation is a message pass. Otherwise the operation is a primitive message pass as defined in the previous section. This section deals with type resolution of user-defined messages.

All user-defined methods are treated as objects in the database. As objects, they require a type for their definition which is called a method type (see Section 4.3.6). Each method object contains the name of the method and the method (value) as defined by a method structure. By examining the method type, we can determine the typing of the method structure, since the method in the object is a value defined by this structure. As explained earlier, the method structure contains an owner, which identifies the type that owns the method. Correspondingly, in the type's behaviour each of these method objects is identified.

The type resolution process determines that a message pass operation is specified. A look-up procedure is now utilized to find the correct message in the database. This procedure is also known as binding and dispatching. The look-up algorithm is detailed below:

1. The typing of the instance has been determined to be an instance of a type.
2. The database is consulted for the type and the behaviour of the type is located. From the behaviour, each method object is examined to determine if it defines the message which is being passed. For each method object, the associated method type is obtained which defines the structure of the method object. The name is first checked against the identifier which is used in the message pass. If the names are different, then proceed to step 5; else this is a candidate method.

3. The structure of the method is obtained from the method type. The structure of the owner component is compared to the structure of the instance. If an object reference is specified, then only an object may be passed this message. If the instance is a value instance, then this message pass is invalid. On the other hand, the owner may only require a value reference to the type. In this case either a value or an object may be used. In the case of an object, it is converted to the corresponding value. If none of the conditions in this step are met, then proceed to step 5.
4. The parameter structure is type checked next. The structure of the parameter value must be a substructure of the parameter component which is specified in the method structure. If this condition is satisfied, then this is the correct method so proceed to step 6.
5. If no correct method is found, then the supertypes in the *supers* set are added to the search queue. Each of the supertypes is queued, provided it has not been queued previously, to effect a breadth-first search of the type hierarchy. If the queue is not empty, then use the type at the front of the queue and repeat from step 2 until a correct method is found. If after searching through all the types, which are supertypes of the instance, no correct method is found, then this message pass is invalid. This occurs when the queue is empty.
6. Once the correct method has been found, the message may be passed to the instance. The method result structure which forms the result of the operation is obtained from the method type.

Since a type may have multiple supertypes, the order in which these types are searched is of importance. The methods in the type hierarchy are searched breadth-first, which means that the most specialized methods are checked first and the most general methods are checked last. This operation requires a queue in which the candidate types are stored. An additional table is kept of all types which have already been checked to ensure no type is checked twice.

Below is a message passing example which illustrates the use of the algorithm. Assume there is the following type hierarchy in the database, as described in Section 3.2.1. Type *Publication* contains information about all publications, such as publication date, title, publisher, etc. The behaviour of this type contains a method *Date*, which returns the publication date. There is also a type called *Journal* which is a subtype of type *Publication*. The *Date* message is passed to a journal as follows:

```
J1.Date
```

and defined by the following method:

```
(Method[Owner:- =>Publication, Param:- Nil, Result:- Date],
   (name:- 'Date'.
   source:-'
      |/   return(this!2.date);          /|'
   code:- ...;))
```

When this expression is resolved, the associated *exp* and *opexp* structures have the following values:

exp: (Determined, =>Journal)
opexp: (message, 'Date');

1. The typing of *J1* is determined to be an instance of type *Journal.*
2. From the set of methods in the behaviour of type *Journal*, it is ascertained that there is no method with name *Date.*
5. All of type *Journal's* supertypes are added to a queue. The queue contains type *Refereed Publication.*
2. From the set of methods in the behaviour of type *Refereed Publication*, it is ascertained that there is no method with name *Date.*
5. All of type *Refereed Publication's* supertypes are added to a queue. The queue contains types *Edited Publication* and *Periodical.*
2. From the set of methods in the behaviour of type *Edited Publication*, it is ascertained that there is no method with name *Date.*
5. All of type *Edited Publication's* supertypes are added to a queue. The queue contains types *Periodical* and *Publication.*
2. From the set of methods in the behaviour of type *Periodical*, it is ascertained that their is no method with name *Date.*
5. All of type *Periodical* supertypes are added to a queue. Since *Publication* already appears in the queue, it is not added.
2. Type *Publication* is obtained and its set of methods are checked. The method with name *Date* is found. This is a candidate method.
3. The structure of method *Date* requires an object of type *Publication.* Since the message is being passed to an object of type *Journal*, this is valid.
4. The method specifies that no parameter is required and none has been given.
6. This is the correct method and the result of the operation is a value of type *Date* as specified in the method type.

result: (Determined, Date)

5.4.3.3 Apply-to-all

The apply-to-all operation requires either a set or list value, an operation and an optional expression. The set or list is held in the expression structure, while the operation and optional expression are held in the *opexp* structure. The expression structure is resolved and tested for either a set or list value. The component of the set or list structure is then used as a typing structure. The operation which is to be applied is obtained from the *opexp* structure. This operation and the typing structure are then passed through the resolution process. The application of the operation to the components in the list or set and the optional expression are thus validated. This resolution process, if successful, yields a result which is the typing structure for the result of the applied operation. This result is now used as the component structure for either a set or list structure, depending upon the original structure, which forms the result of the apply-to-all operation. If there is an error, then the result is the error structure. Consider the following example.

[J1, J2, J3, J4, ...] <= .Date

exp: (Determined, List(=>Journal))
opexp: (apply-to-all, (message, Date))

Given the expression above with the corresponding typing structures, this operation is resolved as follows. The operation is apply-to-all and the expression is typed by a determined list value. From the expression, the component structure of the list is obtained, namely, *=>Journal*, and from the *opexp* structure, the operation is obtained, namely *.Date*. These are used to form the following expression and *opexp* structures.

exp: (Determined, =>Journal)
opexp: (message, Date)

This operation is type checked in exactly the same manner as was illustrated in the previous section on message passing. The result of the operation is a Date value.

result: (Determined, Date)

This result forms the list component for the result of the apply-to-all operation. That is,

result: (Determined, List(Date))

Thus the operation is valid.

5.4.3.4 Insertion

The insertion operation is used to insert an operation in between successive components in either a set or list value. The operation to be inserted is determined by the operation in the *opexp* structure. The

expression structure is required to be either a set or list value which is checked once the expression structure has been determined. The next step is to resolve the operation which is to be inserted.

From the determined expression structure, the component structure for the set or list is obtained. From the *opexp* structure, the operation is obtained. A new expression structure is created using the component structure, while a new *opexp* structure is generated using the operation and the component structure. Thus the component structure is resolved against itself. The resolution process now determines if this operation can be used with these two expressions. If it is valid, then the result of the resolution is the typing structure of the operation's result.

Since the operation is inserted between the result and the next component, it is necessary to resolve the result of the operation and the component. A new pair of expression and *opexp* structures is created. The expression structure is generated from the result structure. The *opexp* structure is generated from the component structure and from the inserted operation. Once again the resolution process is used to validate this operation. If successful, it means that the operation can be inserted between two components in the set or list and will yield a result which can be used as its own argument. The result of the insertion operation is just the typing structure for the result of the inserted operation, provided both checks are successful.

For example, given a determined list of journals, the message *Newer* may be inserted as follows:

```
[J1, J2, J3, J4, ...] / .Newer()
```

Associated with this expression are the following typing structures:

exp: (Determined, List(=>Journal))
opexp: (Insertion, (Message param, Newer)

The expression is a determined list structure and thus passes the first check. Next the component structure is resolved with the inserted operation. The two generated typing structures are as follows:

exp: (Determined, =>Journal)
opexp: (Message param, Newer, (Determined, =>Journal))

These two typing structures are resolved in a similar manner to the date example earlier. The operation is valid since the method *Newer* is defined for type *Publication*. The resulting typing structure is as follows:

result: (Determined, =>Publication)

This structure is now used with the component structure in the second check. The generated structures are as follows:

exp: (Determined, =>Publication)
opexp: (Message param, Newer, (Determined, =>Journal))

This operation is also valid since the *Newer* method requires two objects which are instances of type *Publication* and thus the insertion operation is valid. The typing structure for its result is an object reference to *Publication*.

result: (Determined, =>Publication)

5.5 The Abstract Machine

The compiler produces code for an abstract machine. This code is then interpreted by the abstract machine which performs the specified operations on the database. In this section, we only highlight some of the features of the abstract machine. A full list of machine instructions is given in Appendix C.

The abstract machine is a stack machine which makes use of a six-address code instruction. For most instructions addresses are paired to represent left argument and right argument. The one address is used for the value, and the other address is used for the structure that defines the value. The structure is required because of the complicated values which are manipulated by the machine.

For example, the shallow equality operation below requires the addresses for two values and their corresponding structures. The address of the superstructure and address for the result are also required. The superstructure is required in order to define the structure of the values which shallow equality must compare. Shallow equality is then defined as an inclusion polymorphism for the substructure value. Assume we have the following expression:

(author:- P, title:- 'Object....', appears-in:- J) =+= (author:- Q, appears-in:- K)

where *P* and *Q* are *Person* objects and *J* and *K* are *Journal* objects. In memory are the following values:

T_1 = (author:- P, title:- 'Object....', appears-in:- J)
S_1 = (author:- =>Person, title:- String, appears-in:- =>Journal)
T_2 = (author:- Q, appears-in:- K)
S_2 = (author:- =>Person, appears-in:- =>Journal)
S_0 = superstructure of S_1 and S_2
R = result

The above expression translates to the following instruction:

shall equals S_0, (T_1, S_1), (T_2, S_2), R

S_0 is either S_1 or S_2 depending upon which is the superstructure, in this case S_0 is S_2 and *R* is the address where the boolean result is stored.

Each method has data and code segments. The data segment stores the constants and structures which are required by the instructions in the code segment. The code segment contains a sequence of

instructions which are to be interpreted by the machine. Each instruction has an offset from the beginning of the code segment. The instruction pointer *IP* stores the address of the current code segment and an offset into that code segment. The offsets are used by jump and call operations to change the sequence of instructions, by changing the offset in *IP*. The default action for the *IP* is to increment the offset to access the next instruction. As an example, consider the apply-to-all operation:

$[a_1, ..., a_n]$ <= opexp

which requires a loop to traverse the list or set of values. Each value is then passed to the sequence of instructions which implements the applied operation. The following temporaries are used with the instructions:

a — the list
t_1 — current node in the list
t_2 — value to be processed by the opexp at l_1
s_2 — structure of the component
t_3 — is the result of apply-to-all operation
t_4 — result of the opexp

```
        assign          t1, a   ;   a holds the first node of the list
        assign null     t3      ;   result of apply-to-all operation
l3:     jump zero       t1, l2  ;   if the list is empty jump to the end
        list value      t2, t1  ;   obtain ai from the list node
        call            l1      ;   jump to the instructions at l1
        list append     t3, t4  ;   result of opexp is in t4 which is appended to the result
        list next       t1, t1  ;   the next node in the list is obtained
        jump            l3      ;   jump to the beginning and apply the operation to the next component

l1:     (opexp instruction)     ;   requires t2 as the input parameter
        assign          t4, ti  ;   result of instruction is in ti and is assigned to t4
        return
l2:     null                    ;   end of the operation
```

When a message is passed, an *activation record* is placed on the stack. This saves the information from the previous method and sets up the pointers for the new method. There are three pointers associated with the stack. *Top* is the top of the stack and indicates where new information may be pushed onto the stack. *Base* is the address in the stack where the current activation record begins. From this pointer, offsets are calculated to address values stored in the activation stack. *DS* is the data segment pointer which points to the position in the activation record where the data segment for this activation begins. An activation record on the stack is depicted below.

previous activations

Base: Address where the result of the method is to be stored

System information for the activation record above

Instruction pointer — address of the code and the offset in the code

Base and DS pointers

DS: This — the address for the recipient of the message pass

Param — the address of the parameter used in the message pass

Var_1 ... Var_n — variables defined in the method

$Temp_1$ ... $Temp_n$ — temporaries required by the method

end of activation record

return address for a call operation

Top:

The structures and constants which are stored in a method's data segment are never pushed onto the stack, since this data is constant. These values are addressed by the instructions as offsets into the current method's data segment. There may be many activations of a method, all of which share the same constants and structures. This saves both time and space when interpreting methods.

A method is executed by the instruction *pass object*, which requires an instance and its structure, the identity of the method object, the optional parameter and its structure, and the address for the result. That is,

pass object t_1, s_1, id, t_2, s_2, r

pass object t_1, s_1, id, r

where t_1 is the instance (with structure s_1) and t_2 is the optional parameter (with structure s_2), *id* is the identity and *r* is the result. This operation pushes a new activation record onto the stack by saving the pointers from the previous activation record. The identity of the method object is used to access the method, the method functions (see Section 4.3.6) are used to obtain the code and data segments, *IP* is set to the beginning of the code segment, and the instructions are interpreted. If a *Return* statement is used with a return value in a method, the *return* instruction is used to copy the value to the return address specified at the top of the activation record. At the end of the method, a *return pass* instruction occurs, which pops the activation record off the stack and resets the pointers to their previous values.

5.6 Summary

Our implementation provides the basic structures for representing the values and structures which are defined in the model. Operations are defined to transfer them to and from disk and perform the actions which are defined in the model, although these have not all been presented here. The representations

are recursive in nature and allow complex structures and values to be generated. An object table is used to implement identities and facilitate persistence.

The syntax for a simple method language has been defined, and typing solutions have been presented which allow for the overloading of operations as well as the higher order operations apply-to-all and insertion. Complex values may be denoted and their typing structures are inferred, as is the typing of structures which may be used as values. Methods too may be used as values and typed by a method structure. The parametric polymorphic constant Nil may be used freely in the language and its typing structure is inferred from its context.

The type resolution process makes use of generated typing structures, which store all relevant information relating to the values and operations. The resolution process checks and infers typing structures for the values. This process is used to validate all the operations in the language. A compiler has been written to perform this type resolution and to produce code for an abstract machine.

The abstract machine is a stack machine with six-address code that is an extended form of three-address code. The primitive operations which are defined in the model are defined generally by instructions or templates of instructions in the machine code. The complete abstract machine has been defined (see Appendix C), but most of the primitive operations have not been implemented.

Chapter 6 Conclusion

In this thesis we have presented a model for an object oriented-database and the associated implementation. The model provides a powerful set of structures and extensible types which can be used to store data in a manner which corresponds with the entities in the real world. The system is easier for the user to conceptualize due to this direct correspondence between objects in the system and entities in the real world. Through its extensibility, it can adapt to keep pace with the changes in the real world. By making use of higher-order and meta constructs, such as higher-order structures in the structure system, metatypes and generic types, we specify a data model that is reflective and uniform. All operations in the model are performed in a similar manner which makes the system simple to use. The model is totally extensible, since new types may be defined at will, base types may be implemented outside the system and then linked into the system, metatypes may be used to extend the types in the model, and generic types can be used to extend the constructors which the model provides.

The model supports modularity and encapsulation which make the maintenance of an application simpler. A type contains the implementation of its instances. All operations are performed by methods defined in the type's behaviour; thus all access to the internal structure of an instance is localized to its type.

The implementation has demonstrated the suitability of the constructs in the model. It has provided a means of representing the values defined in the structure system, both in memory and on disk. A simple language has been defined to express the operations of the model. The constraints of the type system and the validity of operations are checked by type checking and type resolution processes. The compiler performs these operations and produces code for an abstract machine, which has been defined with operations to manipulate the database and to perform dynamic types checks.

Clearly there are a number of features which are missing from the system which form natural extensions to it. The database features of reliability, concurrency and recovery are missing and are essential for turning this system into a full database management system. The model can also be extended with a number of the features dealt with in the background chapter, such as versions, composite objects and histories.

As far as the implementation is concerned, future work first involves the implementation of all the abstract machine instructions. The next step is to implement all of the structures which are defined in the model. A user-friendly interface is also required on top of the system to access databases and to perform operations such as queries. The system also lends itself to a graphical browser which can be used to navigate from one object to another while inspecting its contents. Finally, a debugger for the abstract machine will be essential once more complicated code is written.

Appendix A Reference List Example

The types we have described in Section 4.4 are used to set up and define the database system. Once these types and instances are in place, we can use them to model concepts from the real world. The reference list example is described in Section 3.2.1. The types and objects required to model it are presented below. A group of types is defined to model publications. The type *Publication* models a publication in general. It defines data and methods associated with all publications, such as its title, date of publication and publisher.

```
(Type,
    (name:- 'Publication',
    state:- Product(=>Type,
        (title: String,
        date: Date,
        publisher: =>Organization))
    behaviour:- {Print, Volume, Date, Newer, ...},
    objects:- {...},
    subs:- {Authored Publication, Edited Publication, Periodical},
    supers:- {Instance}))
```

A number of subtypes are defined which model specific publications such as authored publications, edited publications and periodicals. The subtyping hierarchy (see Figure 3.4) depicts this classification of the different publications. The *Authored Publication* type adds the attribute 'author' to the above record structure, that is,

```
(Type,
    (name:- 'Authored Publication',
    state:- Product(=>Type,
        (title: String
        date: Date,
        publisher: =>Organization,
        author: =>Person))
    behaviour:- {...},
    objects:- {...},
    subs:- {Book, Report},
    supers:- {Publication}))
```

Type *Book* is an authored publication with the addition of edition and ISBN numbers and is defined as follows:

```
(Type,
    (name:- 'Book',
    state:- Product(=>Type,
        (title: String,
        date: Date,
        publisher: =>Organization,
        author: =>Person,
        edition: Integer,
        isbn: String))
    behaviour:- {...},
    objects:- {Book1},
    subs:- {},
    supers:- {Authored Publication}))
```

The object *Book1* is an instance of this type and is defined as follows:

```
(Book,
    (title:- 'All you wanted to know about OODBs',
    date:- 3/5/90,
    publisher:- Pub1,
    author:- Person1,
    edition:- 1,
    isbn:- '0-7131-3898-3'))
```

An edited publication is a publication, and adds attributes to store the editor and the contents of the publication.

```
(Type,
    (name:- 'Edited Publication',
    state:- Product(=>Type,
        (title: String,
        date: Date,
        publisher: =>Organization,
        editor: =>Person,
        contents: List((work: =>Article, pages: Product(Integer, Integer))))
    behaviour:- {...},
    objects:- {...},
    subs:- {Edited Book, Refereed Publication, Magazine},
    supers:- {Publication}))
```

A periodical is also a publication with the additional information for its ISSN (International Standard Serial Numbering system) and is defined as follows:

```
(Type,
    (name:- 'Periodical',
    state:- Product(=>Type,
        (title: String,
        date: Date,
        publisher: =>Organization,
        issn: String))
    behaviour:- {...},
    objects:- {...},
    subs:- {Report, Refereed Publication, Magazine}},
    supers:- {Publication}))
```

The *Proceedings* type is a specialization of type *Refereed Publication* and adds the attributes for the conference address and the conference date. It is defined as follows:

```
(Type,
    (name:- 'Proceedings',
    state:- Product(=>Type,
        (title: String, date: Date,
        publisher: =>Organization, editor: =>Person,
        contents: List((work: =>Paper, pages: Product(Integer, Integer)),
        issn: String,    conf-add: String, conf-date: Date))
    behaviour:- {...},
    objects:- {Proc1},
    subs:- {},
    supers:- {Refereed Publication}))
```

The ACM proceedings is an instance of this type and is specified as follows:

```
(Proceedings, (title:- 'Proc. of the 1990 ACM SIGMOD Inter. Conf. on Management of Data',
    date:- 1/6/90,
    publisher:- Pub1,
    editor:- Person2,
    contents:- [(work:- Paper1, pages:- (1, 35)), ...],
    issn:- '2367-34578',
    conf-add:- 'Atlantic City, New Jersy',
    conf-date:- 4/6/90))
```

Type *Journal* is also a specialization of an *Refereed Publication*, but it adds the attributes 'volume' and 'number'.

```
(Type,
    (name:- 'Journal',
    state:- Product(=>Type,
        (title: String, date: Date,
        publisher: =>Organization, editor: =>Person,
        contents: List((work: =>Paper, pages: Product(Integer, Integer)),
        issn: String, volume: Integer, number: Integer))
    behaviour:- {...},
    objects:- {...},
    subs:- {},
    supers:- {Refereed Publication}))
```

Type *Article* models the concept of an article which appears in a magazine, newspaper, etc. It defines instances that are specific articles. Each instance contains the title of the article, its author and the *Edited Publication* it appears in.

```
(Type,
    (name:- 'Article',
    state:- Product(=>Type,
        (author: =>Person
        title: String,
        appearsIn: =>Edited Publication))
    behaviour:- {...},
    objects:- {...},
    subs:- {Paper},
    supers:- {Instance}))
```

Type *Paper* is a specialization of type *Article* since it contains the same information but for refereed papers only, which appear in *Refereed Publications*.

```
(Type,
    (name:- 'Paper',
    state:- Product(=>Type,
        (author: =>Person
        title: String,
        appearsIn: =>Periodical))
    behaviour:- {...},
    objects:- {...},
    subs:- {},
    supers:- {Article}))
```

Type *Reference List* defines a reference list consisting of a list of references to publications and articles, as follows:

```
(Type,
    (name:- 'ReferenceList',
    state:- Product(=>Type,
        List(Sum(=>Publication, =>Article))),
    behaviour:- {...},
    objects:- {Ref1},
    subs:- {},
    supers:- {Instance}))
```

The reference list *Ref1* is specified as follows:

```
(ReferenceList,
    [Paper1, Book1, ...])
```

Appendix B Structure System

Listed in the tables below are the various structures, values and their respective operations.

Constructor	*Structure*
Base	*Base*[–] *Base*[*i*]
Product	*Product*(A_1, ..., A_n)
Record	(L_1:A_1, ..., L_n:A_n)
Sum	*Sum*(A_1, ..., A_n)
Set	*Set*(*A*)
List	*List*(*A*)
Array	*Array*(*A*, *T*)
Method	*Method*(*R*, *P*, *C*)
Value Reference	*A*
Object Reference	=>*A*
Obj Ref Generic	=>*A*[a_1:–s_1, ..., a_n:–s_n]
Nil	*Nil*
Gen	*Gen*
GPLS	[a_1:S_1, ..., a_n:S_n]
GPLV	[a_1:–s_1, ..., a_n:–s_n]
Struct	*Struct*
StructLim	*StructLim*
Ref	*Ref*

Constructor	*Structural Equivalence*
Base	$Base[i] \equiv_s Base[j]$ *iff* $i = j$
Product	$Product(A_1, ..., A_n) \equiv_s Product(B_1, ..., B_n)$ *iff*
Record	$(L_1{:}A_1, ..., L_n{:}A_n) \equiv_s (K_1{:}B_1, ..., K_n{:}B_n)$ *iff* $\forall L_i \exists! K_j\ L_i = K_j \wedge A_i \equiv_s B_j$
Sum	$Sum(A_1, ..., A_n) \equiv_s Sum(B_1, ..., B_n)$ *iff* $\forall A_i \exists! B_j\ A_i \equiv_s B_j$
Set	$Set(A) \equiv_s Set(B)$ *iff* $A \equiv_s B$
List	$List(A) \equiv_s List(B)$ *iff* $A \equiv_s B$
Array	$Array(A, T) \equiv_s Array(B, S)$ *iff* $A \equiv_s B \wedge T \equiv_s S$
Method	$Method(R, P, C) \equiv_s Method(S, Q, D)$ *iff* $R \equiv_s S \wedge P \equiv_s Q \wedge C \equiv_s D$
Value Reference	$A \equiv_s B$ *iff* $A \equiv B$
Object Reference	$=>A \equiv_s =>B$ *iff* $A \equiv B$
Obj Ref Generic	$=>[a_1{:-}s_1, ..., a_n{:-}s_n] \equiv_s =>[b_1{:-}t_1, ..., b_n{:-}t_n]$ *iff* $[a_1{:-}s_1, ..., a_n{:-}s_n] \equiv_s [b_1{:-}t_1, ..., b_n{:-}t_n]$
Nil	$Nil \equiv_s Nil$
Gen	$Gen \equiv_s Gen$
GPLS	$[a_1{:}S_1, ..., a_n{:}S_n] \equiv_s [b_1{:}T_1, ..., b_n{:}T_n]$ *iff* $\forall a_i \exists! b_j\ a_i = b_j \wedge S_i \equiv_s T_j$
GPLV	$[a_1{:-}s_1, ..., a_n{:-}s_n] \equiv_s [b_1{:-}t_1, ..., b_n{:-}t_n]$ *iff* $\forall a_i \exists! b_j\ a_i = b_j \wedge (s_i \equiv_s t_j \vee s_i = t_j)$
Struct	$Struct \equiv_s Struct$
StructLim	$StructLim \equiv_s StructLim$
Ref	$Ref \equiv {}_sRef$

Constructor	*Structural Subtyping*
Base	$Base[i] \leq_s Base[j]$ *iff* $i = j$
Product	$Product(B_1, ..., B_m) \leq_s Product(A_1, ..., A_n)$ *iff* $\forall i \in [1..n]\ B_i \leq_s A_i$
Record	$(K_1{:}B_1, ..., K_m{:}B_m) \leq_s (L_1{:}A_1, ..., L_n{:}A_n)$ *iff* $\forall L_i\ \exists!K_j\ (L_i = K_j \wedge B_j \leq_s A_j)$
Sum	$Sum(B_1, ..., B_m) \leq_s Sum(A_1, ..., A_n)$ *iff* $\forall B_j\ \exists A_i\ (B_j \leq_s A_i)\ \wedge \forall B_i \neg\exists B_j\ (B_j \leq_s B_i)$
Set	$Set(A) \leq_s Set(B)$ *iff* $A \leq_s B$
List	$List(A) \leq_s List(B)$ *iff* $A \leq_s B$
Array	$Array(A, T) \leq_s Array(B, S)$ *iff* $A \leq_s B \wedge T \leq_s S$
Method	$Method(S, Q, D) \leq_s Method(R, P, C)$ *iff* $R \leq_s S \wedge P \leq_s Q \wedge D \leq_s C$
Value Reference	$A \leq_s B$ *iff* $A \leq B$
Object Reference	$\Rightarrow A \equiv_s \Rightarrow B$ *iff* $A \leq B$
Obj Ref Generic	$\Rightarrow[a_1{:-}s_1, ..., a_n{:-}s_n] \leq_s \Rightarrow[b_1{:-}t_1, ..., b_n{:-}t_n]$ *iff* $[a_1{:-}s_1, ..., a_n{:-}s_n] \leq_s [b_1{:-}t_1, ..., b_n{:-}t_n]$
Nil	*any structure* $\leq_s Nil$
Gen	$Gen \leq_s Gen$
GPLS	$[b_1{:}T_1, ..., b_m{:}T_m] \leq_s [a_1{:}S_1, ..., a_n{:}S_n]$ *iff* $\forall a_i\ \exists!b_j\ (a_i = b_j \wedge T_i \leq_s S_j)$
GPLV	$[a_1{:-}s_1, ..., a_n{:-}s_n] \leq_s [b_1{:-}t_1, ..., b_n{:-}t_n]$ *iff* $\forall a_i\ \exists!b_j\ (a_i = b_j \wedge (s_i \leq_s t_j \vee s_i = t_j))$
Struct	$Ref \leq_s StructLim \leq_s Struct$
StructLim	
Ref	

Constructor	*Assembly*	*Nil*
Base	*'abc'*	*''*
Product	$(a_1, ..., a_n)$	*(Nil, ...,Nil)*
Record	$(L_1{:}a_1, ..., L_n{:}a_n)$	$(L_1{:}Nil, ..., L_n{:}Nil)$
Sum	$a_i \uparrow Sum(A_1, ..., A_n) = a$	*Nil*
Set	$\{a_1, ..., a_n\}$	*{}*
List	$[a_1, ..., a_n]$	[]
Array	$[= a_1, ..., a_n =]$	[==]
Method	*as per the method language*	*Nil*
References	*only by messages*	*Nil*
Nil	*has no value*	-
Gen	*defined by GPLS*	*GPLS with Nil structures*
GPLS	*defined by GPLV*	*GPLV with Nil values*
Struct	*any structure*	*Nil Structure*
StructLim		
Ref		

Constructor	*Disassembly*	***Relational,** all have equality, shallow and deep, and inequality*
Base	*may be defined externally*	*other external*
Product	$(a_1, ..., a_n)!i = a_i$	-
Record	$(L_1{:}a_1, ..., L_n{:}a_n).L_i = a_i$	-
Sum	$a\ ?[method_1, ..., method_n](p) = a_i$ *$method_i$ is defined for component A_i*	*Nil*
Set	-	*element–of* $\in$, *subset* $\subseteq$, *proper–subset* $\subset$, *superset* $\supseteq$, and *proper superset* $\supset$
List	$[a_1, ..., a_n].head = a_1$ $[a_1, ..., a_n].tail = [a_2, ..., a_n]$	*element–of* $\in$, *sublist* $\leq$, *proper sublist* $<$, *superlist* $\geq$, *proper superlist* $>$
Array	$[= a_1, ..., a_n =]!i = a_i$	-
Method	*message passing*	*identical*
References	*only by messages*	=–=
GPLS	*when parameters are used*	
Struct	*similar to operations above, except applied to a structure and returns a component structure*	*structural equivalence* = *structural subtype*, $\leq$
StructLim		
Ref		

Constructor	*Translation*
Base	*may be defined externally*
Product	$(a_1, ..., a_n)$: $(B_1, ..., B_m) = ((a_1{:}B_1), ..., (a_n{:}B_n), Nil, ..., Nil)$ $(b_1, ..., b_m)$: $(A_1, ..., A_n) = ((b_1{:}A_1), ..., (b_n{:}A_n))$
Record	$(L_1{:-}a_1, ..., L_n{:-}a_n){:}(L_1{:}B_1, ..., L_m{:}B_m) =$ $((L_1{:-}(a_1{:}B_1), ..., L_n{:-}(a_n{:}B_n), ..., L_m{:-}Nil)$ $(L_1{:-}b_1, ..., L_m{:-}b_m){:}(L_1{:}A_1, ..., L_n{:}A_n) =$ $((L_1{:-}(b_1{:}A_1), ..., L_n{:-}(b_n{:}A_n))$
Sum	$a{:}Sum(B_1, ..., B_m) = a_i{:}B_j$
Set	$\{a_1, ..., a_n\}{:}Set(B) = \{a_1{:}B, ..., a_n{:}B\}$
List	$[a_1, ..., a_n]{:}List(B) = [a_1{:}B, ..., a_n{:}B]$
Array	$[= a_1, ..., a_n =]{:}Array(B, T) = [= a_1{:}B, ..., a_n{:}B =]$
Method	*method may only be translated to a substructure* *the code remains the same.*
References	*if the reference is an element of the new structure, then it remains, else it is removed and set to Nil*
Gen	*identity translation*
GPLS	*similar to record*
Struct	*identity translation*
StructLim	
Ref	

Constructor	*Other Operations*
Base	*all defined externally*
Product	-
Record	-
Sum	-
Set	*intersection* ∩, *union* ∪, *difference* –, *insert*, *delete*, *apply–to–all* <=, *size*, *singleton*, *insertion* /
List	*concatenate*, *apply–to–all* <=, *insertion* /
Array	*apply–to–all* <=, *insertion* /
Method	-
Value Reference	-
Object Reference	-
Obj Ref Generic	-
Nil	-
Gen	-
GPLS	-
GPLV	-
Struct	*Multiple Inheritance*
StructLim	
Ref	

Appendix C Abstract Machine Instructions

Listed below are the various instructions which are defined for an abstract machine. The instructions make use of six-addresses and we use t_i to denote a temporary value, s_i to denote a structure, r to denote the result of the operation, and l_i to denote the offset of an instruction line. There are two kinds of addresses which may be used: a value address (*val*) is a pointer to a value, and a reference address (*ref*) is a handle to a value (i.e a pointer to the pointer to the value). There are a number of values which can be used instead of addresses in an instruction: *idP* is a pointer to an object identity and a GPLV, *bool* is a boolean value and *int* is an integer value. A NULL pointer is used in addition to any of the defined Nil values.

Equality

identical	t_1, t_2, r	t_i - idPs, r - bool
shall equal	s_0, t_1, s_1, t_2, s_2, r	s_0 - super of s_1, s_2. s_1 - structure of t_1
deep equal	s_0, t_1, s_1, t_2, s_2, r	s_2 - structure of t_2.
not equal	s_0, t_1, s_1, t_2, s_2, r	r - result - boolean - val

Product

prod make	n, r	product of size n is made and assigned to r - val
prod subscript	t_1, i, r	comp i of product t_1 is placed in r - ref
prod rep	t_1, i, t_2	place value t_2 in product t_1 at comp i

Record

rec make	n, r	record of size n is made r - val
rec sbscrpt	t_1, s_1, attr, r	attribute attr of record t_1 is placed in r - ref
rec rep	t_1, s_1, attr, t_2	place value t_2 in record t_1 at comp attr
rec rep pos	t_1, i, t_2	place in pos i of t_1 val t_2

Set

intersection	$s_0, t_1, s_1, t_2, s_2, r$	result has structure s_0 and contains a copy of the values which occur in both t_1, t_2 - val
set union	$s_0, t_1, s_1, t_2, s_2, r$	"
set diff	$s_0, t_1, s_1, t_2, s_2, r$	"
set add (ins)	t_1, s_1, t_2, s_2	t_1-s_2 set, t_2-s_2 node is translated to a new value which is added to the set value: if t_1 is Null replace with the new set
set del	t_1, s_1, t_2, s_2	the value t_2 is removed from set t_1
set singleton	t_1, r	set t_1 copies a single element to r - va l
set size	t_1, r	r contains the number of elements in set t_1 - val int
set elem	t_1, s_1, t_2, s_2, r	determines if t_2 is an element in t_1 - val bool
subset	$s_0, t_1, s_1, t_2, s_2, r$	"
proper subset	$s_0, t_1, s_1, t_2, s_2, r$	"
superset	$s_0, t_1, s_1, t_2, s_2, r$	"
proper superset	$s_0, t_1, s_1, t_2, s_2, r$	"
set next	t_1, t_2, t_3	t_3 is the set structure, t_2 is the stack containing traversal, t_1 is the current node in the tree.
set value	t_1, t_2	t_2 is a node in the set, its value placed in t_1

List

list value	t_1, t_2	t_2 is a list node and its value is assigned to t_2
list next	t_1, t_2	t_2 a list node and the next list node is t_1
list append	t_1, t_2	t_1 is a list the value in t_2 is added to the end of the list, if t_1 is null the head is placed in t_1
list concat	$s_0, t_1, s_1, t_2, s_2, r$	new list is placed in r with st s_0 - val copies t_1, t_2
list tail	t_1, r	no new value, ptr to tail - ref
list head	t_1, r	r points to the head value of list t_1 - ref
list size	t_1, r	val int
list elem	t_1, s_1, t_2, s_2, r	determines if t_2 is an element of list t_1 - va l bool
sublist	$s_0, t_1, s_1, t_2, s_2, r$	"
proper sublist	$s_0, t_1, s_1, t_2, s_2, r$	"
superlist	$s_0, t_1, s_1, t_2, s_2, r$	"
proper superlist	$s_0, t_1, s_1, t_2, s_2, r$	"

Method

compile	t_1, s_1, r	t_1 is a string value, s_1 is the method structure, result dynamic, determined by source typing params - val
pass value	t_1, s_1, t_2, s_2, r	t_1, s_1 is the method value and its structure, t_2, s_2 is the recipient of the message and the optional paramter with the corresponding structure, r is the result of pass - val

Structures

st product	t_1, r	new product structure - val
st concat	t_1, t_2, r	add product t_2 to the end of t_1 and give a new structure in r - val
st concat	t_1, t_2, r	add record t_2 to t_1 and give a new structure in r - val
st prod sbscrpt	t_1, i, r	give the structure in comp i - ref
st arity	t_1, r	number of components in the product structure t_1 - val
st arity	t_1, r	number of components in the record structure t_1 - val
st comps	t_1, r	convert product into a list of structures - val
st sub product	t_1, t_2, r	produces a copy of the product structure in the range def by t_2 - val
st record	t_1, t_2, r	list of structures t_1 and list of str t_2 used to make a record structure r - val
st attr	t_1, r	r - list of strings (copy) used in record structure t_1 - val
st rec sbscrpt	t_1, attr, r	r contains the structure of attr - ref
st sub record	t_1, t_2, r	new record from the t_1 with att in t_2 - val
st set	t_1, r	set structure with copy of t_1 - val
st comp	t_1, r	comp of set structure, ptr - ref
st comp	t_1, r	comp of list structure, ptr - ref
st list	t_1, r	list structure with t_1 - val
st method	t_1, t_2, r	new method structure t_i - val
st owner	t_1, r	owner of method structure - ref
st param	t_1, r	param of method structure - ref
st result	t_1, r	result of method structure - ref
st object	t_1, r	new object ref structure - val
st type	t_1, r	type used in a ref structure - ref
st gplv	t_1, r	gplv used in a ref structure - ref
st MultInherit	t_1, r	determines a structure which is a sub-structure of all in t_1 - val
substruct	t_1, t_2, r	determine if t_1 is a sub-structure of t_2 - boolean - val

Objects, Types, Instances

new	t_1, t_2, r	generates an object with structure t_1 at level t_2, r has idP - val
level	t_1, r	determines the level of object t_1 - val
convert	t_1, t_2	converts object t_1 to level t_2 - nul
delete	t_1	deletes object t_1 - nul
clear	t_1	removes all objects from level t_1 - nul
clearAbove	t_1	removes all object above level t_1 - nul
merge	t_1, t_2, r	converts two objects into one object, returns id of new obejct
subtype	t_1, t_2, r	determine if t_1 subtype of t_2 - boolean
pass object	t_1, s_1, id, t_2, s_2, r	execute a message pass with a param - val
pass object	t_1, s_1, id, r	execute a message pass with no params - val
copy id	t_1, t_2	t_1 - ref, t_2 - val
copy shallow	t_1, s_1, t_2, s_2	t_1 - ref, nul -- translates t_2 to s_1
obj 2 inst	t_1, t_2	t_1 - IdP for an object, places the instance in t_2
obj 2 inst ref	t_1, t_2	t_1 contains the IdP of the object, a ref to the entire instance is placed in t_2. ptr to the portion of the id table which points to the object.
obj 2 val ref	t_1, t_2	t_1 contains the IdP of the object, places a reference to the value portion of the instance in t_2. see fn val i

Control Operations

jump	l_1	control passes to line l_1
jump non	t_1, l_1	if the value in t_1 is not zero ctrl pass to l_1
jump zero	t_1, l_1	used iternal for ptrs
jump bool t	t_1, l_1	t_1 points to a bool value, if it is true jump to l_1
jump bool f	t_1, l_1	t_1 points to a bool value, if it is false jump to l_1
return pass	t_1, s_1	return value t_1 with stru s_1 in a message pass
base linked	op, t_1, t_2, r	external linked code with operator number op is passed values t_1, t_2 result placed in r.
call	l_1	jumps to line l_1 and saves the next line.
return call		jumps to the line saved by the call

assign	t_1, t_2	the value-ptr in t_2 is copied to t_1.
assign null	t_1	replaces value in t_1 with null
ref2val	t_1	converts the handle in t_1 to a pointer
val2ref	t_1, t_2	places a ptr to temp t_1 in t_2, used for variables
null		no operation used for the lable jumps.
cast	t_1, s_1, s_2, r	the value t_1 is cast to structure s_2 and new val placed in r
free	t_1, s_1	release the instance t_1 from mem sets t_1 to NULL
free stru	t_1	release the structure t_1 from mem sets t_1 to NULL
substitute	t_1, t_2, t_3	t_1 GPLV, t_2 structure containing params whose pos match GPLV place a new structure in t_3 which is the result of structure substitution
name add	t_1, t_2, r	the value t_1 - string, t_2 - identity, r - bool
name get	t_1, r	t_1 - name, r - identity
name del n	t_1	t_1 - name
name del i	t_1	t_1 - identity

Functions

fn val i	t_1, t_2	value instance fn, t_1 idP for an object, place a ptr to the value portion in t_2
fn val s	t_1, t_2	value structure fn, corresponds with the value instance fn, t_1 is a structure and places int_2 a ptr to the value portion
fn state	t_1, t_2	t_1 requires the idP of a type and places a ptr to its state in t_2
fn gen i	t_1, t_2	t_1 contains idP for an object. Obtain the object and use the type fn to determine the object's type (idP) from which a ptr to the GPLV is placed in t_2.
fn gen	t_1, t_2	t_1 (idP) for a type, if it is generic places a ptr to the GPLS in t_2 as defined by the generic fn.

Dynamic Test

dyn sub rel	s_1, s_2, s_i	dynamic test, determines if there is a subtype relation between s_1, s_2 if so returns it in s_i, else abort the execution because remaining ops will be invalid. Used for check in the structures associated with generic instances.
dyn substru	s_1, s_2	determines if s_2 is a substructure of s_1, if not aborts

Bibliography

We have made use of the following papers and books as background material for the thesis. A number of these works are cited in the thesis.

[ABDDMZ90]

M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, S. Zdonik, "The Object-Oriented Database System Manifesto", in "Deductive and Object-Oriented Databases", W. Kim, J.-M. Nicolas, S. Nishio (eds.), pp. 223-240, North-Holland, 1990, ISBN 0-444-88433-5

[AbKa89]

S. Abitboul and P.C. Kanellakis, "Object Identity as a Query Language Primitive", Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data, Volume 18, Number 2, June 1989, pp. 159-173.

[AgGe89]

R. Agrawal and N.H. Gehani, "ODE (Object Database and Environment): The Language and Data Model", Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data, Volume 18, Number 2, June 1989, pp. 36-45.

[Alle86]

R.B.J.T. Allenby, "Rings, Fields and Groups; An Introduction to Abstract Algebra", Edward Arnold Pub., 1986, ISBN 0-7131-3476-3, pp. 10-82.

[ANSI91]

ANSI, "X3/SPARC/DBSSG/OODBTG Final report", E. Fong, W. Kent, K. Moore, C. Thompson (eds.), September 1991.

[ASU86]

A.V. Aho, R. Sethi and J.D. Ullman, "Compilers: Principles, Techniques, and Tools", Addison-Wesley World Student Series, Addison-Wesley Publishing Company, 1986, ISBN 0-201-10194-7.

[AtBu87]

M.P. Atkinson and O.P. Buneman, "Types and Persistence in Database Programming Languages", ACM Computing Surveys, Volume 19, Number 2, June 1987, pp. 105-190.

[BaKh86]

F. Bancilhon and S. Khoshafian, "A Calculus for Complex Objects", Proc. 5th ACM SIGACT-SIGMOD Symp. on Principles of Database Systems, 1986, pp. 53-59.

[BaLa89]

R.A. Baeza-Yates and P.-A. Larson, "Performance of B^+-Trees with Partial Expansions", IEEE Transactions on Knowledge and Data Engineering, Volume 1, Number 2, June 1989, pp. 248-257.

[Banc88]

F. Bancilhon, "Object-Oriented Database Systems", Proc. 7th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, 1988, pp. 152-162.

[Bana88]

M.F. Banahan, "The C Book: featuring the draft ANSI C Standard", The Instruction Set Series, Addison-Wesley, 1988, ISBN 0-201-17370-0.

[Barn89]

J.G.P. Barnes, "Programming in ADA, Third Edition", International Computer Science Series, Addison-Wesley, 1989, ISBN 0-201-17566-5.

[BBMR89]

A. Borgida, R.J. Brachman, D.L. McGuinness and L.A. Resnick, "CLASSIC: A Structural Data Model for Objects", Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data, Volume 18, Number 2, June 1989, pp. 58-67.

[BCD89]

F. Bancilhon, S. Cluet, C. Delobel, "A Query Language for the O2 Object-Oriented Database System", Proceedings of the 2nd International Workshop on Database Programming Languages, June 1989, pp. 122-138.

[BeKi89]

E. Bertino and W. Kim, "Indexing Techniques for Queries on Nested Objects", IEEE Transactions on Knowledge and Data Engineering, Volume 1, Number 2, June 1989, pp. 196-214.

[BKKK87]

J. Banerjee, W. Kim, H.-J. Kim and H.F. Korth, "Semantics and Implementation of Schema Evolution in Object-Oriented Databases", ACM SIGMOD Conference on the Management of Data, Volume 16, Number 3, 1987, pp. 311-322.

[Borg88]

A. Borgida, "Modelling Class Hierarchies with Contradictions", Proc. ACM SIGMOD Conference on the Management of Data, Volume 17, Number 3, September 1988, pp. 434-443

[Borg89]

A. Borgida, "Type Systems for Querying Class Hierarchies with Non-Strict Inheritance", Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 1989, pp. 394-400.

[Borl88]

Borland, "Turbo Pascal Reference Guide", Borland International, 1800 Green Hills Road, P.O. Box 660001 Scotts Valley, CA 95066-0001, 1988, pp. 201-203.

[BOS91]

P. Butterworth, A. Otis, J. Stein, "The GemStone Object Database Management System", Communications of the ACM, Volume 34, Number 10, October 1991, pp. 64-77.

[Capl87]

M. Caplinger, "An Information System Based on Distributed Objects", ACM Conference on Object-Oriented Programming Systems, Languages and Applications, Volume 22, Number 12, 1987, pp. 126-137.

[CaWe85]

L. Cardelli and P. Wegner, "On Understanding Types, Data Abstractions and Polymorphisms", ACM Computing Surveys, Volume 17, Number 4, December 1985, pp. 471-522.

[CaZd87]

M.J. Caruso and S.B. Zdonik, "Report on the Object Oriented Database Workshop: Semantic Aspects", ACM Conference on Object-Oriented Programming Systems, Languages and Applications (Addendum), 1987, pp. 316-325.

[CDV88]

M.J. Carey, D.J. DeWitt and S.L. Vandenberg, "A Data Model and Query Language for Exodus", Proc. ACM SIGMOD Conference on the Management of Data, Volume 17, Number 3, September 1988, pp. 413-423.

[CePe85]

S. Ceri and G. Pelagatti, "Distributed Databases: Principles & Systems", McGraw-Hill International Editions, 1985.

[ChKa89]

E.E. Chang and R.H. Katz, "Exploiting Inheritance and Structure Semantics for Effective Clustering and Buffering in an Object-Oriented DBMS", Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data, Volume 18, Number 2, June 1989, pp. 348-357.

[CoBe84]

D.J. Cooke and H.E. Bez, "Computer Mathematics", Cambridge Computer Science Texts 18, 1985, ISBN 0-521-27324-2

[CoMa84]

G. Copeland and D. Maier, "Making Smalltalk a Database System", Proceedings of the 1984 ACM SIGMOD Conference on the Management of Data, Volume 14, Number 2, June 1984, pp. 316-325.

[Codd70]

E.F. Codd, "A relational model od data for large shared data banks", Communications of the ACM, Volume 13, Number 6, June 1970, pp. 377-387.

[Coin87]

P. Cointe, "Metaclasses are First Class : the ObjVlisp Model", ACM Conference on Object-Oriented Programming Systems, Languages and Applications, Volume 22, Number 12, 1987, pp. 156-167.

[DaTo88]

S. Danforth and C. Tomlinson, "Type Theories and Object-Oriented Programming", ACM Computing Surveys, Volume 20, Number 1, March 1988, pp.29-72.

[Deux90]

O. Deux et al, "The Story of O2", IEEE Transactions on Knowledge and Data Engineering, Volume 2, Number 1, March 1990, pp. 91-108.

[Deux91]

O. Deux et al, "The O2 System", Communications of the ACM, Volume 34, Number 10, October 1991, pp. 34-48.

[DPSW89]

G.N. Dixon, G.D. Parrington, S.K. Shrivastava and S.M. Weather, "The Treatment of Persistent Objects in Arjuna", The Computer Journal, Volume 32, Number 4, August 1989, pp. 323-332.

[FABC89]

D.H. Fisherman, J. Annevelink, D. Beech, E. Chow, "Overview of the Iris DBMS", in "Object-Oriented Concepts, Databases, and Applications", W. Kim, F.M. Lochovsky (eds.), ACM Press, 1989, ISBN 0-201-14410-7, pp. 219-250.

[FiHa89]

A.J. Field, P.G. Harrison, "Functional Programming", Addison-Wesley, 1989, ISBN 0-201-19249-7.

[GaKi88]

J.F. Garza and W. Kim, "Transaction Management in an Object-Oriented Database System", Proc. ACM SIGMOD Conference on the Management of Data, Volume 17, Number 3, September 1988, pp. 37-45.

[GaKo88]

H. Garcia-Molina and B. Kogan, "Achieving High Availability in Distributed Databases", IEEE Transactions on Software Engineering, Volume 14, Number 7, July 1988, pp.886-896.

[Harp86]

R. Harper, "Introduction to Standard ML", LFCS Report Series, Department of Computer Science, University of Edinburgh, November 1986.

[HMM86]

R. Harper, D. MacQueen and R. Milner, "Standard ML", LFCS Report Series, Department of Computer Science, University of Edinburgh, March 1986.

[HoWo89]

T.P. Hopkin and M.I. Wolczko, "Writing Concurrent Object-Oriented Programs using Smalltalk-80", The Computer Journal, Volume 32, Number 4, August 1989, pp. 341-350.

[KaLe89]

D.G. Kafura and K.H. Lee, "Inheritance in Actor Based Concurrent Object-Oriented Languages", The Computer Journal, Volume 32, Number 4, August 1989, pp. 297-304.

[KBBCGW88]

W. Kim, N. Ballou J. Banerjee, H.-T. Chou, J.F. Garza and D. Woelk, "Integrating an Object-Oriented Programming System with a Database System", ACM Conference on Object-Oriented Programming Systems, Languages and Applications, 1988, pp. 142-152.

[KBCGW87]

W. Kim, J. Banerjee, H.-T. Chou, J.F. Garza and D. Woelk, "Composite Object Support in an Object-Oriented Database System", ACM Conference on Object-Oriented Programming Systems, Languages and Applications, Volume 22, Number 12, 1987, pp. 118-125.

[KBG89]

W. Kim, E. Bertino and J.F. Garza, "Composite Objects Revisited", Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data, Volume 18, Number 2, June 1989, pp. 337-347.

[KCB88]

W. Kim, H.T. Chou and J. Banerjee, "Operations and Implementation of Complex Objects", IEEE Transactions on Software Engineering, Volume 14, Number 7, July 1988, pp. 985-996.

[KGBW90]

W. Kim, J.F. Garza, N. Ballou, D. Woelk, "Architecture of the ORION Next-Generation Database System", IEEE Transactions on Knowledge and Data Engineering, Volume 2, Number 1, March 1990, pp. 109-124.

[Kent89]

W. Kent, "An Overview of the Versioning Problem", Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data, Volume 18, Number 2, June 1989, pp. 5-7.

[KhCo86]

S.N. Khoshafian and G.P. Copeland, "Object Identity", ACM Conference on Object-Oriented Programming Systems, Languages and Applications, 1986, pp. 406-416.

[KiLa89]

M. Kifer and G. Lausen, "F-Logic: A Higher-Order Language for Reasoning about Objects, Inheritance, and Scheme", Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data, Volume 18, Number 2, June 1989, pp. 134-146.

[Kilo89]

H. Kilov, "Reviews of Object-Oriented Papers", SIGMOD RECORD, Volume 18, Number 4, December 1989, pp. 50-55.

[KKD89]

K.-C. Kim, W. Kim and A. Dale, "Cyclic Query Processing in Object-Oriented Databases", Proceedings Fifth International Conference on Data Engineering, IEEE, February 1989, pp. 564-571.

[LéRi89]

C. Lécluse and P. Richard, "Complex Structures in Object-Oriented Databases", Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 1989, pp. 360-368.

[LLOW91]

C. Lamb, G. Landis, J. Orenstein, D. Weinred, "The ObjectStore Database System", Communications of the ACM, Volume 34, Number 10, October 1991, pp. 94-109.

[LLPS91]

G.M. Lohman, B. Lindsay, H. Pirahesh, K.B. Schiefer, "Extentions to Starburst: Objects, Types, Functions, and Rules", Communications of the ACM, Volume 34, Number 10, October 1991, pp. 94-109.

[LRV88]

C. Lécluse, P. Richard and F. Velez, "O_2, an Object-Oriented Data Model", Proc. ACM SIGMOD Conference on the Management of Data, Volume 17, Number 3, September 1988, pp. 424-433.

[Masu90]

Y. Masunaga, "Object Identity, Equality and Relational Concepts", in "Deductive and Object-Oriented Databases", W. Kim, J.M. Nicolas, S. Nishio (eds), North-Holland, 1990, pp. 185-202, ISBN 0-444-88433-5.

[MCB89]

M.V. Mannino, I.J. Choi and D.S. Batory, "An Overview of the Object-Oriented Functional Data Language", Proceedings Fifth International Conference on Data Engineering, IEEE, February 1989, pp. 18-26.

[MCB90]

M.V. Mannino, I.J. Choi and D.S. Batory, "The Object-Oriented Functional Data Language", IEEE Transactions on Software Engineering, November 1990, Volume 16, Number 11, pp. 1258-1272.

[MéGu87]

A. Mével and T. Guguen, "Smalltalk-80", Macmillan Education Ltd, 1987.

[Miln78]

R. Milner, "A Theory of Type Polymorphism in Programming", Journal of Computer and System Sciences, 1978, pp. 348-375.

[MSOP86]

D. Maier, J. Stein, A. Otis and A. Purdy, "Development of an Object-Oriented DBMS", ACM Conference on Object-Oriented Programming Systems, Languages and Applications, 1986, pp. 472-482.

[Osbo89]

S.L. Osborn, "The Role of Polymorphism in Schema Evolution in an Object-Oriented Database", IEEE Transactions on Knowledge and Data Engineering, Volume 1, Number 3, September 1989, pp. 310-317.

[PiWi88]

L.J. Pinson and R.S. Wiener, "An Introduction to Object-Oriented Programming and Smalltalk", Addison-Wesley Publishing Company, 1988.

[PJF89]

G. Pathak, J. Joseph and S. Ford, "Object Exchange Service for an Object-Oriented Database System", Proceedings Fifth International Conference on Data Engineering, IEEE, February 1989, pp. 27-35.

[PNP88]

C. Pu, J.D. Noe and A. Proudfoot, "Regeneration of Replicated Objects: A Technique and Its Eden Implementation", IEEE Transactions on Software Engineering, Volume 14, Number 7, July 1988, pp.936-945

[Read89]

C. Reade, "Elements of functional programming", Addison-Wesley, 1989, ISBN 0-201-12915-9.

[Schm86]

D.A. Schmidt, "Denotational Semantics, A Methodology for Language Development", Allyn and Bacon Inc. 1986, ISBN 0-205-08974-7.

[ShCa89]

E.J. Shekita and M.J. Carey, "Performance Enhancement Through Replication in an Object-Oriented DBMS", Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data, Volume 18, Number 2, June 1989, pp. 325-336.

[Stei87]

L.A. Stein, "Delegation Is Inheritance", ACM Conference on Object-Oriented Programming Systems, Languages and Applications, Volume 22, Number 12, 1987, pp. 138-146.

[StKe91]

M. Stonebraker, G. Kemnitz, "The Postgres Next Generation Database Management System", Communications of the ACM, Volume 34, Number 10, October 1991, pp. 78-92.

[Terr87]

P.D. Terry, "An Introduction to Programming with Modula-2", International Computer Science Series, Addison-Wesley, 1987, ISBN 0-201-17438-3.

[That87]

S.M. Thatte, "Report on the Object-Oriented Database Workshop: Implementation Aspects", ACM Conference on Object-Oriented Programming Systems, Languages and Applications (Addendum), 1987, pp. 73-87.

[Ullm88]

J.D. Ullman, "Principles of Database and Knowledge-Base Systems", Volume I,Computer Science Press, 1989, ISBN 0-7167-8158-1, pp. 524-529.

[Wegn87]

P. Wegner, "Dimensions of Object-Based Language Design", ACM Conference on Object-Oriented Programming Systems, Languages and Applications, 1987, pp. 168-182.

[Weik89]

G. Weikum, "Set-Oriented Disk Access to Large Complex Objects", Proceedings Fifth International Conference on Data Engineering, IEEE, February 1989, pp. 426-433.

[WeLo89]

S.P. Weiser, F.H. Lochovsky, "OZ+: An Object-Oriented Database System", in "Object-Oriented Concepts, Databases, and Applications", W. Kim, F.M. Lochovsky (eds), ACM Press, 1989, ISBN 0-201-14410-7, pp. 309-340.

[Yell89]

P.M. Yelland, "First Steps Towards Fully Abstract Semantics for Object-Oriented Languages", The Computer Journal, Volume 32, Number 4, August 1989, pp. 290-296.

[Zani85]

C. Zaniolo, "The Representation and Deductive Retrieval of Complex Objects", Proc. 11th International Conference on Very Large Data Bases, 1985, pp. 458-469.

[ZdMa90]

S.B. Zdonik, D. Maier, "Fundamentals of Object-Oriented Databases", Readings in Object-Oriented Database Systems, Morgan-Kaufman, 1990, ISBN 1-55860-000-0, pp. 1-32.

[Zdon87]

S.B. Zdonik, "Panel on Object-Oriented Database Systems", ACM Conference on Object-Oriented Programming Systems, Languages and Applications (Addendum), 1987, pp. 69-71.

[Zdon89]

S.B. Zdonik, "An Introduction to Object-Oriented Database Systems", Tutorial presented at 15th International Conference on Very Large Data Bases, 1989.

[ZhRo88]

L. Zhao and S.A. Roberts, "An Object-Oriented Data Model for Database Modelling, Implementation and Access", The Computer Journal, Volume 31, Number 2, April 1988, pp. 116-124.