A Methodology for the Design and Implementation of a Functional Object Database using Haskell

Linda Louise Dawson
(B.Sc. (Syd.), Grad Dip. (Eng. Dev.) (UNSW)

Thesis submitted in fulfilment
of the requirements of the degree of
Master of Science

Department of Computer Science
University of Tasmania

December 1993

Declaration

Except where stated herein, this thesis contains no material which has been accepted for the award of any other degree or diploma in any university. To the best of my knowledge and belief, this thesis contains no copy or paraphrase of material previously published or written by another person, except where due reference has been made in the text of the thesis.

Linda Dawson

Linda Dawson

Access to, and copying of, thesis

Linda Dawson

Linda Dawson

Abstract

Semantic data modelling has been a traditional abstract way of representing data and relationships between data for database systems. Recently database designers and developers have been looking to object oriented modelling methods which incorporate the modelling of behaviour as well as data and their relationships.

Databases can also be considered to be functional structures. All operations on databases are functional in that they return values. Transaction and update operations return a new version of the database; queries return values contained in the database; and reports return values from the database in some strictly formatted form.

This thesis firstly develops an extended version of the Entity-Relationship model, called the Entity-Relationship-Object model (ERO model) that incorporates object oriented concepts including behaviour. Secondly, a methodology is defined for mapping this model directly to the functional programming language, Haskell, where all Entity and Relationship objects are implemented as abstract data types and all attributes and methods are implemented as functions.

The modularity and polymorphism of the proposed models and their implementation allow for easy schema extension and modification. Lazy evaluation in the implementation allows for a simple form of persistent data store.

Acknowledgements

I wish to express my gratitude to my supervisor, Dr Chris Keen, for his support and encouragement in the production of this thesis, particularly during the crucial initial and final stages.

I would also like to thank my colleagues in the Database Research Group for their support and, in particular, Simon Milton.

Thanks also go to my colleagues, Andrew Partridge and David Wright for their assistance with the more esoteric aspects of functional programming.

Table of Contents

1 Introduction

1.1 Overview

This thesis investigates the design and implementation of an object-oriented database system in a functional programming language environment. This involves two major areas of investigation. Firstly, the design of an object-oriented database being based on an object-oriented data model [Dobbie, 1991]. Traditionally the design of databases has been based on semantic data models which capture the definition and meaning of data objects and the relationships between them. Object-oriented modelling methods add behaviour to data models so that database users can manipulate data as well as store, retrieve and share data.

Secondly, the implementation is based on a functional programming environment since databases may be considered to be inherently functional structures in that all operations on databases return values. Transaction and update operations return a new version of the database; queries return values contained in the database; and reports return values from the database in some strictly formatted form. The system developed here deals with update operations and some reporting functions but not query facilities.

Although there are two development aspects addressed in this project, the overall methodology is based on Henderson-Sellers' [1992] general object-oriented software development methodology. Henderson-Sellers [1992] offers a general methodological framework that can variously incorporate functional or object-oriented analysis and object-oriented design together with object-oriented or imperative implementation environments. The methodology here uses object-oriented analysis, object-oriented design and a functional programming language implementation.

1.2 Aims

The aims of this project are twofold. Firstly, semantic data models and object data models are investigated with a view to combining the characteristics of both to produce a model that provides semantic information about both data and its behaviour. This model should lend itself to direct implementation in a programming language that provides a high level of data abstraction, polymorphism and rich typing.

The second aim of this project is to investigate the properties of functional programming languages, using a standard accessible language, Haskell [Hudak, 1992], for the implementation of an object-oriented database.

There are several new ideas presented in this thesis. A new data model, the ERO model, has been developed which combines the characteristics of semantic data models and object data models and also provides a higher level of abstraction than either of these models by treating both entities and relationships as classes of the same polymorphic type. This model is also directly implementable in any language which offers polymorphism, data abstraction in the form of abstract data types (ADTs) and modularity.

This project provides a new variant of Henderson-Sellers' [1992] object-oriented development methodology. This new methodology is applicable to database development, incorporates the ERO model at the design stage and provides a methodology for mapping the ERO model directly to a functional programming language where all entities and relationships are implemented as abstract data types and all attributes and methods are implemented as functions.

The data abstraction, modularity and polymorphism provided in the models and their implementation allow for easy schema extension and modification. Lazy evaluation in the implementation allows for a simple form of persistent data store.

1.3 Definitions

Entity-Relationship (ER) models [Chen, 1976, Hansen and Hansen, 1992] provide semantic information about data objects, called *entities*, or collections of data objects, called *entity classes*, and the relationships between them, called *relationships*.

The Object Modelling Technique (OMT) data model [Rumbaugh et al, 1991] provides semantic information about data objects, *objects*, and polymorphic collections of data objects, *classes,* and relationships between them, called *associations*. OMT models also provide information about the behaviour of objects as *operations* which are implemented as *methods*.

1.4 Literature Summary

The Entity-Relationship Model was first proposed by Chen [1976]. This model has been extended or enhanced by several authors including Elmasri, Weeldreyer and Hevner [1985], Elmasri and Navathe [1989] and Hansen and Hansen [1992]. Nijssen and Halpin [1989] have proposed a fact-oriented data modelling technique. Object data models and notations have been proposed by Rumbaugh et al [1991] and Henderson-Sellers [1992]. Object-oriented design methodologies have been proposed by Beck and Cunningham [1989], Boehm [1988], Booch [1986], Henderson-Sellers [1992] and Shlaer and Mellor [1989]. The characteristics of object-oriented databases have been defined by various authors including Atkinson et al [1990], Dobbie [1991] and Fong et al [1991]. Functional databases were proposed by Nikhil [1985] and the functional programming language, Haskell, has been developed, and is still being upgraded, by a group of several researchers. The most recent version is defined in the report on Version 1.2 in Hudak et al [1992].

1.5 Chapter Overview

Chapter 2 provides background in four major areas: semantic data modelling with particular reference to Entity-Relationship models [Chen, 1976] and Extended-Entity-Relationship models [Hansen and Hansen, 1992, and Elmasri and Navathe, 1989]; object oriented systems including object models, object-oriented development methodologies and object-oriented database systems; functional systems including functional programming languages and functional databases; and the functional programming language, Haskell [Hudak et al, 1992] with particular reference to the implementation of object-oriented systems.

Chapter 3 introduces a new data model called the Entity-Relationship-Object Model. This model combines the extended ER models of Hansen and Hansen [1992] and Elmasri and Navathe [1989] and the object models of Rumbaugh et al [1991] and Henderson-Sellers [1992]. This combination provides an ER model that incorporates behaviour for both entities and relationships.

Chapter 4 describes the implementation of an ERO model as a simple database system within a functional programming environment. The conceptual framework is described for mapping an ERO model to an integrated implementation system where every entity and relationship is treated as a class. Considerations for mapping this conceptual framework to a functional programming environment using the functional programming language, Haskell are discussed and a specific Haskell prototype is described. A detailed example is shown providing specific program code within an implemented Haskell prototype.

Chapter 5 discusses some of the advantages and limitations associated with the design and implementation methodology together with some areas for future work.

2 Background

2.1 Introduction

This chapter provides background in four major areas. The first area is semantic data modelling with particular reference to Entity-Relationship models [Chen, 1976] and Extended-Entity-Relationship models [Hansen and Hansen, 1992, and Elmasri and Navathe, 1989]. The second area discussed is object oriented systems including object models, object-oriented development methodologies and object-oriented database systems. Thirdly, functional systems are discussed including functional programming languages and functional databases. Finally, there is a discussion of the functional programming language, Haskell [Hudak et al, 1992] with particular reference to the implementation of object-oriented systems.

The marriage of concepts and features from all these areas may be considered to be useful in addressing the problem that modern database systems suffer in that they lack the rich type systems, expressive power and data abstraction that is available in modern programming languages. Conversely, modern programming languages deal inadequately with the persistent (long lived) structured data that is required by modern databases. This causes an unnecessary "semantic mismatch" [Nikhil 1985] between programming languages and databases. This mismatch needs to be overcome so that truly flexible and general database systems can be developed.

2.2 Semantic Data Models

Codd's [1970] relational data model provided a model that is independent of the details of the physical implementation. Since then a number of data models have been developed to extend Codd's original concept so that a model can more faithfully represent the *meaning* of the modelled domain. These models, called "semantic" models, attempt to express meaning in a model by supporting representation of relationships, complex objects, data abstraction and inheritance.

Although there is no concensus on one particular model, Peckham and Maryanski [1988] present a survey of semantic data models whose "one unifying characteristic is that they attempt to provide more semantic content than the relational model.".

Peckham and Maryanski [1988] discuss four main characteristics that are represented in many of the semantic data models.

Generalisation and Specialisation - Generalisations can be formed by considering a set of concepts and identifying common elements that characterise the set. For example, vehicles can be considered to be a generalisation of cars, trucks and motorcycles. Specialisation is the inverse of generalisation and in the above example a car can be considered a specialisation of vehicle. "Generalisation is the means by which differences among similar objects are ignored to form a higher order type in which the similarities are emphasised" [Peckham and Maryanski 1988]. Generalisations can be considered to be "is-a" relationships i.e. a car is-a vehicle. Therefore, generalisation is a form of data abstraction that can be viewed as an inheritance relationship in the object oriented paradigm which provides a direct mapping of generalised relationships into a system's implementation.

Aggregation according to Peckham and Maryanski [1988] "... is the means by which relationships between low-level types can be considered a high level type. The relational data model employs this concept by aggregating attributes to form a relation". Aggregation supports the representation of an abstraction from several smaller and simpler elements. In the simplest form this may correspond to the declaration of the components within a record but can also be seen to incorporate the concept of complex objects. Aggregation can be considered as a "has-a" relationship. For example, a person has-a name and a person has-a bank account.

Classification "... is a form of abstraction in which a collection of objects is considered a higher level object class. Essentially it represents an *is-instance-of* relationship." [Peckham and Maryanski 1988]. Classification differs from specialisation in that classification defines the type of a specific object whereas specialisation provides the derivation or inference of a type from an existing type.

Association indicates that one abstraction serves as a container for instances of other abstractions. It can be considered to represent set membership abstractions. For example, a car-person is-a member-of-the-set of people who drive cars.

Models surveyed by Peckham and Maryanski include SDM [Hammer and McLeod 1981] which organises a collection of entities into classes or types which specify member and class attributes, interclass connections and derivations rather than relationships between classes and the functional data model [Shipman 1981] which limits the constructs to entities and functions providing a direct language for data definition called DAPLEX. Nijssen and Halpin [1989] have proposed a conceptual schema design procedure (CSDP) as part of a methodology called NIAM (Nijssen's Information Analysis Methodology) which is also called fact-oriented modelling. NIAM provides a method for building a system design by starting with specific examples, a set of metaconcepts and a graphical notation and following a well-defined design procedure of nine steps.

In 1978 the ANSI/SPARC committee proposed a three level architecture for database systems consisting of external, conceptual and internal levels. The *external* level corresponds to the user's view of the data in the application domain. The *conceptual* level corresponds to a high level of logical design representing the meanings of entities or objects and the relationships between them. The *internal* level corresponds to a physical or implementable model of the database system. In an object oriented model these three levels represent application relationships between classes, class specification, and class implementations, respectively.

2.2.1 The Entity-Relationship Model

The Entity-Relationship model [Chen, 1976] is a highly abstract semantic model. It is a simple high level design tool that supports external and conceptual modelling by identifying and describing entities that represent the user's view of the data and can also provide mapping from these views to the actual data structures at the internal level. This high level of abstraction and other characteristics that are similar to the characteristics of object-oriented models makes the Entity-Relationship model an ideal basis for the development of a new model that combines the features of semantic and object models.

The Entity-Relationship (ER) Model provides a simple diagrammatic notation for representing the essential details of entities or objects and the relationships between them. Its semantic modelling power comes from its simplicity and generality. There are no limitations placed on the types of entities or relationships that can be represented.

Entities represent "things" identified in the domain that is being modelled. Entities can be physical or conceptual. For example, people, vehicles and books are physical entities and transactions, skills and bank accounts are conceptual entities. A set of entities with the same characteristics is called an entity set or *entity class*. A *relationship* is a linking between two entities. For example, if an employee works in a particular department, *employee* and *department* can be considered entities and *works in* is a relationship linking the two entities. An *attribute* is a named characteristic of an entity. The cardinality of a relationship between entity X and entity Y may be considered as the number of X's associated with a single Y and the number of Y's associated with each X. Cardinality can represent zero-to-many, one-to-many or many-to-many associations

2.2.2 Enhanced Entity Relationship Models

There have been several proposals for enhanced or extended entity-relationship models that incorporate additional semantic modelling concepts such as generalisation, classification and aggregation. Elmasri and Navathe [1989] propose an enhanced-ER or EER model based on an earlier Entity-Category-Relationship (ECR) model [Elmasri et al 1985]. The EER model incorporates the concepts of subclasses, superclasses, specialisation, generalisation, categories and attribute inheritance.

Hansen and Hansen [1992] propose a model that incorporates standard ER notation (which models 'objects' rather than 'entities') plus specialisation and 'aggregate object sets'. There is no attempt to model behavior. Hansen and Hansen define an **object set** as "... a set of things of the same kind" and an **object instance** as "... a particular member of an object set".

Object sets are represented as rectangles with the object set name inside it in upper case and an object instance can be represented as a point within the rectangle with its name in lower case. **Attributes** are represented as ellipses coming off a class rectangle. (see Figure 2.1)

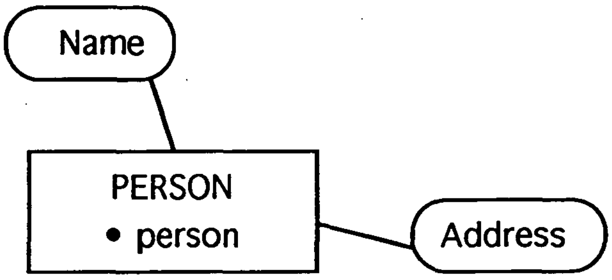


Figure 2.1
The class PERSON contains an object called person. All objects of class person have the attributes Name and Address.

Specialisation-Generalisation or **inheritance** is represented as a U symbol (the same symbol is used by Elmasri and Navathe [1989]) where the open part of the U points to the super class (see Figure 2.2).

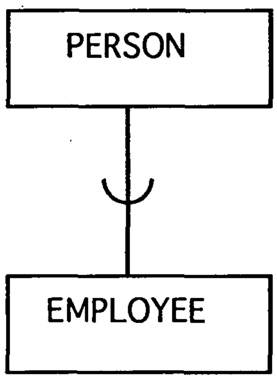
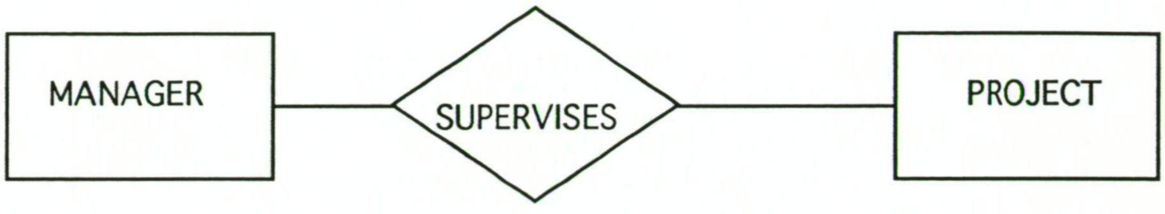
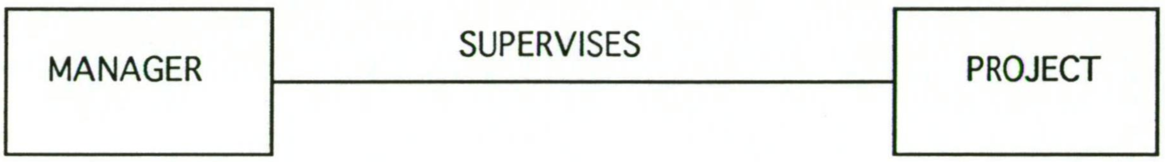


Figure 2.2
An EMPLOYEE is a specialisation of PERSON.

Hansen and Hansen [1992] view relationships that have their own attributes as special object sets and define them as *aggregate object sets*. A simple **relationship** between two classes is represented as a named arc between the two classes with an optional embedded diamond (see Figure 2.3). Aggregate object sets are represented by enclosing the participating classes in a larger rectangle and the end points of arcs to attributes indicate the semantics of attributes within the relationships. This leads to the need for "colouring" the ER diagram (see Figure 2.4)



(a)



(b)

Figure 2.3 (a) and (b) both represent the same relationship SUPERVISES

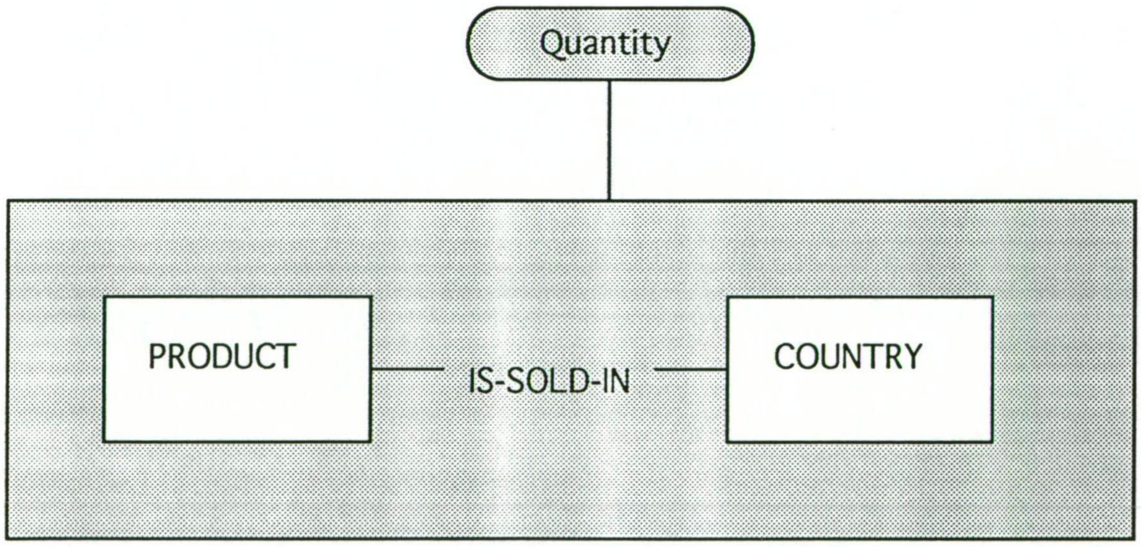


Figure 2.4 (from Hansen and Hansen, 1992, p140)
The QUANTITY attribute depends on both PRODUCT and COUNTRY and is therefore an attribute of the relationship between PRODUCT and COUNTRY

Cardinalities or the number of instances of one object that correspond to the instances of another object in the relationship are represented using

"1" and "*" (indicating 'many') at the end of a relationship arc (see Figure 2.5)

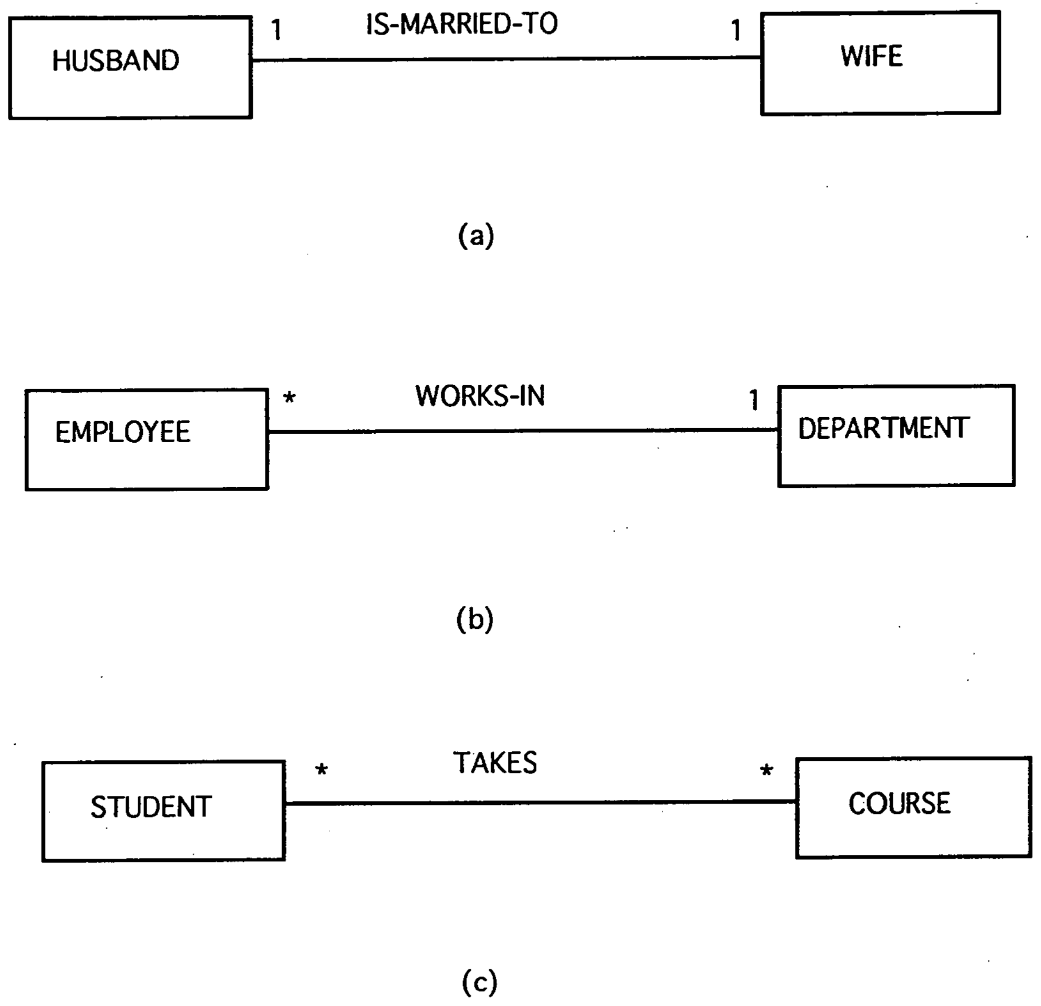


Figure 2.5
(a) A one-to-one relationship - a wife has one husband; a husband has one wife
(b) An employee is in one department; a department has many employees
(c) A student takes many courses; a course has many students

Figure 2.6 is an example of an EER model for a data model of an Invoice for Manwaring Consulting Services taken from Hansen and Hansen [1992] p 152. This model contains two relationships represented as aggregate object sets: ENGAGED-IN and ON which is a relationship between the entity PROJECT and the relationship ENGAGED-IN. The use of aggregation in EER models provides extra semantic expressiveness by allowing relationships between relationships.

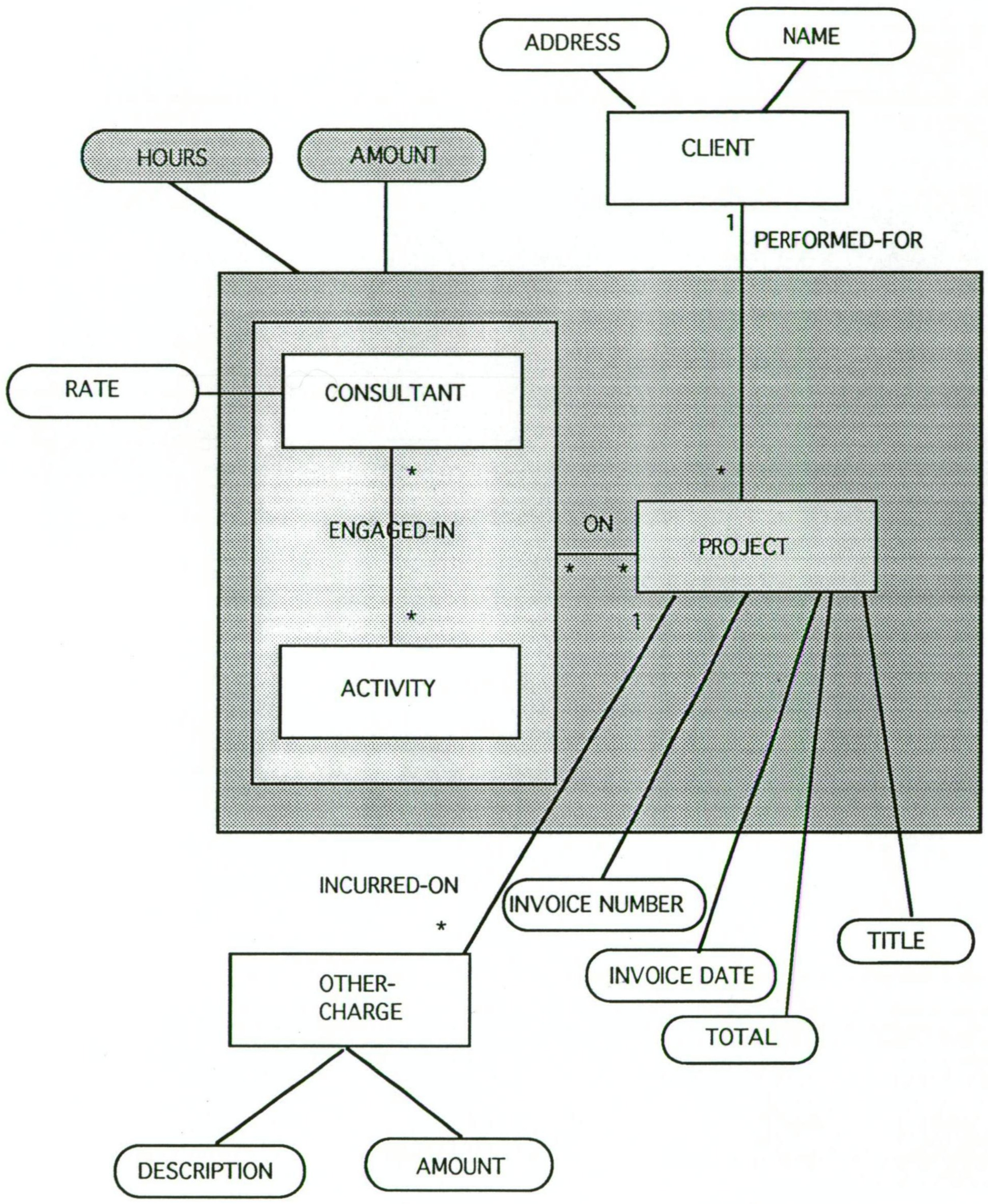


Figure 2.6

Data Model for an Invoice for Manwaring Consulting Services, from Hansen and Hansen, 1992.

It should be noted that there are a couple of semantic problems with the model in Figure 2.6. A CONSULTANT should have a Name attribute and an ACTIVITY should also have a Name attribute. Otherwise, the model is suitable for modelling the entities and relationship for an invoice.

2.3 Object Oriented Systems

Although there seems to be no agreement as to exactly what constitutes an object-oriented system, a useful definition to adopt is Wegner [1987] in which a system which supports objects, classes and inheritance may be considered to be object-oriented. Object-oriented systems are based on decomposing problems to a set of objects in the problem domain. Classes encompass the concepts of data abstraction and polymorphism in that they provide a way of classifying objects in terms of behavioural and data abstraction. Inheritance allows the use of existing definitions to be the basis of new definitions.

Therefore, data abstraction rather than procedural abstraction is the core concept in object-oriented software development. Instead of data being passed from one procedure to another, as in the procedural paradigm, flow of control is passed from one data abstraction to another.

The development of object-oriented systems is not just a modelling technique and/or programming style but is based on a software development paradigm that incorporates object-oriented analysis (OOA), object-oriented design (OOD) and object-oriented programming (OOP). Traditional software development life cycles tend to be linear and sequential, or top down, as in the waterfall model [Henderson-Sellers, 1992]. Some models provide for feedback from various stages of analysis, design and implementation as in the spiral model [Boehm, 1988] and the fountain model [Henderson-Sellers, 1992]. The object-oriented life cycle is not considered to be linear. Some analysis may be carried out before the design begins but parts of the analysis may proceed in parallel with the design and implementation of other parts of the system. Throughout the development cycle the concept of an object remains the same. Objects are identified, then modelled as classes and then refined and, if possible, reused in implementation.

Henderson-Sellers [1992] provides a number of alternative methodologies for a complete object-oriented life cycle which includes analysis, design and implementation. The choice of methodology depends on the application environment. Methodologies proposed by Henderson-Sellers include: Object-oriented analysis, Object-oriented design and Object-oriented implementation (OOO), Functional analysis, Object-oriented design and Object-oriented implementation (FOO) and Object-oriented analysis, Object-oriented design and Functional implementation (OOF).

2.3.1 Object Models

Rumbaugh et al [1991] propose the Object Modelling Technique (OMT) which models information in three different forms: the Object Model, Functional Models and Dynamic Models but there is no linking between these models. Rumbaugh et al [1991] regard OMT as an "enhanced form of ER" although it is not presented as a object oriented database development tool but as a tool to assist in the design of relational databases.

The main characteristics of the OMT model are as follows:

An *object* is a distinct concept, abstraction or thing which has its own unique identity and can be used to model things in the real world and "provide a basis for computer implementation" [p 21].

An *object class* (or *class*) "describes a group of objects with similar properties (attributes), common behaviour (operations), common relationships to other objects, and common semantics" [p 22]. For example, Mary Smith is an object of class person.

An *object diagram* provides "... a formal graphic notation for modelling objects, classes, and their relationships to one another" [p 23].

An *attribute* is a data value or property of every object in a class. For example, the object Mary Smith may have an attribute containing her address.

An *operation* is "... a function or transformation that may be applied to or by objects in a class." [p 25]. A *polymorphic operation* is an operation that may apply to many different classes. A *method* is the implementation of an operation for a class. For example, there may be an operation for changing a person's address.

A *link* provides a physical connection between objects and an *association* describes a group of links with common structure and semantics. For example, an employee may work in a department. "Work in" is an association between employees and departments.

Multiplicity "...specifies how many instances of one class may relate to a single instance of an associated class" [p 30]. For example, many employees may work in one department.

Generalisation is a relationship between a general class, called the *superclass* and more refined versions of it, called *subclasses.* Each subclass is said to *inherit* the features (attributes and operations) of its superclass. For example, employee may be a subclass of person.

An *aggregation* is a strong association where an aggregate object comprises components which are objects. Rumbaugh et al [1991] describe this more abstractly as "[t]he aggregate is semantically an extended object that is treated as a unit in many operations, although physically it is made of several lesser objects" [p 57].

Apart from the concept of modelling behaviour, the OMT concepts of objects, classes, object diagrams, associations, multiplicity, generalisation and aggregation can be considered to be approximately equivalent to the EER concepts of entities, entity classes, ER diagrams, relationships, cardinality, generalisation and aggregation respectively.

Figure 2.7 shows two classes with an association between them in the OMT model and the ER model structural equivalent.

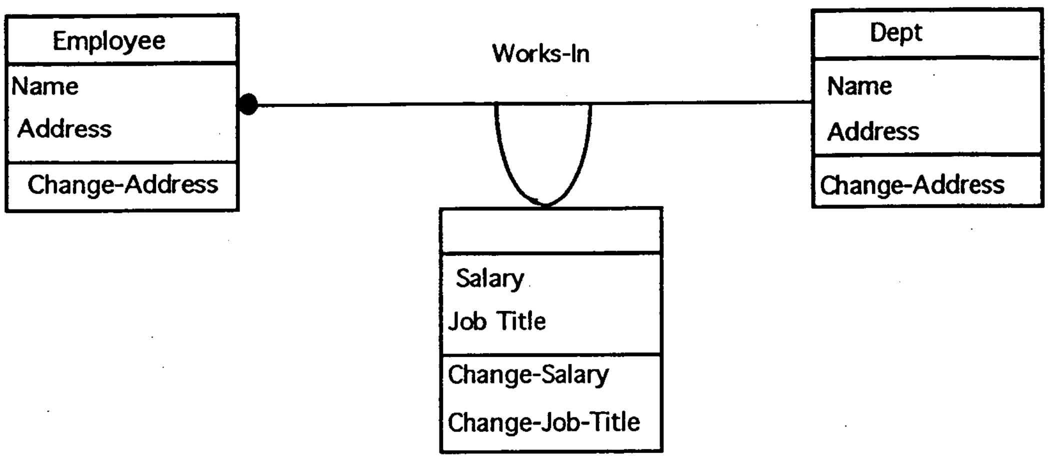


Figure 2.7(a)

An OMT model for Employees who works in Departments

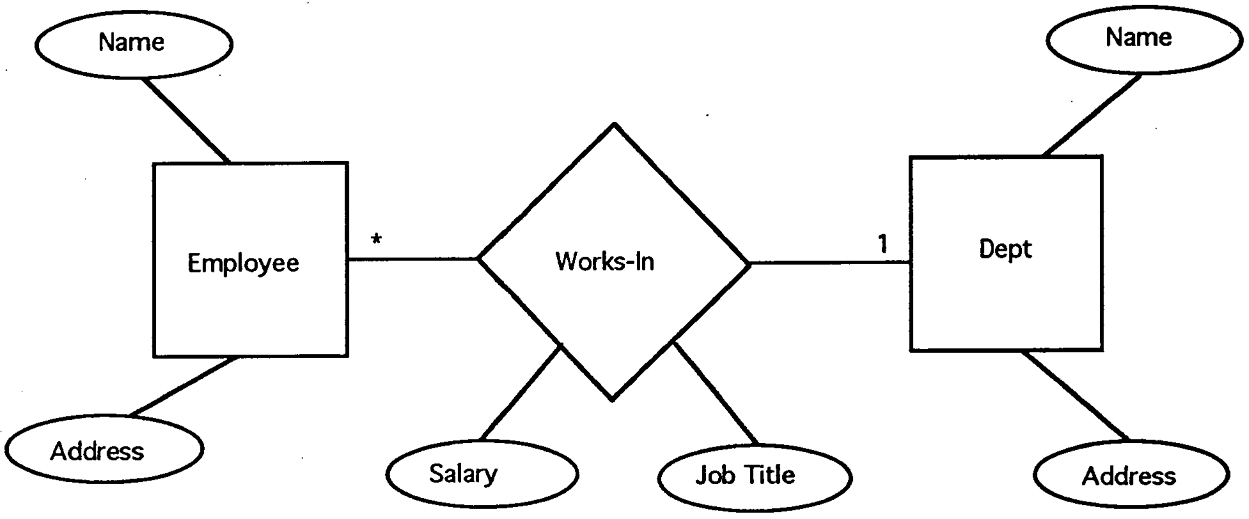


Figure 2.7 (b)

A structurally equivalent EER model for Employees working in Departments

Beck and Cunningham [1989] suggest a technique based on CRC cards. CRC stands for Class, Responsibility and Collaborators. The technique was developed for use in team sessions where entities can be identified and behaviours can be specified for interfaces. An entity is given a descriptive name and the responsibilities or behaviours are then listed under this name. Other entities that will know about this entity or with which this entity will interact, are listed as collaborators. This technique is useful for initial or high level design where a group can brainstorm about the overall structure of a system.

Other object-oriented models include those of Shlaer and Mellor [1989] and Booch [1986]. These models are more complex. Booch diagrams require the user to learn a new syntax as specific as any programming language. Shlaer and Mellors's technique of capturing all information about the states of attributes and the transitions between states leads to extremely complex diagrams that may be difficult to read.

2.3.2 Object Oriented Databases

From the early 1970s until recently relational database systems have been the standard model for many commercial database systems. The main limitation of relational systems is that they only model data and the relationships between data not the behaviour of entities or objects in the model. Currently researchers are looking to object oriented data models as the next generation of database systems. [Atkinson et al 1990]. Hansen and Hansen [1992] suggest that "object-oriented databases are the result of the convergence of two research disciplines: semantic data modelling and object-oriented languages. These disciplines developed independently but in recent years have begun to merge with important implications for database processing".

Another limitation of relational database systems is that there is a "... need to translate data from a 'real world' abstract model to an 'implementable' physical model" [Dobbie 1991]. This translation can take many forms and tends to become increasingly customised during development making modification and maintenance difficult while maintaining semantic integrity of the original model.

Database implementers need to perform more complex operations on stored data than retrieving it and sharing it so that more complex software systems can be built around database systems. This has been recognised for some time in software engineering and the building of large software systems. The core or base of most large systems is a database. Built around this database are usually a number of application programs written in some host language. The database language, which usually allows some form of navigation around the database and explicit adding and deleting of records, is incompatible with the application programming language which must have a specific interface set up so that database records can be explicitly transformed into the data structures of that language. The customised nature of the interface and these programs makes the whole system difficult to modify and maintain particularly in terms of data integrity.

The object oriented paradigm for large software systems and now specifically databases, allows the high level design to flow through the development cycle without changing the basic elements. An object exists as a conceptual entity from the abstract model through to the actual implementation.

Although Codd [1970] gave a clear specification of relational systems and most researchers agree that there is no clear specification for object oriented database systems [Dobbie 1991, Atkinson et al 1990] there is some agreement on the characteristics of object oriented database systems.

The final report of the Object-Oriented Database Task Group (OODBTG) organised by the ASC X3/SPARC Database Systems Study Group [Fong et al 1991] provides broad recommendations in the areas of Object Information Management (OIM) which covers the general area of object management in programming languages, networks, design methodologies, user interfaces etc and Object Data Management (ODM) which covers object models and database systems. Because of the breadth of this report the task group does not propose any new concepts or specific models but the part of the report most relevant to this project is that the report has marshalled existing generally accepted ideas into a reference model for ODM. The characteristics of the OODBTG reference model are similar to the necessary characteristics proposed by a number of researchers.

Atkinson et al [1990] differentiate between mandatory, optional and open characteristics as summarised below.

Mandatory	**Optional**	**Open**
complex objects	multiple inheritance	type systems
object identity	type checking	programming paradigms
encapsulation	type inference	ways of representing objects
types or classes	distribution	
class hierarchies	design transactions	
overloading	versions	
overriding		
late binding		
computational completeness		
persistence		
secondary storage management		
concurrency		
recovery		
ad hoc querying		

Dobbie [1991] differentiates between object oriented features and database features as summarised below.

Object-Oriented Features	**Database Features**
complex objects	persistence
object identifiers	secondary storage management
encapsulation	concurrent users
inheritance	authorisation mechanisms
overloading	recovery procedures
overriding	efficient access methods
late binding	schema modification

The main examples given by Dobbie [1991] of currently available object-oriented database management systems that provide complex objects, object identity, encapsulation, types or classes inheritance, overloading, overriding and late binding, extensibility, persistence and secondary storage management, concurrency, recovery, ad hoc querying and schema modification include O2 [Deux et al, 1990], Iris [Wilkinson et al, 1990], ONTOS [1989] and Gemstone [Bretl et al, 1990].

2.4 Functional Systems

2.4.1 Functional Programming Languages

Functional languages are programming languages in which computations are carried out entirely by the evaluation of expressions (functions) to produce values. Functions and values are treated as first class entities and functions can be recursive, higher order and polymorphic. Functional languages have no side effects or assignment statements. That is, whereas imperative languages '...are characterised as having an implicit *state* that is modified (i.e. side effected) by *constructs* (i.e. commands) in the source language.', declarative or applicative languages '...are characterised as having *no* implicit state, ...[and] ... state-oriented computations are accomplished by carrying the state around explicitly rather than implicitly, and looping is accomplished via recursion rather than by sequencing' [Hudak, 1989].

Nikhil [1985] advocates using functional programming languages for developing functional databases because of their expressiveness, use of lazy evaluation, rich type systems and opportunities for optimisation using program transformation and parallelism.

Rich typing and polymorphism

Every value in a functional program has an associated type. Values are "first class" in that they may be passed as arguments to functions, returned as results and placed in data structures. Types in a functional language are not first class and most functional languages provide a static type system where the types of all values can be inferred by the type system at compile time and checked for errors. Most functional languages provide built in types for characters, integers, booleans etc but these are semantically no different to user defined types. Polymorphic types are types that are universally quantified over all types. For example, [*] (in Miranda) or [a] (in Haskell) defines a list of "things" where things can be any valid type such as integers, characters, strings or even complex types such as lists.

This is the first step towards data abstraction in that a set of functions can be defined to operate over a collection of things whatever type those things may be.

Lazy evaluation

Lazy evaluation in functional programming languages is based on the characteristic that functions need not be evaluated strictly, ie. each value, if computed is computed only once and no value is computed unless it is needed. The non strictness of functions means that a function that would have been non terminating if it was strict can return a result if evaluated lazily because it never attempts to evaluate its non terminating argument. Hudak [1989] argues that lazy evaluation provides expressiveness in a language as follows, "First, lazy evaluation frees a programmer from concerns about evaluation order ... [and second provides] ... the ability to compute with unbounded 'infinite' data structures".

Data Abstraction

Functional languages provide data abstraction as either algebraic data types (user defined types in other languages) and abstract data types (ADTs) which are implemented using the keywords "abstype ... with" in Miranda and using modules in Haskell. An abstract data type defines a collection of data and the valid set of operations on that collection where the actual representation of the data structures and functions is hidden from other modules using instances of that ADT.

2.4.2 Functional Databases

Nikhil [1985] suggests a new way of looking at database systems using a functional approach. In this new view, *functional* databases are databases that are never updated but may be seen as a potentially infinite stream of versions of the database. This is because the functional programming approach has no side effects (i.e. there is no assignment statement and therefore no values are ever actually changed). All values are created by the application of functions to an existing value, however complex.

Types

Current databases are characterised by the inadequacies of their type structures. Most databases support little more than scalar types and sets of records. This limited typing can prove inadequate for many applications involving large amounts of data. Nikhil suggests that rather than "[u]sing a database sublanguage embedded in an existing programming language [e.g. EQUEL in C, SQL in PL/1] due to the semantic mismatch between the two" that "[w]e would like to explore the opposite evolutionary path: that of beginning with a programming language, attaching great importance to the expressive power and semantic elegance of the language while gradually incorporating features normally found in databases".

Nikhil [1985] suggests that the type theory of a programming language determines the domain of definable types for that programming language in much the same way that a data model determines the domain of definable types for a database. Therefore a database itself could be considered the equivalent of a data structure in a programming language. The natural way to retrieve information from or change a data structure is to apply an operation or a function to it. If databases are considered in this way then the natural way to query (retrieve information from) a data base or update a database is to apply a function to the database.

Data Abstraction

Traditional data manipulation languages for databases have no facilities for data abstraction. Most database operations are hard-coded and application specific. If the long held tenet of "data independence" is to be achieved in modern databases then data abstraction must be naturally incorporated into database programming languages. Data independence in the form of abstract data types is a feature of most high level programming languages. That is, an abstract data type may only be manipulated using the functions associated with that type and because these functions are only associated with a specific abstract data type then the functions can be changed without affecting other applications that use that abstract data type.

Updates and Queries

Queries and updates in current database systems are either done using a specific query/update language (e.g. SQL, QUEL) which are incompatible with programming languages, or using one of these database sublanguages embedded in some programming language.

A functional database approach would treat queries and updates in a different way. A query could be considered to be an expression to be evaluated within the database domain. The answer to the query is the value of the expression. An update could be considered to be the application of a function to the current database and the result returned would be a new version of the database.

2.5 The Functional Programming Language Haskell

The language chosen as the basis for this project is the functional programming language Haskell. Haskell is a general purpose, purely functional programming language incorporating many of the features of other functional programming languages including higher order functions, lazy evaluation, static polymorphic typing, user-defined data types, pattern matching and list comprehensions. In addition it also includes a module system, a purely functional I/O system and a rich set of primitive datatypes, including lists, arrays, arbitrary and fixed precision integers and floating point numbers. The latest version of Haskell was released by Yale University in March 1992. The Haskell Committee (the group responsible for the design of Haskell) considers that "[Haskell] ...should be suitable for teaching, research, and applications, including building large systems." [Hudak et al, 1992]

Apart from lazy evaluation, strong typing and polymorphism, the features of particular interest to this project are type classes, modules, abstract data types, preludes and continuation style functional I/O.

Type Classes

One of the main additions to the type system in Haskell that is not in other functional languages is type classes. Originally called "classes" but in the new report "type classes", type classes were originally introduced to provide a clean, flexible method of dealing with overloading of arithmetic operators. Classes in Haskell serve a different purpose to classes in object oriented programming languages. They provide definitions of overloaded operations associated with a class or "...*ad hoc polymorphism,* better known as *overloading*" [Hudak and Fasel, 1992]. For example,

```
class Eq a where
      (==) :: a -> a -> Bool
```

defines a class called **Eq** with a single operation == (equality) in the class. "**Eq a** is not a type expression, but rather it expresses a constraint on a type ... [the] declaration may be read ' a type **a** is an instance of the class **Eq** if there is an (overloaded) operation ==, of the appropriate type, defined on it'" [Hudak and Fasel 1992]. Thus Haskell allows other functions to be defined within the *context* of **Eq**. For example,

```
elem :: (Eq a) => a -> [a] -> Bool
```

"... expresses the fact that **elem** is not defined on *all* types, just those for which we know how to compare its elements for equality" [Hudak and Fasel, 1992]

Although Haskell classes offer some measure of abstraction and inheritance, since they are mainly used for overloading existing operators and functions where the emphasis is on the functions not data, a better abstraction for object-oriented classes is the ADT using modules. Indeed, Hudak and Fasel [1992] state "... modules provide the only way to build abstract data types (ADTs) in Haskell.".

Modules and Abstract Data Types

Modules are the basic building block of all Haskell programs. Yale Haskell allows modules to be separately compiled increasing efficiency and allowing truly modular program development. "A module defines a collection of values, datatypes, type synonyms, classes etc.... and *exports* some of these resources, making them available to other modules" [Hudak et al 1992]. Encapsulation and information hiding can be achieved in the development of ADTs using modules. Modules can also be used to implement inheritance (albeit a little crudely) since any resources that should be inherited by other modules can (and must) be exported explicitly. Modules are also important for continuation style I/O (see below) in that "A Haskell *program* is a collection of modules, one of which, by convention, must be called `Main` and must export the value `main`. The *value* of the program is the value of the identifier `main` in module `Main`, and `main` must have type `Dialogue`." [Hudak et al, 1992].

Preludes

Some special modules in Haskell are known as "preludes". Preludes can be considered to be library modules containing logically associated values and datatypes. Two special preludes, together called the "standard prelude", PreludeCore and Prelude are part of the Haskell language definition and are imported automatically into all Haskell programs. User defined preludes are also permitted.

Purely Functional I/O

The Haskell I/O system is purely functional and is based on lazy evaluation and higher order functions. Haskell I/O provides two models: stream based I/O and continuation based I/O. Hudak [1989] states that these "... two seemingly very different solutions ...turn out to be exactly equivalent in terms of expressiveness". The Haskell Report actually defines continuation based I/O in terms of stream based I/O. A Haskell program communicates with the operating system via a stream of messages or a lazy list and the program maps a stream of responses to a stream of requests. The use of lazy evaluation means that a program need not look at the responses before it issues a request. So a Haskell program using I/O has the type:

```
    type Dialogue = [Response] -> [Request]
```

Continuation-based I/O is the preferred methodology [Hudak and Fasel, 1992] for writing interactive Haskell programs. This programming method is called Continuous Passing Style (CPS). The concept of a *continuation* is fundamental to this method and is best explained by Hudak and Fasel [1992] as follows:

> "A continuation is basically a (possibly nullary) function that maps an 'intermediate value' (possibly empty) to 'the rest of the program.' In this way, continuations are used to explicitly manage 'flow of control', and thus we tend to define functions that, instead of returning with an answer, will apply a continuation (passed in as an argument) to the answer."

A Haskell program can be thought of as communicating with the operating system using continuations. To set up this communication the operating system expects a top-level identifier called "main" whose type is Dialogue. The flow of control in the program is maintained by cascading and/or recursive calls to functions of type Dialogue. That is, every function that interacts with user input must be of type Dialogue.

Developing Continuation-based I/O Implementations

The Haskell Report [Hudak et al, 1992] defines a program in the continuation-based I/O model as "...a collection of *transactions*...[which capture]...the effect of each request/response pair" as defined in type Dialogue above. Transactions are built-in system functions. For the purpose of developing some examples we will deal with a subset of the available transaction functions defined as follows:

```
data Request =
     -- file system requests:  (file I/O)
            readFile     String
          | writeFile    String String

     -- channel system requests (standard I/O)
          | readChan     String
          | appendChan   String String
```

All these requests work with strings.

```
     data Response = Success
                   | Str     String
                   | StrList [String]
                   | Failure IOError
```

"The response to a request is either Success, when no value is returned; Str *s*, when a string value *s* is returned; or Failure *e*, indicating failure with I/O error *e*" [Hudak et al, 1992]. That is, a response can either be a success continuation (SuccCont), a string continuation (StrCont) or a failure continuation (FailCont) defined by the following synonyms:

```
     type SuccCont   =    Dialogue
     type FailCont   =    IOError -> Dialogue
     type StrCont    =    String -> Dialogue
```

Note that all these possible returned values are of type Dialogue and so can be replaced by other functions of type Dialogue which in turn have continuations and so on. This provides the infrastructure for the cascading calls outlined above which are the core of CPS programming.

In order to construct some small examples we will use the following transaction functions (with their signatures):

```
done::Dialogue
appendChan::String->String->FailCont->SuccCont->Dialogue
readChan::String->FailCont->SuccCont->Dialogue
readFile::String->FailCont->StrCont->Dialogue
```

and, by definition:

```
stdin = "stdin"
stdout = "stdout"
stderr = "stderr"
```

The function "appendChan" provides for writing a string to standard output; "readChan" allows for reading a single (infinite) string from standard input; and "readFile" allows for reading a named file as a single string.

Continuation I/O is lazy and requires synchronisation. Synchronisation is achieved by passing results back through anonymous, or lambda, variables which can be used as input to other functions.

Examples

Since input is read lazily as an infinite string, a useful (supplied in PreludeList) function is *lines* which breaks a string up into a list of strings at newline characters.

A simple program that writes the string "Hello world" to standard output is:

```
module PrintString where

main :: Dialogue

main = appendChan stdout "Hello world\n" abort done
```

The failure continuation is "abort" or abnormal termination and the success continuation is "done" or normal termination.

A program which prompts for and reads a string from standard input and then prints "Hello <string>" is shown below. It replaces the success continuation from the first example with another function.

```
module ReadAndPrintString where

main :: Dialogue

main = appendChan stdout "Enter your name" abort (
          readChan stdin abort (\userinput ->
          response (lines userinput)))

response (line:_) =
          appendChan stdout ("Hello" ++ line) abort done
```

The string continuation for the "readChan" function reads whatever is typed at the keyboard into the lambda variable "userinput" which is then used in the function "response".

A program that prompts for a file name and the reads the contents of the file is shown below.

```
module ReadAFile where

main :: Dialogue

main = appendChan stdout "Enter file name" abort (
        readChan stdin abort (\userinput ->
        response (lines userinput)))

response (line:_) = readFile line
     (\ioerror ->
      appendChan stdout ("cannot open" ++ line) abort done)
     (\contents ->
      appendChan stdout contents abort done)
```

Finally, a skeleton of a program that sets up an interactive menu is shown below.

```
module MenuDemo where
import  ....

main :: Dialogue
main = appendChan stdout "Welcome\n" abort (
        readChan stdin abort (\userinput ->
        processMenu (lines userinput)))

menu =     "1. Do This \n" ++
           "2. Do That \n" ++
           "3. Do The Other\n\n" ++
           "Select option 1, 2 or 3: "
```

```
processMenu :: [String] -> Dialogue
processMenu inp =
     appendChan stdout menu abort
          (case inp of (line:rest) ->
          case line of
               "1" -> DoThis ....
               "2" -> DoThat ...
               "3" -> DoTheOther ...
               _   -> appendChan stdout
                         "error - try again\n" abort
                         (processMenu rest))

DoThis ...
DoThat ...
DoTheOther ...
```

A further discussion of this style of programming is in Hudak and Fasel [1992].

3 Data Model Design

3.1 Introduction

This chapter introduces a data model called the Entity-Relationship-Object Model. This model combines the extended ER models of Hansen and Hansen [1992] and Elmasri and Navathe [1989] and the object models of Rumbaugh et al [1991] and Henderson-Sellers [1992]. This combination provides an ER model that incorporates behaviour for both entities and relationships. Entities and relationships are treated as classes which inherit methods for addition, deletion and retrieval from the generic superclass. Instances of each of these classes are known as entity-objects (E/Os) and relationship-objects (R/Os) respectively. The only difference between them is that a relationship-object provides a link between two objects (E/O or R/O) participating in that relationship.

3.2 Analysis, Design and Implementation Methodology

Object-oriented methodologies are relevant to this design and implementation since they provide a high level of continuity due to objects being the same from one phase to the next. The development methodology used here is based on Henderson-Sellers [1992] O-O-F (Object-oriented analysis, Object-oriented design, Functional implementation) methodology which provides a good framework for the design and implementation of an object database in a functional programming language. The term "functional implementation" as used by Henderson-Sellers [1992] means implementation in one of the available procedural languages such as Cobol, Ada, Pascal etc. For this project the O-O-F methodology is modified to incorporate the ERO model in design and to provide for implementation in the functional language Haskell.

Henderson-Sellers' [1992] O-O-F methodology has seven steps:

1. Object-oriented decomposition in analysis
2. Analysis/identification of objects
3. Identify object interactions
4. Analysis merges to design. Provide a detailed model.
5. Consideration of library classes

6. Reevaluate set of objects with respect to constraints of procedural language
7. Code objects into procedural language.

The first three steps in this methodology involve Object Oriented Analysis, an area not addressed in this thesis. It is assumed that the designer/implementer is sufficiently experienced to be able to identify appropriate objects, classes, relationships, associations and generalisations.

Steps 4 to 7 represent the design phase which may be considered to be specific to a particular application area. This thesis develops specific methodologies for the implementation of an object database within a functional language environment and replaces steps 4 to 7 with:

4. Analysis merges to design. Produce an ERO model.

5. In a functional language provide a library that implements a polymorphic superclass and its default methods within a suitable interactive interface.

6. Map the classes directly from the ERO model to the functional programming language as abstract data types which inherit the generic superclass.

7. Link all the components of the system that have been provided in steps 5 and 6 into an integrated database system.

The consequence of this modified methodology is that it provides a single, integrated methodology incorporating the ERO model and maintains the continuity from design to implementation by mapping an ERO model directly into Haskell. An implementation methodology for the new steps 5, 6 and 7 is presented in detail in chapter 4.

3.3 An Entity-Relationship-Object Model

The Entity-Relationship-Object Model (ERO model) is a semantic data model designed to provide the highest level of abstraction possible while maintaining semantic integrity. It is based on a combination of the ER model and the OMT model.

The ER model [Chen, 1976] can be considered to be one of the most useful of the current semantic data models. Firstly, it provides support at both the external and conceptual modelling levels by identifying and describing entities that represent the user's view of data in some application domain. Secondly, the ER model can provide the mapping of this view to actual data structures at the internal modelling level.

The OMT model [Rumbaugh et al, 1991] has many of the same characteristics as the ER model, together with the facility for modelling behaviour as operations that may be applied to, or by, objects in a class.

If we take the level of abstraction in these models a step further, we can consider *all* components in the model, both entities/objects *and* relationships/associations, as classes. The OMT model allows for this in special cases but not all cases.

The ERO model has three major characteristics. Firstly, it represents both entities and relationships as *classes of the same polymorphic type.* The only difference between entity instances, called entity-objects or E/Os and relationship instances, called relationship-objects or R/Os, is that the unique identifier (uid) for an E/O is system generated and the uid for an R/O is a composite uid made up of the uids of the E/Os participating in the relationship. All relationships in the ERO model are binary relationships although n-ary relationships can be modelled as binary relationships as discussed in 3.3.3. If the same two E/Os participate in two different relationships the composite uids in each of the relationships can be distinguished as unique by virtue of the specific R/O to which they belong.

Secondly, as with all object models, uids are system generated and are therefore not explicit in the model. The polymorphic nature of entity and relationship classes ensures that a class has only one uid and that the composite uids necessary in relationships can only be constructed from other existing uids. This is discussed in the example in 3.3.5 below.

Thirdly, the ERO model has been developed specifically for the functional implementation methodology set out in 3.2, above, so operations, or methods implemented as functions, cannot have side-effects.

The Entity-Relationship-Object (ERO) model incorporates notations from both object oriented models and Extended-Entity-Relationship (EER) models. It provides icons based on object oriented models for representing objects and/or classes (or O/Cs as Henderson-Sellers [1992] calls them). EER notation is used for representing generalisation/specialisation, cardinalities of relationships and aggregations.

All objects in the ERO model inherit the default methods for add, delete and retrieve from the polymorphic superclass. Consequently it is not necessary to represent these methods explicitly in the model.

3.3.1 Entity-Objects and Entity-Object Classes

The icons for entity-objects (E/Os) are based on Rumbaugh et al [1991] and Henderson-Sellers [1992]. E/O classes are represented as rounded rectangles with the class name inside it in upper case and attributes listed under a single line within the rectangle (see Figure 3.1). Any methods associated with the class are listed under the attributes below a double line.

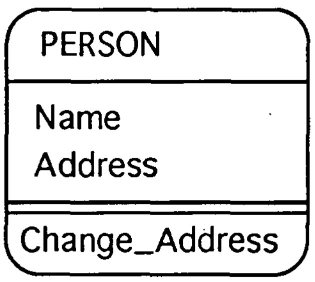


Figure 3.1
All entity-objects of class PERSON have the attributes Name and Address and the method Change_Address.

3.3.2 Generalisation

Generalisation or inheritance is based on EER notations and is represented using a **U** symbol where the open part of the **U** points to the super class (see Figure 3.2). When referring to inheritance the generalised class will be called a **superclass** and the specialised class will be called a **subclass**.

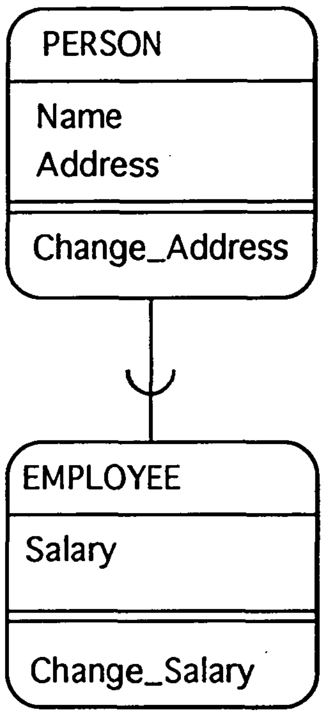


Figure 3.2
PERSON is the superclass and EMPLOYEE is the subclass which inherits the attributes Name and Address and the method Change_Address.

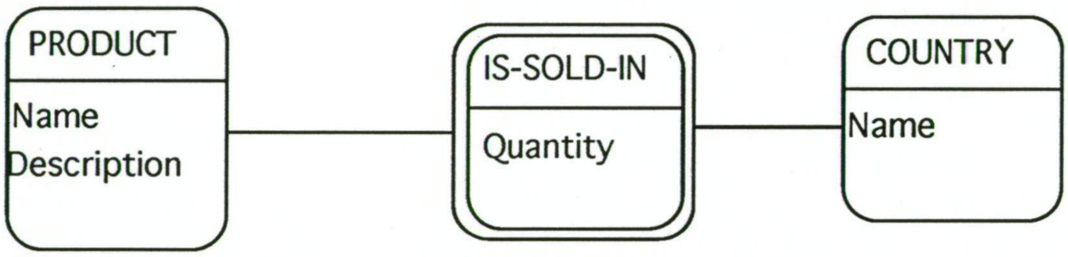
3.3.3 Relationships

Although all relationships in the ERO model are binary relationships, n-ary relationships can be modelled as combinations of binary relationships as discussed below. A relationship-object in the ERO model is like a *link* in the OMT model in that it is the "...physical or conceptual connection between object instances." [Rumbaugh et al, 1991]. The difference is that in the ERO model the objects instances can be other relationship-objects not just entity-objects. This leads to the concept of two types of relationships in the ERO model.

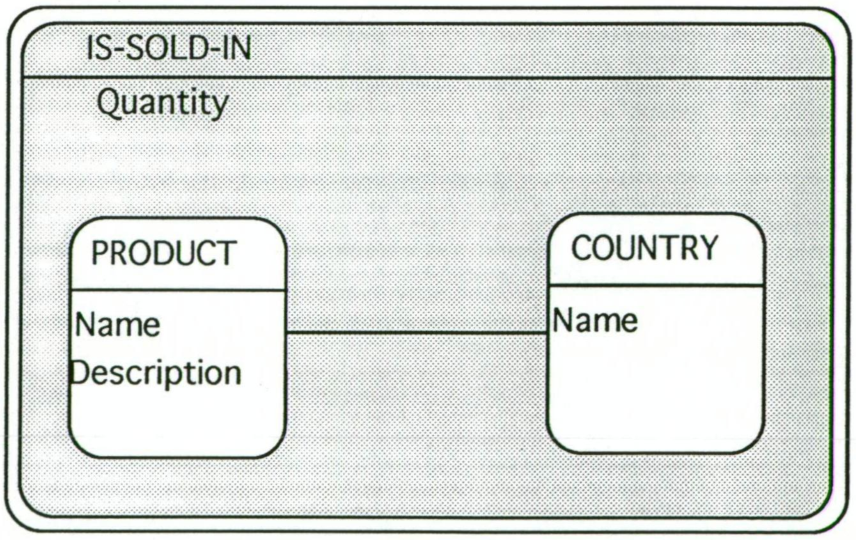
Simple relationships in the ERO model are relationships that do not participate in any other relationship and can be represented in a similar way as in ER and OMT models.

Complex relationships, involve relationships between relationships as well as entities and should be represented in a closed, coloured notation similar to Hansen and Hansen's [1991] extended ER models.

In ERO notation a **relationship** between two classes is represented using a class icon differentiated for semantic purposes using a double border indicating that this class has a composite uid. With this notation simple relationships can be represented linearly as in Figure 3.3 (a) and complex relationships can be represented as in Figure 3.4(b). The rule is that if a relationship participates in a relationship with another relationship then it must be represented as a complex relationship. This means that a relationship may be represented in different ways depending on the semantic context.



(a)



(b)

Figure 3.4 (a) and (b)
The QUANTITY attribute depends on both PRODUCT and COUNTRY and is therefore an attribute of the relationship between PRODUCT and COUNTRY

The ERO model only provides for binary relationships. However, higher order, or n-ary, relationships can be represented as nested binary relationships. Rumbaugh et al [1991] state that "Associations may be binary ternary, or higher order....We have encountered a few general ternary and few, if any, of order four or more. Higher order associations ... should be avoided if possible." An OMT example of a ternary relationship is given in Rumbaugh et al [1991], page 28-29 which is said to be "... an atomic unit and cannot be subdivided into binary associations without losing information...[since]...a programmer may know a language and work on a project, but might not use the language on the project". This example is reproduced in Figure 3.5

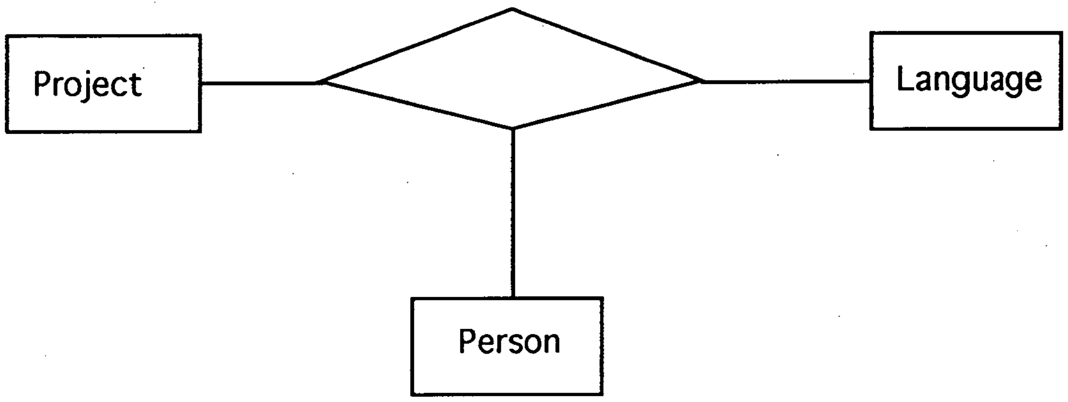


Figure 3.5 OMT Example of a ternary relationship [Rumbaugh et al, 1991]

The diamond in the OMT example is the OMT symbol for general ternary and n-ary associations. Rumbaugh et al [1991] choose not to name the association or links in this example since "...association names are optional and a matter of modelling judgement ... [and]... are often left unnamed when they can be easily identified by their classes.". If the purpose of data modelling is to provide semantically complete information about some application domain then it should be considered dangerous to leave components of the model unlabelled. The ERO model does not allow any unnamed entities or relationships.

The ERO model can represent the ternary relationship from Figure 3.5 with an overhead of three binary relationships as in Figure 3.6. Considering the rarity of higher order relationships this overhead is not considered significant in the ERO model. Unlike the OMT version, explicit semantic information is contained in the names of the binary relationships. The "Implemented-In" relationship is a relationship between the two entities, "Project" and "Language". It is a complex object because it participates in another relationship, "Works-On". The relationship "Knows" is a simple relationship between the entities "Language" and "Person". Also, if it were participating in another relationship in a larger data model, the "Works-On" relationship could be a complex object encompassing the relationship "Implemented-In" and the entity "Person". As in all ERO models the structure of the model represents the semantics for a specific context.

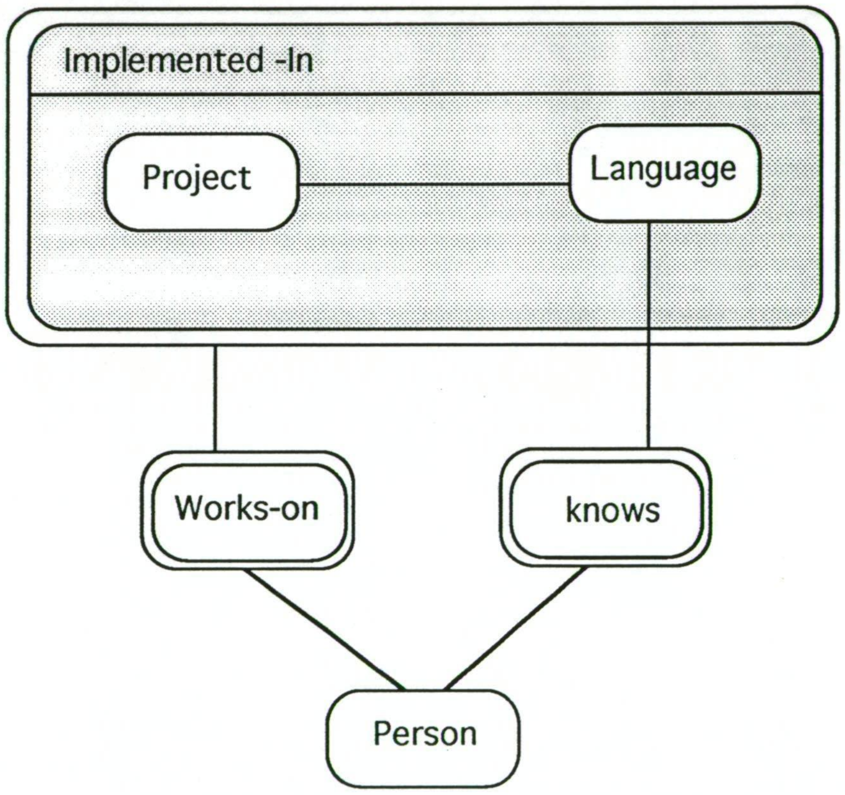


Figure 3.6 ERO Version of OMT Ternary Example

3.3.4 Cardinality

As in the EER models **cardinality** or the number of instances of one object that correspond to the instances of another object in a relationship are represented using "1" and "*" (indicating 'many') at the end of a relationship arc (see Figure 3.7)

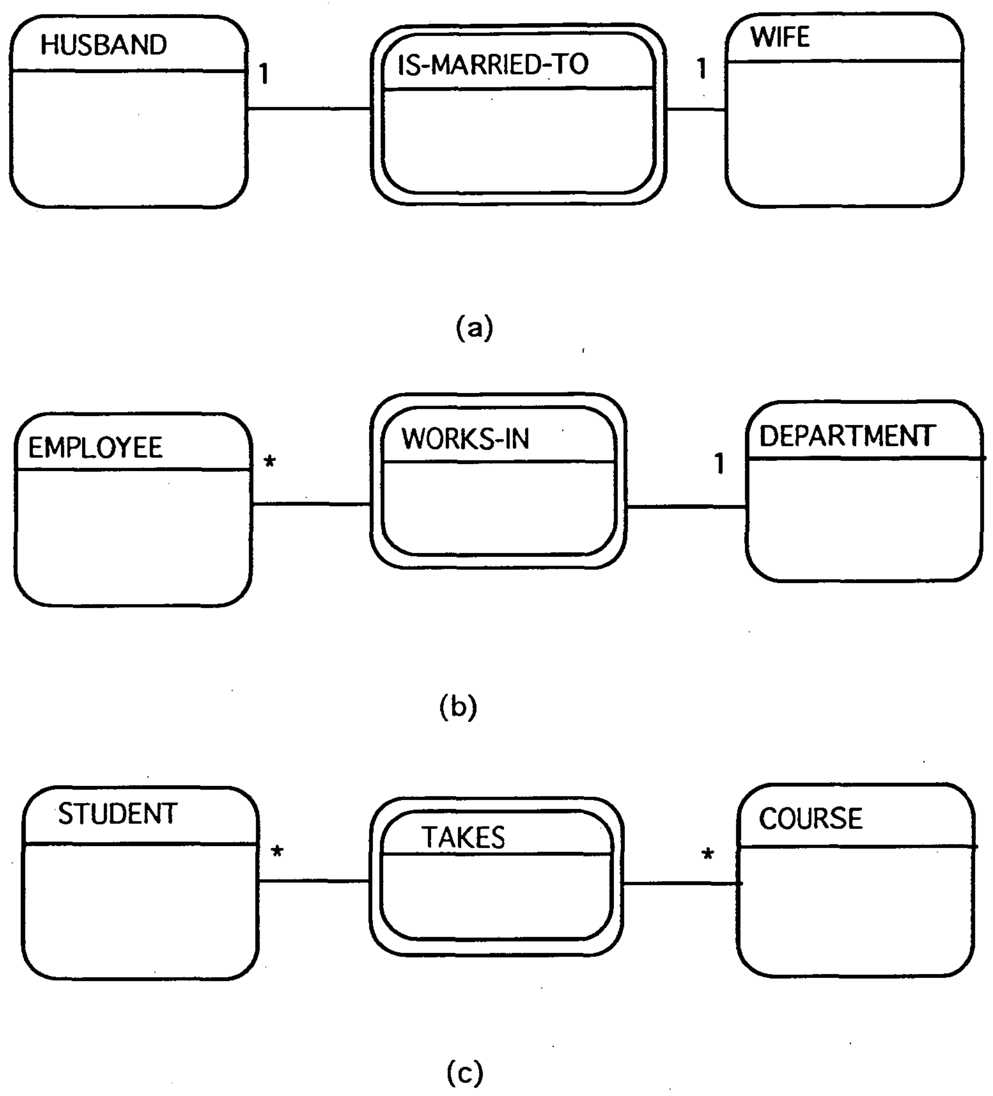


Figure 3.7
(a) A one-to-one relationship - a wife has one husband; a husband has one wife
(b) An employee works in one department; a department has many employees
(c) A student takes many courses; a course has many students

3.3.5 ERO Version of Hansen and Hansen's EER Example

Figure 3.8 is an ERO version of Hansen and Hansen's [1991] Manwaring Consulting Services Invoice model previously shown in Figure 2.6.

The complex object ON is a relationship between the entity PROJECT and the relationship ENGAGED-IN. ENGAGED-IN is a relationship between the entities CONSULTANT and ACTIVITY. It is represented as a complex object because it participates in the ON relationship. On the other hand, the relationship PERFORMED-FOR is a relationship between the entities

CLIENT and PROJECT and does not participate in any other relationship and so is modelled as a simple linear relationship.

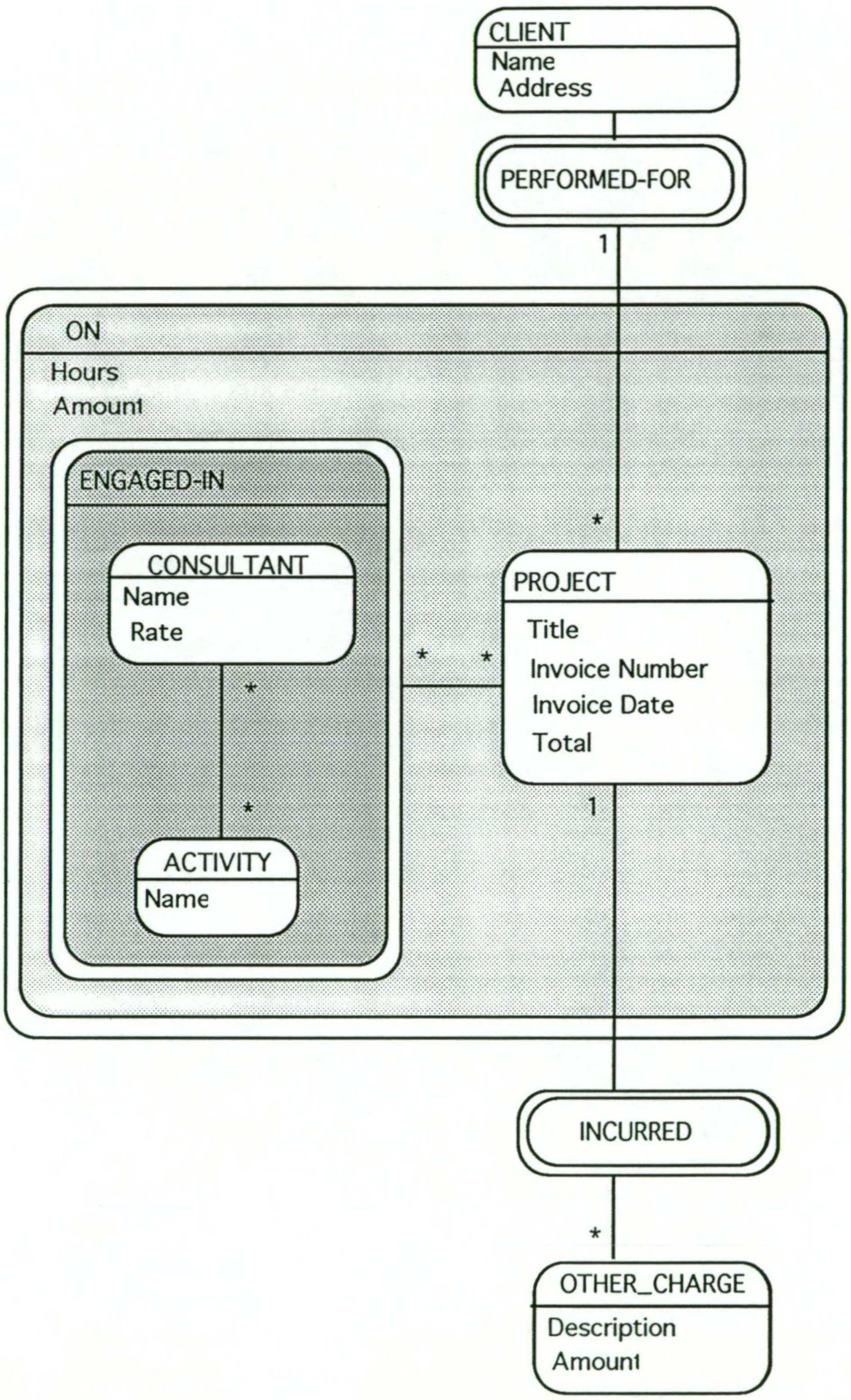


Figure 3.8 ERO Model of Hansen and Hansen's [1992] example as seen in Figure 2.6

[Note that a semantic inconsistency in Hansen and Hansen's [1991] original model has been corrected and there is now a Name attribute for the CLIENT entity and a Name attribute for the ACTIVITY entity.]

No matter how many levels of relationships there are within a complex relationship, the polymorphic nature of all classes means that every class has only one uid and for relationships that uid is a composite of only two other uids. For example, the ENGAGED-IN relationship has a composite uid from CONSULTANT and ACTIVITY and the ON relationship has a composite uid made up of the uids from ENGAGED-IN and PROJECT although the uid for ENGAGED-IN is itself a composite uid.

3.3.6 ERO Version of Rumbaugh's OMT Example

The example in Figure 3.9 is an ERO equivalent to the OMT model in Figure 2.7 . If it is presumed that the relationship `Works-In` is part of a larger data model where it participates in another relationship it can be modelled as a complex object which imports the entity-objects `Employee` and `Department`.

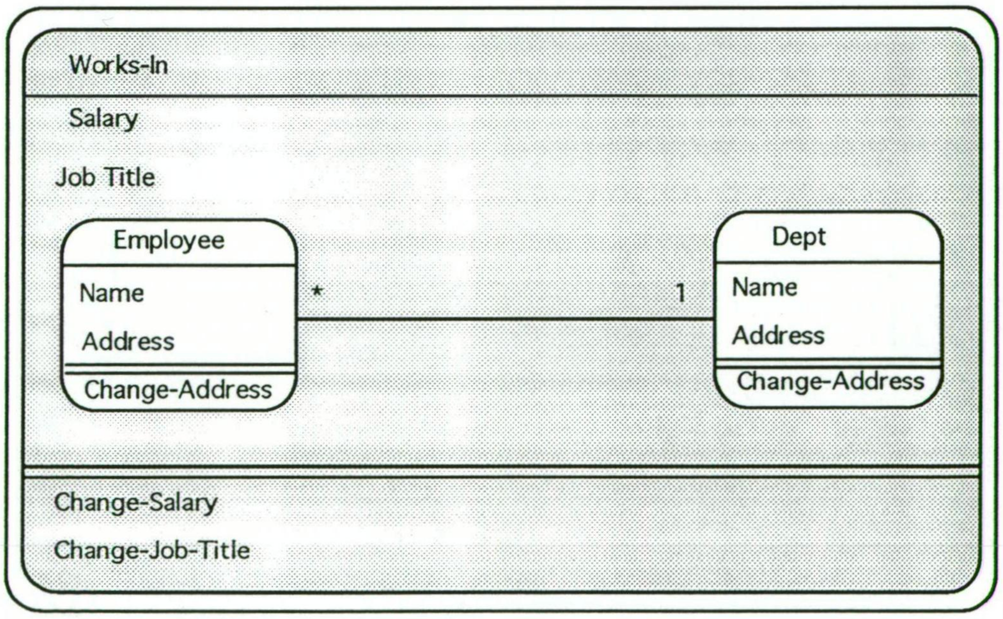


Figure 3.9 ERO Model of the OMT example from Figure 2.7

4 Implementation

4.1 Introduction

This chapter describes the implementation of an ERO model as a simple database system in a functional programming environment. Firstly, the conceptual framework is described for mapping an ERO model to an integrated implementation system where every entity and relationship is treated as a class. Secondly, considerations for mapping this conceptual framework to a functional programming environment using the functional programming language Haskell are discussed. Thirdly, a specific Haskell prototype is described and finally, a detailed example is shown providing specific program code within an implemented Haskell prototype.

4.2 Conceptual Framework

Conceptually in this implementation every component of the simple database system is considered to be an object. The main module imports all the components and thus links all the components together to form a database system.

4.2.1 The Database Subset

A database system usually includes facilities for creation and deletion of the database; update facilities for adding, deleting and modifying records; reporting facilities and query facilities. This implementation deals with the update operations: create, insert and delete plus operations for retrieving and displaying the current value of a class or object.

4.2.2 The User Interface

Because databases are usually dynamic structures the implementation uses an interactive menu-based interface where a user can add, delete, retrieve and display objects or classes within the system.

4.2.3 A Generic Superclass

This implementation adopts the view of Cardelli and Mitchell [1989] that "Object-oriented programming is based on record structures (called *objects*) intended as named collections of values (*attributes*) and functions (*methods*). Collections of objects form *classes*. A *subclass* relation is defined on classes with the intention that methods work 'appropriately' on all members belonging to the subclasses of a given class ... we are interested here in more powerful type systems that smoothly incorporate *parametric polymorphism.*"

Objects in a database system can be considered to be tuples or records of *key, value* pairs where the *key* is a unique identifier for an object and the *value* is the rest of the record. Using data abstraction the concept of "the rest of the record" can be extrapolated to include both attributes and methods. The implementation is based on the concept of a superclass which implements classes as collections of objects and objects as key, value pairs. This superclass is inherited by all classes in the system and provides the default methods for the addition, deletion or retrieval of any object in a class. Since this implementation is a prototype capable of further development the key (or unique identifier) is system generated but is visible to the user so that the user can access objects directly by their keys. A query interface, which could be a development added later, would not make use of visible keys.

4.2.4 Persistent Data

Like all database systems this implementation provides for persistent or long-lived stored data. This data is stored in ASCII files and has to be mapped onto the executing program in object or class form.

This is a functional implementation, so operations, or methods implemented as functions, cannot have side-effects. Since the result of the application of any function is a value, only the value need be stored as persistent data not the method itself.

4.2.5 An Abstract Implementation Scheme

The implementation is based on the mapping of the ERO model directly to program code in the prototype. The basic rules are as follows:

1. There should be a modularised abstract data type (ADT) to represent each class, entity or relationship, in the specific ERO model. Each ADT should contain definitions of all that class's attributes and methods. Each class should inherit the superclass and in the case of relationships also import any participating entities in that relationship since relationships provide the links between entities.

2. There should be a main module that imports all the components of the database thus linking all the components together.

3. A file for persistent data should be initialised for each class defined in 1.

4.3 The Haskell Environment

The power of functional languages lies in their characteristics of polymorphism, strong typing, lack of side effects and lazy evaluation. This implementation relies on these characteristics.

4.3.1 The Polymorphic Superclass

Haskell allows the implementer to define polymorphic types that can be applied to any arguments of other valid types. For example, [a] refers to a list of "things" where those things can be numbers, characters, strings or any other valid things including user defined types.

The superclass that is inherited by all classes in this implementation must necessarily be polymorphic so that it can be applied to any class, whether it be entity or relationship. The implementation provides the superclass as an ADT called *CollectionOf* which sets up an abstract type for a collection of key,value pairs.

The choice of data structure for any application depends on the "mix of operations" required by that application. That is, the frequency of insertions, deletions, traversals, retrievals etc. The overriding consideration for any ordered collection of data like a database is the efficiency of searching. Insertion depends on finding the correct place to insert a new record; deletion requires finding the correct record and all retrievals of information involve a search.

For this application the simplest, and possibly the most intuitive, data structure is a list of pairs or an association list. In Haskell a list is a dynamic data structure that can grow during execution as opposed to an array that requires its maximum size to be set at compile time. An association list is easy to implement and provides O(n) operations for insert, delete and retrieve based on a linear search algorithm.

A more efficient data structure, especially for large data sets, is a binary search tree which, if maintained as a balanced tree, provides O(logn) operations for insert, delete and retrieve based on a binary search algorithm.

The prototype provides the user with a choice of either an association list or a binary search tree but can use any polymorphic data structure that maintains a collection of key,value pairs and obeys the interface syntax.

Both data structures were implemented for testing purposes and to demonstrate that the specific implementation of the superclass is completely transparent to the rest of the program.

4.3.2 The Database Prelude

The implementation requires various utility or library functions particularly for I/O. Using standard Haskell practice these functions are provided in a "prelude" called *DBPrelude.* This also contributes to the modularity of the implementation so that an implementer needs only the superclass, prelude, main initialising module and a set of class definitions reflecting a particular ERO model to set up a new database.

4.3.3 The Main Module

The main initialising module presented in the prototype provides the linking mechanism for all the components of the database. It also provides the interface between the program and the persistent data stored on files. The main module imports all the components of the database (superclass, prelude, and class definitions) and sets up the top level menu and the lazy evaluated input streams from the user and the persistent data store.

4.3.4 Persistent Data Files

Persistent data for the implementation is stored as ASCII files which are mapped onto the program by functions in the prelude which are called by the main module. Lazy evaluation of input streams in Haskell means that the data files are not resident in memory for the duration of execution of the program but are accessed on a "need to know" basis as execution requires. This also means that updates to persistent data must be done explicitly using a save option.

4.3.5 Flow of Control using Continuation-based I/O

The implementation uses continuation-based I/O and continuation programming style as outlined in chapter 2. Every function that interacts with user input must be of type *Dialogue*. The flow of control in the program is maintained by cascading and/or recursive calls to functions of type *Dialogue*.

4.3.6 Simulating Global Variables

Since functional programming languages do not have assignment operators (all values are generated by the evaluation of a function) there is no concept of a global variable. A database system that generates its own unique identifiers (uids) needs to create a new uid for each new object in the system. This can be done in a functional language by setting up an infinite list of possible uids from which successive uids are retrieved or, as in this system, each new uid is generated as the successor of the last uid. This means that the current uid must be passed to every *Dialogue* function in the system during execution so that the "global variable" is defined for all program control functions.

In practice, each file storing an entity class has the last uid used for an object in that class saved to it. This allows the next uid for that entity class to be generated from that last uid.

4.4 The Haskell Prototype

Hudak and Fasel [1991] state that " ...modules provide the only way to build abstract data types (ADTs) in Haskell." Following this philosophy, this implementation builds all components of the system as ADTs implemented as separately compilable modules.

4.4.1 The ADT CollectionOf

The ADT *CollectionOf* is a separately compilable module representing the superclass. It maintains a collection of objects as a binary search tree. The type constructor *Collection* is applied to the two polymorphic variables *a* and *b*. The default methods for insertion, deletion and retrieval are provided by the functions *insert, delete* and *retrieve*. The function *flatten* maps the tree to a list and is used for I/O and tree balancing. The functions *bal* and *balance* provide an explicit operation for balancing the tree and are used after any insertion or deletion that might unbalance the tree. Figure 4.1 shows the complete module.

The implementation is based on an ADT that maintains a collection of key,value pairs. This ADT can take any form as long as it provides the operations: *retrieve, insert, delete, empty, flatten, bal* and *balance* and obeys the interface syntax. *Retrieve* has two arguments: a collection and a key and returns the key,value pair with that key. *Insert* takes three arguments: a collection, a key and a value and returns a new collection with that key,value pair inserted in the appropriate position. *Delete* takes two arguments: a collection and a key and returns a new collection with the pair corresponding to that key deleted. *Empty* defines an empty collection. *Flatten* maps a collection to a list, if necessary. *Bal* and *balance* should be do nothing for non-tree data structures.

```
module CollectionOf where
data Collection a b = Nil | Node (a,b) (Collection a b)(Collection a b)
                        deriving (Eq, Text, Binary)

empty = Nil

flatten Nil = []
flatten (Node (x,y) l r) = flatten l ++ [(x,y)] ++ flatten r

retrieve Nil z = error "Node not in tree"
retrieve (Node (x,y) l r) z | z == x = (x,y)
                            | z < x = retrieve l z
                            | z > x = retrieve r z

insert Nil x y = Node (x,y) Nil Nil
insert (Node (x1,y1) l r) x y | x == x1 = Node (x,y) l r
                              | x < x1 = Node (x1,y1) (insert l x y) r
                              | x > x1 = Node (x1,y1) l (insert r x y)

delete Nil z = error "Node not in tree"
delete (Node (x,y) l r) z | z == x = delitem (Node (x,y) l r)
                          | z < x = Node (x,y) (delete l z) r
                          | z > x = Node (x,y) l (delete r z)
--
--Delete a node :
--      1. If node is a leaf then just delete it
--      2. If the node has 1 child  then that node's child becomes one of that
--         node's parent's children (symmetric for left and right)
--      3. If the node has two children then replace it with its inorder
--         successor i.e. the leftmost node of that node's right subtree
--
delitem (Node (x,y) Nil Nil) = Nil
delitem (Node (x,y) Nil r) = r
delitem (Node (x,y) l Nil) = l
delitem (Node (x,y) l r) = Node (leftmost r) l (delete r (fst(leftmost r)))

leftmost (Node (x,y) l r) | l == Nil = (x,y)
                          | otherwise = leftmost l

bal [] = Nil
bal t = Node r (bal left) (bal right)
              where  n = length t
                     left = take ((div) n 2) t
                     r:right = drop ((div) n 2) t

balance t = bal (flatten t)
```

Figure 4.1

To demonstrate that an implementation that maintains a collection of pairs and obeys the interface syntax is valid and transparent to the rest of the system, Figure 4.2 presents the simple association list version of *CollectionOf*. The only necessary concessions are null versions of the tree-specific functions *flatten, bal* and *balance.*

```
module CollectionOf where
type Collection a b = [(a,b)]

empty = []

flatten l = l

bal l = l

balance t = t

retrieve [] x = error "UNDEFINED"
retrieve ((x,y):t) z | z == x = (x,y)
                     | z < x = error "UNDEFINED"
                     | z > x = retrieve t z

insert [] x y = [(x,y)]
insert ((x1,y1):t) x y  | x == x1  = (x,y):t
                        | x < x1   = (x,y):(x1,y1):t
                        | x > x1   = (x1,y1): insert t x y

delete [] z = []
delete ((x,y):t) z | z == x = t
                   | z < x  = (x,y):t
                   | z > x  = (x,y): delete t z
```

Figure 4.2

4.4.1 Types for Classes and Objects

For Entity-Objects (E/Os), each pair in the association list represents an object where the first item in the pair is the unique identifier (uid) for that object and the second item in the pair is a variable length string of strings representing the attributes of that object. Figure 4.3 shows the type definitions for E/Os.

```
    type Attribute = String

    type Uid = String

    type Info = [Attribute]


    type EntityObject = (Uid, Info)

    type EntityClass = Collection Uid Info
```

Figure 4.3

For R/Os, there is another level of nesting using the polymorphic nature of the superclass ADT. The first item in the R/O pair is in itself a pair providing the link between the two E/Os participating in the relationship thus producing a composite uid for the R/O. Conceptually, the type definitions for R/Os are shown in Figure 4.4.

```
        type Link = (Uid,Uid)

        type RelationshipObject = (Link, Info)

        type RelationshipClass = Collection Link Info
```

Figure 4.4

In practice, the type Link needs to be defined as an algebraic type to allow the overloading of equality and ordinal operators. This can be done in Haskell by setting up a user defined datatype and then defining instances of that type that specifically define the relational operators needed in the operations in the ADT *CollectionOf* as in Figure 4.5.

```
    data Link = Linkid (Uid, Uid)


    instance Eq Link where

        (Linkid (x, y)) == (Linkid (x1, y1)) = (x == x1 && y == y1)


    instance Ord Link where

        (Linkid (x,y)) < (Linkid (x1,y1)) = (x == x1 && y /= y1) ||

                                            (x <= x1 && x /= x1)

        (Linkid (x,y)) >= (Linkid (x1,y1)) = x1 <= x

        (Linkid (x,y)) > (Linkid (x1,y1)) = x1 < x
```

Figure 4.5

The data statement sets up a user-defined type called *Link* with a type constructor *Linkid* similar to a scalar or tagged type in an imperative language. The Haskell compiler could "infer" equality for two Links where the first item in each pair is equal and the second item in each pair is equal, but the system could not infer whether one pair is greater than another pair since each contains two discrete values. The equality and ordinal relationships of these Link pairs has to be spelled out explicitly using instance declarations.

Defining the equality and ordinality of links is purely arbitrary since they are not, by definition, part of an ordinal set. For simplicity, this implementation has defined ordinality to be based on the first uid in the link. Ordinality could just as easily have been defined based on the second uid in the link or some formula involving both uids. The choice is up to the implementer based on the specific application.

In Figure 4.5 the first instance declaration defines equality for this link type within the inbuilt definition or Haskell class **Eq** for equality. That is, we have defined two links to be equal if both items in each pair are equal. The second instance declaration defines the relational operators (which belong to the Haskell built in class **Ord**) for the type Link. This definition takes many-to-many relationships into account where the pair is distinct even though one of the items may be the same.

4.4.2 The DBPrelude

The *DBPrelude* (see Appendix) is a separately compilable module that provides standard utility functions for all update operations on all objects in the system. It also provides functions for the interactive, menu-based interface for adding, deleting, retrieving (by uid) and displaying any object, or set of objects, in the system.

The functions *enterentity/enterrelationship, findentity/findrelationship, deleteentity/deleterelationship,* and *displayentity/displayrel* place the polymorphic functions from the ADT *CollectionOf* into the appropriate I/O environment.

The *enter* functions are for user entry of new E/Os and R/Os. The *find* and *delete* functions prompt the user for a uid so that an object can be retrieved or deleted. The *display* functions display the current class on the screen. Display operations can be used by a user to find out a uid for the delete and find operations.

Most of the other functions in *DBPrelude* relate specifically to mapping ASCII strings to class types and vice versa for I/O and are of type *Dialogue* or *String*.

4.4.3 Function Synonyms

Continuation-based I/O, the passing of higher order functions and the need to carry around global variables can lead to a verbosity in functional programs particularly in terms of long and cumbersome parameter/argument lists. To make functions more readable this implementation uses "function synonyms" for often used patterns in argument lists. Figure 4.6 below shows the three synonyms used in the DBPrelude. *UserInput* is the infinite lazy input stream from the keyboard. *Continuation* is a synonym for the application of this input stream to a function of type *Dialogue*. *EntityFunction* and *RelationshipFunction* are synonyms for the standard set of arguments used by functions dealing with entities or relationships.

```
type UserInput = [String]

type Continuation = UserInput -> Dialogue

type EntityFunction = FileName -> Titles -> EntityClass -> Uid ->
                                 Continuation

type RelationshipFunction = FileName -> Titles -> RelationshipClass ->
                                 Continuation
```

Figure 4.6

4.5 A Mapping Methodology for the Haskell Prototype

The mapping methodology is based on the abstract mapping scheme presented in 4.2.5. Each E/O and R/O in the ERO model is mapped to an ADT implemented as a module containing definitions of all that class's attributes and methods. Each class inherits the superclass and in the case of relationships also imports any participating entities in that relationship.

A main module imports all the components of the database thus linking all the components together and a file for storing each class as persistent data is set up.

The prototype implementation requires three standard Haskell source modules:

collection.hs - defining the standard polymorphic ADT *CollectionOf* for the superclass

dbprelude.hs - defining the *DBPrelude,* containing all basic I/O and utility functions

maindb.hs - the main driver program containing the mandatory module *Main*

For any new implementation the modules *collection.hs* and *dbprelude.hs* remain unchanged but *maindb.hs* needs to be edited so that the main menu reflects the specific ERO model.

The following are the specific steps needed to set up an implementation using the Haskell prototype for a specific ERO model.

4.5.1. Create Class modules

For each E/O in the model create a source module that imports the generic, polymorphic ADT Collection and the DBPrelude containing the standard functions.

For each R/O create a module that imports the generic, polymorphic ADT Collection, the DBPrelude containing the standard library functions and the modules for each of the E/Os that participate in the relationship.

In each module declare a type for each attribute, a filename for where the persistent class will be stored and a list of titles, or labels, that can be used for I/O prompts and display formatting.

4.5.2. Add Methods

For any class that has methods associated with it, set up a methods menu and a methods driver function of similar format to the default methods menu and menu driver in the DBPrelude.

Implement method functions for each method using appropriate functions from the superclass and prelude.

4.5.3. Customise Main Menu for specific ERO Diagram

Edit the main program module, *maindb.hs* so that the menu contains a case instance for each E/O or R/O module created in 4.5.1 above. Then modify the main menu driver (i.e. the function *setClass*) to provide the appropriate function calls. E/O options call the function *readEntity* and R/O options call *readRelationship*. Each function call should pass parameters for filename, titles, the class's method driver function, an (empty) initialised class and the continuation for interactive input. The remainder of *maindb.hs* remains unchanged.

4.5.4. Initialise Persistent Data Stores

Create a file for each of the persistent data stores (i.e. one for each E/O or R/O). The prototype only provides for simple unique identifiers represented as integers chosen to suit the size of the data set. For example, for a data set not expected to exceed 99 objects in each class, seed each E/O file by inserting some integer as a starting unique identifier, e.g. the first E/O might be seeded with 100, the second with 200, the third with 300, and so on. The unique identifiers for R/Os are generated from the uids of each of the participating E/Os.

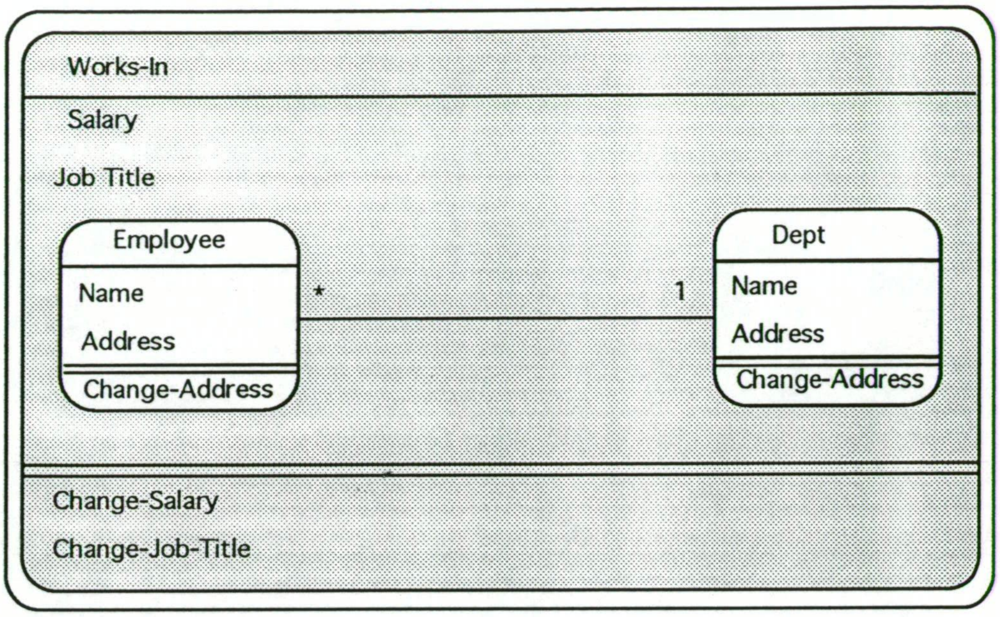
4.5.5. Compile, Link and Run

Compile and link all the modules. Execute *maindb*.

4.6 An Example

4.6.1 An ERO Model

The following worked example is based on the model in Figure 3.7 reproduced below. The R/O *Works-In* has two attributes: *Salary* and *Job Title* and two methods: *Change-Salary* and *Change-Job-Title*. Because it represents a relationship it also imports two E/Os: *Employee* and *Department*. The *Employee* E/O has two attributes: *Name* and *Address* and one method: *Change-Address*. The *Dept* E/O also has two attributes: *Name* and *Address* and one method: *Change Address*.



4.6.2 Step 1 of the Mapping Methodology

A module is created for each of the two E/Os: *Employee* and *Dept* participating in the relationship. Each E/O imports the superclass module and the prelude as shown for the E/O Employee below:

```
module EmployeeEntity where
import CollectionOf
import DBPrelude
```

Attributes, class declarations, titles and filename are added as follows:

```
--Attributes--
type EmpName = Attribute
type EmpAddress = Attribute

--Entity/Class Declarations--
type Employee = EntityObject
type Employees = EntityClass

--Titles--
employeeTitles = ["Employee Name","Employee Address"]
--Filename--
employeeFile = "Employees"
```

The module for the E/O *Dept* would be created in a similar way.

The R/O *Works-In* imports the superclass, the prelude and its participating E/Os as follows:

```
module WorksInRelationship where
import CollectionOf
import DBPrelude
import EmployeeEntity
import DeptEntity
```

Attributes, class declarations, titles and filename are added in the same way as in the E/Os as follows:

```
--Attributes--
type Salary = Attribute
type JobTitle = Attribute

--Relationship/Class Declarations--
type Worksin = RelationshipObject
type WorksIn = RelationshipClass

--Titles--
worksinTitles = ["Salary", "Job Title"]

--Filename--
worksInFile = "WorksIn"
```

4.6.3 Step 2 of the Mapping Methodology

Add a methods menu to the module of any class that uses methods. Below is the methods menu for the E/O *Employee*:

```
--Methods Menu--
empmenu = "\n" ++ employeeFile ++ "Operations\n" ++
              "10. Change an employee's address\n\n" ++
              "Hit return to return to main menu\n\n" ++
              "Command: "
```

Now, set up a driver function for the methods menu as below:

```
--Methods Driver--
empops :: (Continuation) -> EntityFunction
empops setClass fname titles coll lid inp =
 appendChan stdout empmenu abort
  (case inp of
   (line1 : rest) ->
    case (line1) of
     "10" -> appendChan stdout "Changing address\n" abort
              (empchangeaddress setClass empops fname titles coll lid rest)
     _    -> prompt setClass empops fname titles coll lid rest)
```

There will now be three levels of menus in the system. At the first level is the main menu controlled by function *setClass* (see 4.6.4) which chooses which class is currently being operated on. The second level menu is controlled by the function *prompt* provided in DBPrelude. This level provides functions for the default superclass methods (e.g. add, delete, retrieve etc). Adding new methods requires a third level menu of the form outlined above.

The main menu function *setClass* must be passed as a "global" function to allow recursive calls where appropriate. There must be a case instance for each method and an escape case instance (using the Haskell wildcard "_") for returning to the second level menu. In all instances the calls must carry, as parameters, the functions necessary for recursive calls - *setClass* for eventual return to the main menu and *empops* for recursive calls to the driver itself so another method may be chosen.

Lastly, a function for each method in the E/O should be added as below:

```
--Methods--
empchangeaddress :: (Continuation) -> ((Continuation) -> EntityFunction) ->
                      EntityFunction
empchangeaddress s ops fname titles coll lid inp =
  readItem "Enter Entity Id " inp
   (\i inp1 ->
    readItem "Enter new address " inp1
```

```
        (\a rest ->
          prompt s ops fname titles (insert colll (i,info)) lid rest
                 where  current = retrieve coll i
                        colll = delete coll (retrieve coll i)
                        atts = snd current
                        empname = head atts
                        info = empname:[a]))
```

Below is the methods menu for the R/O *Works In* which demonstrates that more than one method can be added to a class by adding lines to the menu and appropriate case instances in the driver.

```
--Methods Menu--
worksinmenu = "\n" ++ worksInFile ++ "Operations\n" ++
              "10. Change an employee's salary\n\n" ++
              "11. Change an employee's job title\n\n" ++
              "Hit return to return to main menu\n\n" ++
              "Command: "

--Methods Driver--
worksinops :: (Continuation) -> RelationshipFunction
worksinops setClass fname titles coll inp =
 appendChan stdout worksinmenu abort
  (case inp of
   (line1 : rest) ->
    case (line1) of
     "10" -> appendChan stdout "Changing salary\n" abort
             (workschangesalary setClass worksinops fname titles coll rest)
     "11" -> appendChan stdout "Changing job title\n" abort
             (workschangejob setClass worksinops fname titles coll rest)
     _    -> rprompt setClass worksinops fname titles coll rest)
```

Below is the implementation of the two methods associated with the R/O *WorksIn*. Each uses the *readItem* function from the prelude and passes control back to the default methods menu in the prelude with a call to the function *rprompt*.

```
--Methods--
workschangeasalary :: (Continuation) -> ((Continuation) ->
                         RelationshipFunction) -> RelationshipFunction
workschangesalary s ops fname titles coll inp =
  readItem "Enter Owner Id " inp
   (\id1 inp1 ->
    readItem "Enter Member Id " inp1
     (\id2 inp2 ->
      readItem "Enter new salary " inp2
       (\sal rest ->
         rprompt s ops fname titles (insert coll1 (link,info)) rest
              where  link = Linkid (id1,id2)
                     current = retrieve coll link
                     coll1 = delete coll (retrieve coll link)
                     oldsal:jobtitle = snd current
                     empname = head atts
                     info = sal:jobtitle)))

workschangeajob :: (Continuation) -> ((Continuation) ->
              RelationshipFunction) -> RelationshipFunction
workschangejob s ops fname titles coll inp =
  readItem "Enter Owner Id " inp
   (\id1 inp1 ->
    readItem "Enter Member Id " inp1
     (\id2 inp2 ->
      readItem "Enter new job title " inp2
       (\jobtitle rest ->
         rprompt s ops fname titles (insert coll1 (link,info)) rest
              where  link = Linkid (id1,id2)
                     current = retrieve coll link
                     coll1 = delete coll (retrieve coll link)
                     sal:oldjobtitle = snd current
                    cempname = head atts
                     info = sal:[jobtitle])))
```

4.6.4 Step 3 of the Mapping Methodology

Edit the main module *maindb.hs* to import the modules for the ADT *CollectionOf*, the *DBPrelude* and every class in the system as follows:

```
module Main where

import CollectionOf

import DBPrelude

import EmployeeEntity

import DeptEntity

import WorksInRelationship
```

Edit the class menu and menu driver (the function *setClass*) in *maindb.hs* to contain a case instance for each of the E/Os and R/O as below.

```
main :: Dialogue


main = appendChan stdout "Welcome\n\n" exit (

       readChan stdin exit (\userInput ->

       setClass (lines userInput)))


classmenu = "\n\nSelect a class as follows:\n" ++

              "1. Employees\n" ++

              "2. Departments\n" ++

              "3. Works In \n" ++

              "Class: "


setclass :: Continuation

setClass inp = appendChan stdout classmenu abort

   (case inp of

       (line1 : rest) ->

       (case line1 of

          "1" -> readEntity employeeFile employeeTitles empops empty rest

          "2" -> readEntity departmentFile deptTitles deptops empty rest

          "3" -> readRelationship worksInFile worksinTitles worksinops empty rest

          _   -> appendChan stdout "error - try again" abort (setClass rest))
```

4.6.5 Step 4 of the Mapping Methodology

Create a file for the E/O *Employee* containing the initial uid, say, 100. Create a file for the E/O *Dept* containing the initial uid, say, 200. Create an empty file for the R/O *Works In* since R/O uids are composed of participating E/O uids.

4.6.6 Step 5 of the Mapping Methodology

Compile and link all the modules. Execute *maindb*. Control is passed to the menu based user interface.

5 Discussion, Conclusions and Future Work

This chapter discusses some of the advantages and limitations associated with the design and implementation methodology presented in this thesis. Some areas for future work are also discussed.

5.1 Advantages

5.1.1 Integrated Methodology

The design and implementation methodology presented in this thesis is an integrated one in that it provides a model and a methodology for mapping that model directly to an implementation in a functional programming language.

A designer can model a database as a set of interrelated classes using the ERO model. From the ERO model a system can be directly implemented in the functional language Haskell by following the steps of the mapping methodology as described in Chapter 4 and including the standard database prelude and a suitable polymorphic ADT defining classes.

This direct mapping of the ERO model requires only small modifications to the driver program and the addition of standard format modules for each class in the ERO diagram.

5.1.2 Preservation of semantics

The ERO model provides for the preservation of the semantics of a modelled system largely due to its derivation from the Entity-Relationship Model [Chen,1976] and the incorporation of object oriented concepts by adding behaviour to the model using methods. The ERO model is further enhanced by treating not just entities as classes but also relationships (associations in object-oriented terminology) as classes. This gives a more abstract view of a data model than currently available semantic and object models, where all identified components of a data model are treated as objects.

5.1.3 Functionally Complete for Update

The implementation described in this thesis is functionally complete for the basic update operations of: insert an object, retrieve an object (by uid) and delete an object. Necessary associated operations for loading, saving and displaying objects are also provided together with a simple menu driven user interface. No attempt has been made to provide querying facilities. The development of querying facilities is discussed in 5.3.2 below.

5.1.4 Ease of Schema Modification

Like the Entity-Relationship Model [Chen, 1976] the ERO model is easily modified or extended. The implementation methodology used here also provides for ease of modification by its modular structure. Any new entity class or relationship class that is added to the ERO model can be implemented as a separately compilable Haskell module using the steps of the mapping methodology described in Chapter 4. Only the module itself and the modules that import it need to be recompiled.

5.1.5 Advantages of using Haskell as an implementation language

The main advantages of using Haskell in this implementation lie in the power of polymorphism, including overloading, and lazy evaluation. Polymorphism and the ability to overload arithmetic operators allows for the definition of a truly generic superclass that is the basis of all classes in the system. Lazy evaluation allows for the implementation of a form of persistence where objects are not resident in memory but retrieved from secondary storage as needed and are explicitly updated on secondary storage after completion of a transaction.

5.2 Limitations

5.2.1 Localisation of Methods in Object-Oriented Models

The main limitation of this database implementation is the same as that for any object-oriented database system. Object-oriented models tend to constrain the modelling of database systems because of the localisation of methods to specific objects or classes. Many database transactions require the participation of several entities and relationships. For example, consider a model taken from McFadden and Hoffer [1991]. Figure 5.1 shows the complete model. Attributes have been omitted for simplicity.

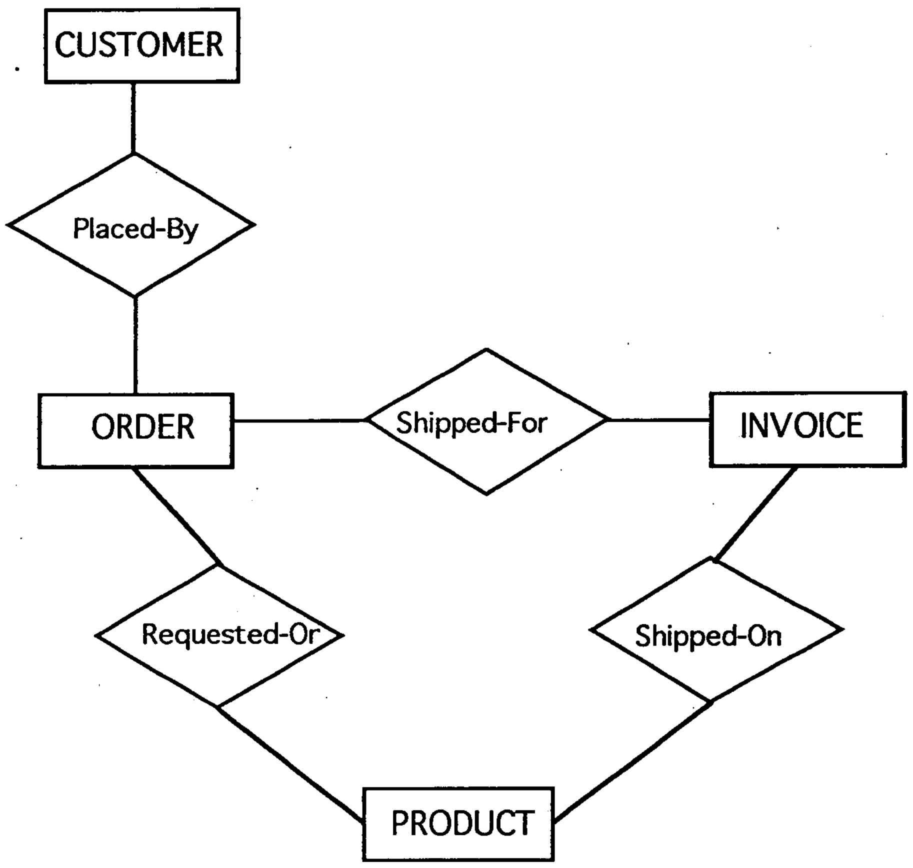


Figure 5.1
McFadden and Hoffer's Enterprise data model for the Pine Valley Furniture Company.

One management requirement of the system is for a *daily order log report* which would require a method covering the part of the model shown in Figure 5.2.

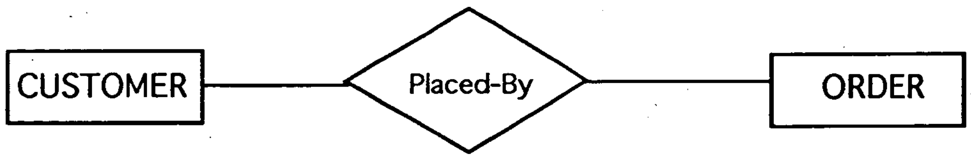


Figure 5.2
ER diagram for daily order log report for enterprise shown in Figure 5.1.

Another requirement is for a *customer order query* for which a method covering the section of the diagram shown in Figure 5.3 would be required.

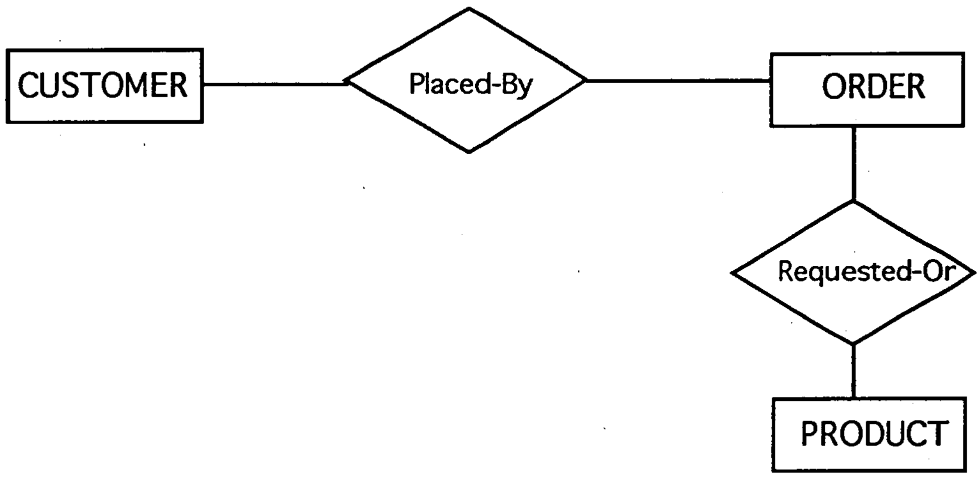


Figure 5.3
ER diagram for customer order history query for enterprise shown in Figure 5.1.

In each of these examples the method to generate the report or satisfy the query does not logically belong to any single object but to some group of objects. This constraint needs to be addressed for object modelling to be useful in database design. An attempt to address this is outlined in 5.3.4 below.

5.2.2 Statelessness of Functional Programming

Database systems are dynamic systems and database operations often require knowledge of the state of the system or parts of the system. As discussed in 2.4.1, functional languages are stateless and any state-oriented computations are achieved by passing the state around explicitly. The method used in this prototype for passing state information around using pseudo global variables is discussed in 4.3.6. This can lead to the need for long parameter/argument lists, sometimes spreading over several lines. This makes programs difficult to read and sometimes difficult to maintain. Some steps have been taken in this implementation to address this problem by incorporating "function synonyms" as discussed in 4.4.3.

Current work involving the addition of monads into functional languages like Haskell [Wadler, 1990] may lead to the incorporation of a notion of "state" into functional programming systems.

5.2.3 Very Large Database Management Systems

The prototype in this implemenation has only been tested on small data sets of up to eight classes and less than 50 objects/class. The problems associated with the design and implementation of Very Large Database Management Systems have not yet been considered.

5.2.4 Variety of Data Types

For simplicity, the prototype implementation presented here only allows for data to be of type *String*. This allows for a smooth interface between the run time environment and the persistent data store, since Haskell treats files as single ASCII strings. Of course, database systems should manage data of any type including numeric, or complex combinations of several types. In order to implement other data types, the prototype would need to include conversion functions for mapping other types to, and from,*String* types for secondary storage.

5.3 Future Work

5.3.1 A Graphical Interface Development Tool and Compiler

The integrated nature of the design and implementation method presented here lends itself to the development of a graphical model development interface with which a user could produce an ERO model directly on the screen. The graphical tool could provide standard icons for entity-classes and relationship classes and a system of colouring complex relationships. A compilation system could then take the graphical model and produce a direct implementation in Haskell code using the mapping methodology outlined in chapter 4. Specific methods would still have to be hand coded but all other parts of the mapping methodology should be capable of being automated.

5.3.2 A Functional Query Interface

A functional query interface could be developed for this implementation using some of the built-in functions already available in Haskell. Many database queries are based on selecting objects from a class or classes which satisfy some predicate. These kinds of operations are naturally functional. That is, a query is the application of a function to a database that returns some value.

A good starting point for a query interface would be the Haskell function in the style of *filter*. The function *filter* is provided in one of Haskell's standard preludes called *PreludeList* and is described as follows:

```
--filter, applied to a predicate and a list,
--returns the list of those elements that satisfy the
--predicate; i.e.,
--filter p xs == [x | x <- xs, p x]
filter     :: (a -> Bool) -> [a] -> [a]
filter p   = foldr (\x xs -> if p x then x:xs else xs) []
```

Functions could be developed that are applied to a predicate and a class and return a list of all objects satisfying that predicate. This approach would allow queries to be expressed explicitly as functions that can be applied directly to classes of objects.

5.3.3 Private/Public Class Interfaces

This implementation has not made use of Haskell's provision for explicit control over import/export across module boundaries. The prototype uses modules in their simplest form where, by default, all functions within any module are exported and available to other importing modules. The ability to control the import/export of functions across module boundaries could be investigated for implementing private and public operations within classes.

5.3.4 Transaction and Report Modelling

The problem of the localisation of methods discussed above in 5.2 needs to be addressed in terms of providing methods for complex transaction and report modelling required by database systems. One area that could be investigated is the extension of the ERO model to another level of modelling for transactions and reports. At this new level transactions and reports could be based on some form of the complex relationships already existing in the model where a transaction or report imports all the participating classes and can be implemented as a separate module. For example, consider the example from McFadden and Hoffer [1991] given in 5.2. The customer order query could be modelled as in Figure 5.4.

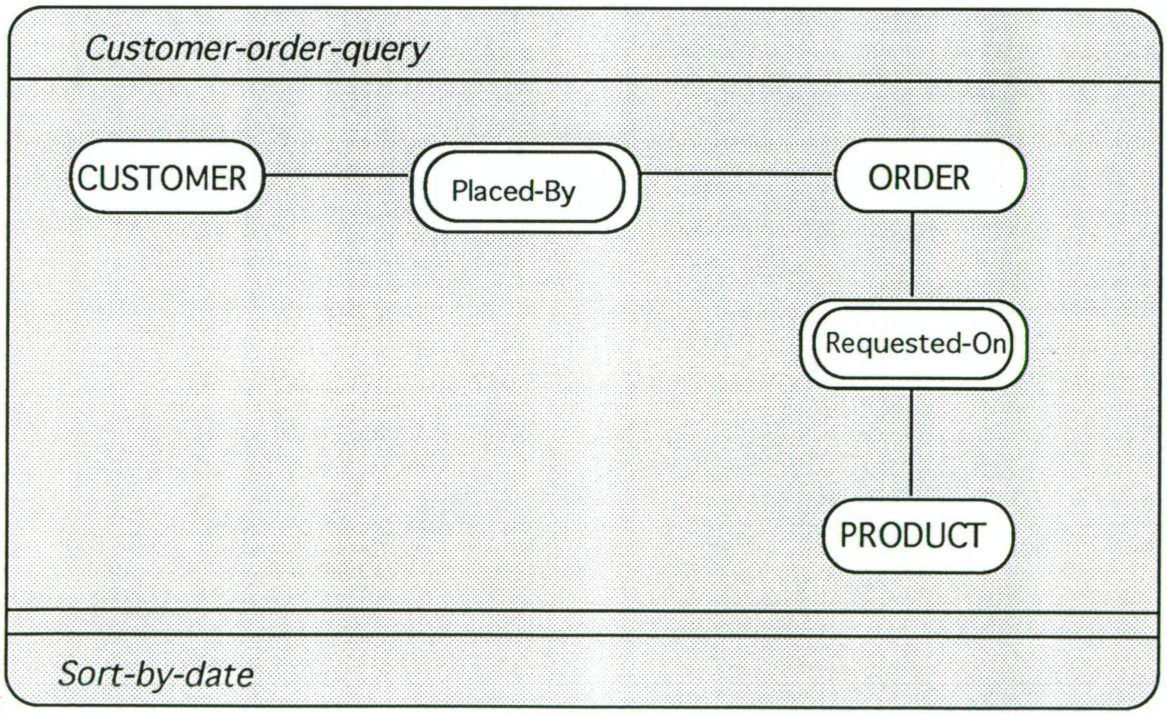


Figure 5.4
A transaction modelling scheme for Figure 5.3

Conceptually, queries and reports like these could be considered to be another type of class with their own unique identifiers and could be implemented as separate modules that import the classes that participate in the transaction or report.

Appendix - The DBPrelude

```
-------------------------------------------------------------------
--Standard database prelude
--Must import a polymorphic ADT that provides at least the standard
--operations:
-- retrieve
-- insert
-- delete
-- Defines the standard types and methods for generic classes
--
-- Written by Linda Dawson
-- Last Modified November, 1993
-------------------------------------------------------------------
module DBPrelude where
import CollectionOf

-------------------------------------------------------------------
--STANDARD TYPES
-------------------------------------------------------------------
type Attribute = String
type Uid = Attribute             -- the key or uid for an entity object
type Info = [Attribute]          -- variable length list of attributes
type EntityObject = (Uid, Info)         -- the entity object type
type EntityClass = Collection Uid Info  -- a collection of entities
data Link = Linkid (Uid, Uid)           -- the link for a relationship

instance Eq Link where
        (Linkid (x, y)) == (Linkid (x1, y1)) = (x == x1 && y == y1)

instance Ord Link where
        (Linkid (x, y)) < (Linkid (x1, y1)) = (x == x1 && y /= y1) ||
                                               (x <= x1 && x /= x1)
        (Linkid (x, y)) >= (Linkid (x1, y1)) = x1 <= x
        (Linkid (x, y)) > (Linkid (x1, y1)) = x1 < x

type RelationshipObject = (Link, Info)   -- the realtionship object
type RelationshipClass = Collection Link Info
type FileName = String                   -- file names
type UserInput = [String]                -- input from the keyboard
type Title = String                      -- attribute label
type Titles = [Title]                    -- each class has a set of
                                         -- attribute labels

-------------------------------------------------------------------
--UTILITIES
-------------------------------------------------------------------
--Function synonyms

type Continuation = UserInput -> Dialogue
type EntityFunction = FileName -> Titles -> EntityClass -> Uid ->
Continuation
type RelationshipFunction = FileName -> Titles -> RelationshipClass ->
Continuat
ion

-- Null methods drivers
eops :: Continuation -> EntityFunction
eops s f t c l i = appendChan stdout "No methods for this class\n"
abort
                        (prompt s eops f t c l i)

rops :: Continuation -> RelationshipFunction
rops s f t c i = appendChan stdout "No methods for this class\n" abort
                        (rprompt s rops f t c i)

-- Field delimiters should  allow spaces in attributes
fieldDelim = '*'
tab = '\t'
doubletab = "\t\t"
space = ' '
isDelim c = c == fieldDelim

----------------------------------------------------------------------
-- Formatting functions

width = 20
idwidth = 5
linkwidth = 10

rep :: Int -> a -> [a]
rep 0 x = []
rep (n+1) x = x:rep n x
rep other x = []

spaces :: Int -> String
spaces n = rep n ' '
ljustify :: Int -> String -> String
ljustify n s = s ++ spaces (n - length s)

rjustify :: Int -> String -> String
rjustify n s = spaces (n - length s) ++ s
----------------------------------------------------------------------
-- Removes field delimiters from attributes in a persistent class

makefields :: String ->  Info
makefields line = case dropWhile isDelim line of
                   "" -> []
                   line1 -> field: makefields line2
                            where (field, line2) = break isDelim line1

----------------------------------------------------------------------
-- Converts a set of attributes into a single string for output
--
unmakefields :: Char -> [String] -> String
unmakefields  c [] = ""
unmakefields c fs = foldr1 (\w s -> (ljustify width w) ++ c:s) fs

----------------------------------------------------------------------
-- Converts an entity class into a string for output
--
makeopEntity  [] = []
makeopEntity (firstrec:rest) =
                 makeopEntityObject firstrec ++ makeopEntity rest

----------------------------------------------------------------------
-- Converts an entity object into a string for output
--
makeopEntityObject (i, rest) =
        i ++ [fieldDelim] ++ (unmakefields fieldDelim rest) ++ "\n"

----------------------------------------------------------------------
-- Converts a relationship class into a string for output
--
makeopRelationship [] = []
makeopRelationship (firstrec:rest) =
                         makeopRel firstrec ++ makeopRelationship rest
----------------------------------------------------------------------
-- Converts a relationship into a string for output
--
makeopRel ((Linkid (x,y)), rest) =
      x ++ [fieldDelim] ++ y ++ [fieldDelim] ++
            (unmakefields fieldDelim rest) ++ "\n"

----------------------------------------------------------------------
-- Converts a string into an entity class
--
makeentityclass :: UserInput -> EntityClass -> EntityClass
makeentityclass [] e = balance e
makeentityclass (line:rest) e =
                        makeentityclass rest (insert e x y)
                        where      eo = makeentity (makefields line)
                                   x = fst eo
                                   y = snd eo

----------------------------------------------------------------------
-- Converts a list of attributes into an entity object
--
makeentity :: Info -> EntityObject
makeentity (field:fields) = (field,fields)

----------------------------------------------------------------------
-- Converts a string into a relationship class

makerelationship :: UserInput -> RelationshipClass ->
RelationshipClassmakerelationship [] r = balance r
makerelationship (line:rest) r = makerelationship rest (insert r x y)
                                 where  ro = makerel (makefields line)
                                        x = fst ro
                                        y = snd ro

----------------------------------------------------------------------
-- Converts a list of attributes into a relationship object

makerel :: Info -> RelationshipObject
makerel (field1:field2:fields) = (Linkid(field1,field2),fields)

----------------------------------------------------------------------
-- Displays an entity class
--
showEntityclass [] = []
showEntityclass (firstrec:rest) =
                        showentity firstrec ++ showEntityclass rest

----------------------------------------------------------------------
-- Displays a relationship class

showrelationship [] = []
showrelationship (firstrec:rest) =
                        showrel firstrec ++ showrelationship rest

----------------------------------------------------------------------
--Displays an entity object
--
showentity (i,rest) =
            (ljustify idwidth i) ++ (unmakefields tab rest) ++ "\n"

----------------------------------------------------------------------
--Displays a relationship object
```

```
showrel ((Linkid (x,y)), rest) =
                        (ljustify linkwidth link) ++
                        (unmakefields tab rest) ++ "\n"
                             where  link = "(" ++ x ++ "," ++ y ++ ")"

----------------------------------------------------------------------
-- Displays the current entity class
--
displayentity s eclass = "\n" ++ (ljustify idwidth "Uid") ++
                              (dispetitles s eclass)

----------------------------------------------------------------------
-- Displays the current relationship class

displayrel s rclass = "\n" ++ (ljustify linkwidth "Uid") ++
                              (disprtitles s rclass)

----------------------------------------------------------------------
-- Display entity titles

dispetitles [] coll = "\n" ++ (showEntityclass coll)
dispetitles (t:ts) coll = (ljustify width t) ++ (dispetitles ts coll)
----------------------------------------------------------------------
-- Display relationship titles

disprtitles [] coll = "\n" ++ (showrelationship coll)
disprtitles (t:ts) coll = (ljustify width t) ++ (disprtitles ts coll)
----------------------------------------------------------------------
--STANDARD MENUS
----------------------------------------------------------------------
updatemenu = "\n\nEnter a command as follows:\n" ++
      "1. insert object - inserts a new object in the database\n" ++
      "2. retrieve - returns a record\n" ++
      "3. delete - deletes the record with key, id\n" ++
      "4. print - displays the database\n" ++
      "5. options - for current class\n" ++
      "6. change to another class\n" ++
      "7. save current class\n" ++
      "8. quit - exit program\n\n" ++
      "Command: "

prompt :: Continuation -> (Continuation -> EntityFunction) ->
EntityFunction
prompt s ops fname titles coll lid inp =
 appendChan stdout updatemenu abort
   (case inp of
      (line1 : rest) ->
         case (line1) of
            "1" -> appendChan stdout "inserting\n" abort
              (enterentity s ops fname titles titles [] coll lid rest)
            "2" -> appendChan stdout "retrieving\n" abort
                   (findentity s ops fname titles coll lid rest)
            "3" -> appendChan stdout "deleting\n" abort
                   (deleteentity s ops fname titles coll lid rest)
            "4" -> appendChan stdout
                       (displayentity titles (flatten coll))abort
                       (prompt s ops fname titles coll lid rest)
            "5" -> appendChan stdout "options\n" abort
                       (ops s fname titles coll lid rest)
            "6" -> appendChan stdout "changing\n" abort
                       (s rest)
            "7" -> appendChan stdout ("saving" ++ fname) abort
```

```
                         (writeFile fname (lid ++ "\n" ++
                          (makeopEntity (flatten coll))) abort
                         (s rest))
             "8" -> appendChan stdout "goodbye\n" abort done
             _   -> appendChan stdout "error - try again"
                      abort (prompt s ops fname titles coll lid rest))

rprompt :: Continuation -> (Continuation -> RelationshipFunction) ->
RelationshipFunction
rprompt s ops fname titles coll inp =
 appendChan stdout updatemenu abort
  (case inp of
     (line1 : rest) ->
        case line1 of
          "1" -> appendChan stdout "ins rel\n" abort
            (enterrelationship s ops fname titles titles1 [] coll rest
             where titles1 = ["Enter owner id: "]
                             ++["Enter member id: "] ++ titles)
          "2" -> appendChan stdout "retrieving rel\n" abort
                         (findrel s ops fname titles coll rest)
          "3" -> appendChan stdout "deleting rel\n" abort
                         (deleterel s ops fname titles coll rest)
          "4" -> appendChan stdout (displayrel titles (flatten coll))
                         abort (rprompt s ops fname titles coll rest)
          "5" -> appendChan stdout "options\n" abort
                         (ops s fname titles coll rest)
          "6" -> appendChan stdout "changing\n" abort
                         (s rest)
          "7" -> appendChan stdout ("saving" ++ fname) abort
                 (writeFile fname
                        (makeopRelationship (flatten coll)) abort
                        (s rest))
          "8" -> appendChan stdout "goodbye\n" abort done
          _   -> appendChan stdout "error - try again"
                         abort (rprompt s ops fname titles coll rest))


-----------------------------------------------------------------------
--UPDATE OPERATIONS FOR CLASSES
-----------------------------------------------------------------------
-- Reads entity data typed in at the terminal
-- one attribute per Title
--
enterentity :: Continuation -> (Continuation -> EntityFunction) ->
FileName -> Titles -> Titles -> Info -> EntityClass -> Uid ->
Continuation

enterentity s ops fname tl [] info coll lid rest =
 prompt s ops fname tl (balance (insert coll nextid info)) nextid rest
               where nextid = show ((read lid)+1)
enterentity s ops fname tl (t:ts) info coll lid rest =
 readItem t rest
  (\i rest1 -> enterentity s ops fname tl ts
                  (info ++ [i] ) coll lid rest1)

-----------------------------------------------------------------------
-- Sets up a relationship between 2 objects
--
enterrelationship :: Continuation -> (Continuation ->
RelationshipFunction) -> FileName -> Titles -> Titles -> Info ->
RelationshipClass -> Continuation

enterrelationship s ops fname tl [] (id1:id2:info1) coll rest =
 rprompt s ops fname tl (balance (insert coll link1 info1)) rest
                where   link1 = Linkid (id1,id2)
enterrelationship s ops fname tl (t:ts) info coll rest =
 readItem t rest
  (\i rest1 -> enterrelationship s ops fname tl ts
                 (info ++ [i] ) coll rest1)
----------------------------------------------------------------------
-- Finds an entity object
--
findentity :: Continuation -> (Continuation -> EntityFunction) ->
EntityFunction

findentity s ops fname titles coll lid inputLines =
  readItem "Enter Entity Id: " inputLines
    (\i rest -> appendChan stdout (showentity (retrieve coll i)) abort
                      (prompt s ops fname titles coll lid rest))
----------------------------------------------------------------------
-- Finds a relationship object

findrel :: Continuation -> (Continuation -> RelationshipFunction) ->
RelationshipFunction

findrel s ops fname titles coll inputLines =
  readItem "Enter owner id: " inputLines(\x inputLines1 ->
           readItem "Enter member id: " inputLines1
               (\y rest -> appendChan stdout
                      (showrel (retrieve coll (Linkid (x, y)))) abort
                      (rprompt s ops fname titles coll rest)))
----------------------------------------------------------------------
-- Deletes an entity object
--
deleteentity :: Continuation -> (Continuation -> EntityFunction)->
EntityFunction

deleteentity s ops fname titles coll lid inputLines =
  readItem "Enter Entity id: " inputLines
        (\i rest ->
           prompt s ops fname titles (delete coll i) lid rest)
----------------------------------------------------------------------
-- Deletes a relationship object

deleterel :: Continuation -> (Continuation -> RelationshipFunction) ->
RelationshipFunction

deleterel s ops fname titles coll inputLines =
 readItem "Enter owner id: " inputLines
    (\x inputLines1 ->
        readItem "Enter member id: " inputLines1
          (\y rest ->
              rprompt s ops fname titles
                      (delete coll (Linkid (x,y))) rest))
----------------------------------------------------------------------
-- Reads a single string from the keyboard
--
readItem :: Title -> UserInput -> (String -> [String] -> Dialogue) ->
Dialogue

readItem aprompt inputLines succ = appendChan stdout aprompt abort
      (case inputLines of
          (x : rest) -> succ x rest
          _       -> appendChan stdout "EOF" abort done)
```

References

Atkinson M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D., and Zdonik, S. 1990 'The Object-Oriented Database System Manifesto' in *Deductive and Object-Oriented Databases,* eds W. Kim, J.-M. Nicolas and S. Nishio, Elsevier Science Publishers B.V. (North Holland), pp 223-240.

Beck, K. and Cunningham, W. 1989 'A laboratory for teaching object-oriented thinking' *OOPSLA '89 Proceedings,* pp 1-6.

Boehm, B.W. 1988 'A spiral model of software development and enhancement' *IEEE Computer,* Vol 25, No 5, pp 61 -72.

Booch, G. 1986 'Object-oriented development' *IEEE Transactions on Software Engineering,*Vol 12, No 2, pp 211-221.

Bretl, R., Maier, D., Otis, A., Penney, B., Schuchardt, B., Stein, J., Williams, E.H. and Williams, M. 1990 'The Gemstone data management system. In *Object-Oriented Concepts, Databases and Applications,* Kim, W. and Lochovsky, F.H. eds. pp 283-308. ACM Press Books.

Cardelli, L. and Mitchell, J.C. 1989 'Operations on Records.' In *Proceedings of 5th International Conference on Mathematical Foundations of Programming Semantics,* New Orleans, Louisiana, USA, March 1989.

Chen, P.P. 1976 'The Entity-Relationship Model: Toward a Unified View of Data.' *ACM Transactions on Database Systems,* Vol 1, No 1, pp 9-36.

Codd, E.F. 1970 'A relational model of data for large shared data banks' *Communications of the ACM,* Vol 13, No 6, pp 377-387.

Deux, O. 1990 'The story of O2.' *IEEE Transactions on Knowledge and Data Engineering,* Vol 2, No 1, pp 91-108.

Dobbie, G. 1991 'Object Oriented Database Systems: A Survey'*Proceedings of the Fourteenth Australian Computer Science Conference,* Sydney, Australia, February 6-8, 1991.

Elmasri, R. and Navathe, S.B. 1989 *Fundamentals of Database Systems* The Benjamin/Cummings Publishing Company, Inc.

Elmasri, R., Weeldreyer, J. and Hevner, A. 1985 'The Category Concept: An Extension to the Entity-Relationship Model', *International Journal on Data and Knowledge Engineering,* Vol 1, No 1.

Fong, E., Kent, W., Moore, K. and Thompson, C. 1991 *X3/SPARC/DBSSG/OODBTG Final Report.*

Hammer, M., and Mcleod, D. 1981 'Database description with SDM: A semantic database model.' *ACM Transactions on Database Systems,* Vol 6, No 3, pp 351-386.

Hansen, G. W. and Hansen, J.V. 1992 *Database Management and Design* Prentice-Hall.

Henderson-Sellers, B. 1992 *A Book of Object-Oriented Knowledge* Prentice-Hall.

Hudak, P. 1989 'Conception, Evolution, and Application of Functional Programming Languages' *ACM Computing Surveys,* Vol 21, No 3, pp 359-411.

Hudak P. and Fasel, J. 1992 'A Gentle Introduction to Haskell' *ACM SIGPLAN Notices,* Vol 27, No 5, Section T.

Hudak, P., Peyton Jones, S., Wadler, P., Boutel, B., Fairbairn, J., Fasel, J., Guzman, M.M., Hammond, K., Hughes, J., Johnsson, T., Kieburtz, D., Nikhil, R., Partain, W. and Peterson, J. 1992 'Report on the Programming Language Haskell. A Non-strict, Purely Functional Language, Version 1.2' *ACM SIGPLAN Notices,* Vol 27, No 5, Section R.

McFadden, F.R. and Hoffer, J.A. 1991 *Database Management,* (Third Edition), Benjamin/Cummings.

Nijssen, G.M. and Halpin, T.A. 1989 *Conceptual Schema and Relational Database Design: A Fact Oriented Approach* Prentice-Hall.

Nikhil, R. S. 1985 'Functional Databases, Functional Languages. Extended Summary.' In *Proceedings 1985 Persistence and Data Types Workshop Appin, Scotland,* August 1985.

ONTOS Object Database, *Learning to use ONTOS for OS/2 Systems,* 1989.

Peckham, J. and Maryanski, F. 1988 'Semantic Data Models' *ACM Computing Surveys*, Vol 20, No 3, pp 153-189.

Rumbaugh J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W. 1991 *Object-Oriented Modeling and Design* Prentice-Hall, Englewodd Cliffs, NJ.

Shipman, D.W. 1981 'The functional data model and the data language DAPLEX' *ACM Transactions on Database Systems,,* Vol 6, No 1, pp 140-173.

Shlaer, S. and Mellor, S.J. 1989 *Object-Oriented Analysis: Modeling the World with Data*. Prentice-Hall, Englewood Cliffs, NJ.

Wadler, P. 1990 'Comprehending Monads' In *Proceedings of 1990 ACM Conference on LISP and Functional Programming,* Nice, France, June 1990.

Wegner, P. 1987 'Dimensions of object-based language design.' In *OOPSLA '87 Proceedings,* New York, ACM.

Wilkinson, K., Lyngboek, P. and Hasan, W. 1990 'The Iris architecture and implementation'. *IEEE Transactions on Knowledge and Data Engineering,* Vol 2, No 1, pp 63-75.