An Overview of the Object-Oriented Functional Data Language

Michael V. Mannino, In Jin Choi, and Don S. Batory

†Department of Management Science & Information Systems
‡Department of Computer Science
The University of Texas at Austin, Austin, TX 78712

ABSTRACT

The Object Oriented Functional Data Language (O'FDL) is an interactive, strongly typed database programming language. It features inheritance and encapsulation of object-oriented languages and a functional notation for message expressions. The main contributions of the O'FDL are: 1) Functions of indefinite nesting levels that simplify nested list processing and form the basis of an efficient computation model; 2) Path expressions that allow a flexible, graphical correspondence for database queries; 3) A type system featuring parametric and inclusion polymorphism, function restrictions, and type inferencing; and 4) System-defined functions that provide a concise and convenient notation for filtering, aggregating, and combining objects. We demonstrate the versatility of the O'FDL for both relationally-complete expressions and more general computations.

1. INTRODUCTION

Database technology is being applied to many applications that are very different from the debit-credit transactions of business-oriented databases. Computer-aided design, expert databases, statistical databases, and text databases are recent examples. These emerging applications require support of hierarchically structured objects, new data types, and more powerful manipulation operators. Most existing database systems do not adequately support these features in terms of expressive power and efficiency.

Recently, the functional and object-oriented paradigms have been proposed as ways to extend database systems to meet the challenges of these emerging applications. The definitive property of functional programming is the principle of referential transparency [Hughes 1984] which means that an expression's only effect is to compute its value. Expressions in functional programming languages have no assignment statements and hence, implicit flow of control. This regime leads to numerous benefits including concise programs, modularity, program verification, type inferencing, and system controlled parallel execution. The definitive properties of object-oriented programming [Cox 1986] are encapsulation and inheritance. Encapsulation hides all the details of an object except the messages to which the object responds. Inheritance permits objects to be organized into taxonomies in which more specialized objects inherit the data and functions of more generalized objects. These principles lead to numerous benefits such as reduced software maintenance, programming by similarity, and increased software reusability.

Despite the advantages of these approaches, combining them into one language suitable for both database and general computation is difficult because they conflict in several ways and lack database features. Traditional object-oriented languages such as Smalltalk (Goldberg and Robson 1983) rely on side effects and have loose typing. Traditional functional languages such as Miranda [Turner 1985] are strongly typed but do not support a hierarchy of types, nor inheritance. From a database perspective, neither provides general purpose functions to filter, group, aggregate, combine, and update objects as expected by database users.

Because of the potential benefits of combining these approaches, we designed the Object-Oriented Functional Data Language (O'FDL). In general, our goal was a uniform and concise language for both database and general computation that preserves the major benefits of both approaches. We combined the object-oriented and functional programming paradigms by adapting the discipline and combining forms of functional programming into a framework with classes, inheritance, and message expressions. To support database manipulations, we added 1) functions with indefinite nesting levels that simplify nested list processing and form the basis of an efficient computation model; 2) convenient system-defined functions to filter, aggregate, combine, and update objects; and 3) path expressions that provide a flexible, graphical correspondence for database queries. This paper provides an informal introduction to the O'FDL. A complete description of the language including its formal grammar is given by Mannino, Choi, and Batory [1988].

The O'FDL evolved from work on the GENESIS Database Management System [Batory et al. 1986] and the GENESIS Data Language (GDL) [Batory, Leung, and Wise 1988]. The GDL was designed as an improvement over existing functional query languages in terms of notation and implementation techniques. The O'FDL formalizes and revises the GDL and adds other features to support the object-oriented and functional programming paradigms. These features include user-defined functions, the type system, and referential transparency. However, the O'FDL is still compatible with the underlying implementation concepts of the GDL.

The outline of this paper is as follows. Section 2 summarizes the features of the O'FDL. Section 3 describes salient aspects of the basic types, constructed types, and type inferencing. Section 4 presents the user-defined, data-defined, and system-defined functions including examples of their use. Section 6 briefly compares the O'FDL to other recent functional and object-oriented database languages. Section 7 concludes the work.

2. OVERVIEW OF THE O'FDL

The O'FDL was designed with two major goals: 1) to combine the object-oriented and functional programming paradigms 2) to provide a convenient and concise notation for both database and general purpose programming tasks. The features that support the first goal are:

(1) The O'FDL is an object-oriented language with support for classes, inheritance, and message expressions (i.e., sending a message to an object). In the body of an expression, the O'FDL uses the combining forms and the type discipline of functional programming.

(2) The O'FDL type system is based on the Fun type language [Cardelli and Wegner 1985] and the Milner type algorithm [Milner 1978]. Like Fun, the O'FDL supports parametric and
inclusion polymorphism. It extends Fun with function restrictions and incorporates them into the type checking algorithm. The O2FDL type algorithm can infer the types of expressions if they are well-typed or flags them as incorrectly typed.

3. The O2FDL is a higher order language in that functions are denotable values. Thus, a function can be passed as a value, embedded in the structure of a class, and returned as a value.

To accommodate both general purpose and database programming, the O2FDL features three different types of functions (user-defined, data-defined, and system-defined) plus a more flexible notation (path expressions) for expressing function compositions. In addition, the concept of referential transparency of functional programming was adapted for an environment with both persistent and non-persistent objects. The highlights of these aspects are:

1. User-defined functions are based on guarded recursive equations of functional programming languages. Equations can use pattern matching on the conditional part of an equation and recursion in the body.

2. Data-defined functions preserve the nesting level of their input sequence. They simplify database queries because increasingly nested sequences are not generated, and they have an efficient implementation as demonstrated by Batory, Leung, and Wise [1988]. Nested sequences in earlier functional query languages posed problems from both a notational and an implementation perspective.

3. The O2FDL includes convenient system-defined functions for filtering, aggregating, and combining objects. In particular, the filter function WHERE permits traditional infix Boolean expressions and combining forms support traditional function composition as well as several types of joins.

Path expressions give the O2FDL a flexible, graphical interpretation for database expressions. Path expressions permit compositions to be given as any unambiguous path in a network of classes. The translation from path expressions into regular O2FDL expressions is handled by a preprocessor.

The O2FDL supports a strict form of referential transparency for temporary objects and a modified form for persistent objects. For expressions with only temporary objects, strict referential transparency holds even though the O2FDL permits assignment to temporary objects. Assignment is restricted to the single assignment rule [Peyton-Jones 1987]. For expressions with persistent objects, the referential transparency concept is limited to the parts of a transaction between data manipulation functions.

The dual goals of the O2FDL make its efficient implementation an important research question. Several techniques may contribute to an efficient implementation. The function processor of the GENESIS Data Language has been demonstrated by Batory et al. [1988] as a useful technique for data-defined and system-defined functions in a traditional single processor environment. In addition, they showed that traditional query optimization techniques can be applied to the query subset of the O2FDL. More generally, other techniques such as graph reduction for data flow languages [Ackerman 1982] and recursion strategies and rule analysis of logic database languages [Tsar and Zaniolo 1986] will be appropriate to larger classes of expressions in the O2FDL. At this point in our work, we rely on a small prototype written in Miranda [Turner 1985] as a vehicle to experiment with the O2FDL.

3. CLASS DEFINITION

Objects are instances of classes where all classes belong to a single network with OBJECT as the root. Every class has a collection of functions to which it can respond. Functions are distinguished by their implementation (stored versus computed), the type of object to which they apply (class versus instance), and their origin (direct or inherited). Data-defined functions return stored object values, while user-defined functions return values computed by expressions in the O2FDL. Class functions are sent to the class or the entire collection of objects of the class, and they return values that do not depend on the instances of the class. Instance functions are sent to an individual object of the class, and they return values that vary by the instance. Normally, a subclass inherits all functions of its superclass. The implementation of user-defined functions can be changed in the subclass to accommodate different. Implementations of data-defined functions cannot be changed because they are invisible to the class definer.

As an example of these concepts, consider the partial definition of the student class (Figure 1). Membership in the student class is by explicit creation. It can also be given as an expression in the O2FDL which returns members of its superclass. For example, gradstudent could be defined as the collection of students with an enrollment in the graduate college. The domain of functions is implicit. For class functions, the domain is the class (data-defined functions) or a sequence of instances (user-defined functions). For instance functions, the domain is always an instance of the class.

Cardinality only applies to data-defined functions. It can be an integer indicating an exact cardinality, an integer followed by a plus sign indicating at least or a minus sign indicating at most, or a letter indicating an indeterminate cardinality. So 1+ means one or more and 1- means one or less. Existence dependencies are stated by using an integer greater than zero in the cardinality. Inverse functions always exist for data-defined instance functions. They have an implicit name which is formed by appending an apostrophe to the end of the function name.

Classes can be presented as a schema diagram (Figure 2) in which data-defined instance functions (arrows) connect classes. The arrows are shown in one direction here to avoid clutter although inverse mappings always exist. A double-headed arrow indicates a multivalued function, e.g., the function takes is multivalued from student to enrollment. The isa function is a built-in data-defined function that denotes a subclass relationship. The functions can also be summarized (Table 1) as a faster guide to a database than the complete definitions and as a more precise reference than the schema diagram. Note in Table 1, the function enroll_key which maps from an enrollment to a unique pair of student and offering. Unique functions can be defined in this way for classes which have composite keys.

<table>
<thead>
<tr>
<th>CLASS student</th>
<th>SUPERTYPE: OBJECT</th>
<th>MEMBERSHIP: EXPPLICIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLASS FUNCTIONS:</td>
<td>DATA-DEFINED:</td>
<td># define class components</td>
</tr>
<tr>
<td></td>
<td># define instance components</td>
<td></td>
</tr>
<tr>
<td>USER-DEFINED:</td>
<td># define class components</td>
<td></td>
</tr>
<tr>
<td></td>
<td># class is domain; returns all instances</td>
<td></td>
</tr>
<tr>
<td>INSTANCE FUNCTIONS:</td>
<td>DATA-DEFINED:</td>
<td># defines instance components</td>
</tr>
<tr>
<td></td>
<td># isa function</td>
<td></td>
</tr>
<tr>
<td>USER-DEFINED:</td>
<td># name</td>
<td></td>
</tr>
<tr>
<td></td>
<td># isa function</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1: Partial Definition of the Student Class
4. TYPE SYSTEM

In the OFDL, there is a strict separation between types and classes. The network of classes is a conceptual classification of real-world objects that can reduce the effort to understand a system and simplify the task of extending and using a system. A type is a collection of values manipulated by a program. When an expression is written, the type system ensures that the values denoted by expression components are compatible. The separation between classes and types promotes independence between computer manipulated values and real world objects and supports efficient reasoning about inheritance, correctness of expressions, and encapsulation. Class users can manipulate objects without knowing the principles of the type system or the difference between types and classes. The type system is responsible for mapping between classes and types and making inferences about types.

The OFDL type system shares characteristics of the Fun type language [Cardelli and Wegner 1985], but we have developed our own notations and type checker. The Fun language, which is based on the typed lambda calculus [Hindley 1969], was proposed as a framework for strongly typed object-oriented languages. Like Fun, the OFDL type system consists of a set of basic types, type constructors, and type inference rules. The basic types are INT (integer), FLOAT (floating point number), BOOL (boolean), CHAR (character), and STR (string). Type constructors are "->" (function), "[" (sequence), "<" (tuple), and "{}" (labeled record). We use ":=" to denote the type of an object. For example, 5 : INT, 'a' : CHAR, [1,2,3] : INT (sequence of integer), <3.5,True,'abc'> (3-tuple of FLOAT, BOOL, and STR). A sequence is an ordered collection of homogeneous objects. A tuple is a finite ordered collection of non-homogeneous objects.

In the following sections, we introduce labeled record and function constructors in more detail along with a discussion of type inferencing. A more detailed and formal presentation is given by Mannino and Choi [1988].

4.1. Labeled Record and Inheritance

In OFDL, database objects and/or composite objects are represented as labeled records; instance variables are attributes of the record which correspond to that class. Consider the representation of class student of the university database using labeled record type:

```
student :: {ssn::INT, name::STR, ..., class::STR}
```

Labeled record type is similar to the record type of other languages such as Pascal. An important characteristic of labeled record is that it provides a basis for inheritance among composite objects: a record type with the same attributes and additional ones as another record type is a subtype of the latter and the former will inherit all the properties of the latter. In the class hierarchy, it means that the former is a subclass of the latter. For example, class "undstudent" with additional instance variable "advisor" is a subclass of class "student".

```
undstudent :: {ssn::INT, name::STR, ..., class::STR, advisor::STR}
```

This is intuitively acceptable since a subclass represents a specialization with additional properties (attributes) over its superclass. The reader is referred to [Cardelli and Wegner 1985] for detailed treatment of labeled record types.

4.2. Function Type and Polymorphism

The OFDL is a higher order language in that functions are denotable values. Thus, a function can be passed as a value, embedded as a data structure, and returned as a value. The type of a function is either declared by the user or it is inferred by the system. Furthermore, the system will signal a type error if the declaration given by the user is not consistent with system-inferred type (i.e., type checking). Type declaration of a function consists of a domain, an optional argument list, a range, and an optional restriction organized in the following manner where the square brackets indicate optional elements:
The domain is the type of the principle object of the function because an expression is interpreted as sending a message to a domain object. The message includes the function name and the instantiated members of the argument list.

The domain, range, and each member of the argument list are type expressions - that is basic types or constructed types. A basic type consists of a type name optionally followed by a type variable. The type name restricts the most general type, while the type variable restricts the compatibility among the instantiated types of the domain, range, and arguments. When a function is analyzed as part of a message, a type error occurs if the type variable is not instantiated to the most general type or a descendant of it. For example, if F is a function from type C1 to C2, we declare its type as:

\[ F :: C_1 \rightarrow C_2 \]

C1 is the most general type to which F can be applied and C2 is the most general type returned by an application of F. In this case, the type variables indicate no restrictions because they are different. Since this is a frequent occurrence, the type variables are necessary only when they restrict the instantiated types.

As realistic examples of type declaration, consider the following:

1. COUNT :: [OBJECT] \rightarrow \text{INTEGER};
2. ++ :: ([OBJECT x]) [OBJECT x] \rightarrow [OBJECT x]; \# concatenate

Table 1: Data-Defined Functions of the University Database

<table>
<thead>
<tr>
<th>NAME</th>
<th>DOMAIN</th>
<th>RANGE</th>
<th>DOM→RANGE</th>
<th>RANGE→DOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>nameof</td>
<td>student</td>
<td>name</td>
<td>1</td>
<td>n</td>
</tr>
<tr>
<td>ssnof</td>
<td>student</td>
<td>ssn</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>addrif</td>
<td>student</td>
<td>address</td>
<td>1-</td>
<td>n</td>
</tr>
<tr>
<td>ggapof</td>
<td>student</td>
<td>gpa</td>
<td>1-</td>
<td>n</td>
</tr>
<tr>
<td>classof</td>
<td>student</td>
<td>class</td>
<td>1</td>
<td>n</td>
</tr>
<tr>
<td>majorof</td>
<td>student</td>
<td>major</td>
<td>1-</td>
<td>n</td>
</tr>
<tr>
<td>takes</td>
<td>student</td>
<td>enrollment</td>
<td>n</td>
<td>1</td>
</tr>
<tr>
<td>adv_of</td>
<td>undstudent</td>
<td>advisor</td>
<td>n</td>
<td>1</td>
</tr>
<tr>
<td>thesis_of</td>
<td>gradstudent</td>
<td>thesis</td>
<td>n</td>
<td>1</td>
</tr>
<tr>
<td>nameof</td>
<td>faculty</td>
<td>name</td>
<td>1</td>
<td>n</td>
</tr>
<tr>
<td>ssnof</td>
<td>faculty</td>
<td>ssn</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>deptof</td>
<td>faculty</td>
<td>dept</td>
<td>n</td>
<td>m</td>
</tr>
<tr>
<td>salaryof</td>
<td>faculty</td>
<td>salary</td>
<td>l</td>
<td>n</td>
</tr>
<tr>
<td>rankof</td>
<td>faculty</td>
<td>rank</td>
<td>l</td>
<td>n</td>
</tr>
<tr>
<td>suprof</td>
<td>faculty</td>
<td>faculty</td>
<td>1-</td>
<td></td>
</tr>
<tr>
<td>teaches</td>
<td>faculty</td>
<td>offering</td>
<td>n</td>
<td>1</td>
</tr>
<tr>
<td>gradeof</td>
<td>enrollment</td>
<td>grade</td>
<td>1-</td>
<td>n</td>
</tr>
<tr>
<td>enroll_key</td>
<td>enrollment</td>
<td>&lt;student, offering&gt;</td>
<td>l</td>
<td>1</td>
</tr>
<tr>
<td>courseof</td>
<td>course</td>
<td>courseo</td>
<td>l</td>
<td>n</td>
</tr>
<tr>
<td>descof</td>
<td>course</td>
<td>desc</td>
<td>l</td>
<td>n</td>
</tr>
<tr>
<td>offerings_of</td>
<td>course</td>
<td>offering</td>
<td>n</td>
<td>l</td>
</tr>
<tr>
<td>prereq</td>
<td>course</td>
<td>course</td>
<td>n</td>
<td>m</td>
</tr>
<tr>
<td>uniqueofferings_of</td>
<td>offering</td>
<td>uniqueno</td>
<td>l</td>
<td>1</td>
</tr>
<tr>
<td>semsof</td>
<td>offering</td>
<td>sems</td>
<td>l</td>
<td>n</td>
</tr>
<tr>
<td>yearof</td>
<td>offering</td>
<td>year</td>
<td>l</td>
<td>n</td>
</tr>
<tr>
<td>roomof</td>
<td>offering</td>
<td>room</td>
<td>l</td>
<td>n</td>
</tr>
<tr>
<td>timeof</td>
<td>offering</td>
<td>time</td>
<td>l</td>
<td>n</td>
</tr>
<tr>
<td>studentsof</td>
<td>offering</td>
<td>enrollment</td>
<td>n</td>
<td>1</td>
</tr>
</tbody>
</table>

The domain is the type of the principle object of the function because an expression is interpreted as sending a message to a domain object. The message includes the function name and the instantiated members of the argument list.

\[ F :: ([\text{arg}\_\text{list}]) \Rightarrow \text{<domain>} \rightarrow \text{<range>} \Rightarrow \text{<type-restr>}; \]

The domain, range, and each member of the argument list are type expressions - that is basic types or constructed types. A basic type consists of a type name optionally followed by a type variable. The type name restricts the most general type, while the type variable restricts the compatibility among the instantiated types of the domain, range, and arguments. When a function is analyzed as part of a message, a type error occurs if the type variable is not instantiated to the most general type or a descendant of it. For example, if F is a function from type C1 to C2, we declare its type as:

\[ F :: C_1 \rightarrow C_2 \]

As realistic examples of type declaration, consider the following:

1. COUNT :: [OBJECT] \rightarrow \text{INTEGER};
2. ++ :: ([OBJECT x]) [OBJECT x] \rightarrow [OBJECT x]; \# concatenate

The above type declaration indicates that function "takes" can also assume any subclass (subtype) of class "student" as an argument. O^2FDL captures both parametric and inclusion polymor-
phism using the same notation whereas Fun uses universal and bounded quantification, respectively.

We have added the optional restriction of a function as a technique to further restrict compatibility because the use of type variables alone may not be sufficient. In some cases, it may be too strict to limit the instantiated types to be the same or it may be too loose to let them be different. Function restrictions provide an additional layer to control the consistency among a function’s arguments. Consider the following example:

\[ \text{4. } = :: (\text{OBJECT } x) [\text{OBJECT } y] \rightarrow [\text{BOOLEAN}] \]

\[ \text{RESTRICTED BY } (x = y) \text{ OR (x DESCENDANT } y) \text{ OR (y DESCENDANT } x) \]

The equality function can compare objects when they have the same instantiated type or when one is a descendant of the other. So, comparing a subtype of integer to an integer is not a type violation, but comparing an integer to a character string is a violation.

To ensure correctness, we require that the type expression of a inherited function is at least as strict as the type expression in the function definition of its supertype object. For ease of type checking, stricter type expressions are given by conjoining additional terms to those of its supertype.

4.3. Type Inference

As mentioned above, in the O’FDL a user does not have to specify the types of objects defined by expressions. The O’FDL type system can infer the types, if the expressions are well-typed. Thus, the users receive the obvious benefit of detecting errors as well as the more subtle benefit of sometimes inferring a more general type than realized by the user. Type inferring reduces to type checking when the types of all subexpressions are known. The O’FDL type algorithm can infer the type of an expression as well as check whether a user-supplied type expression is equal to the inferred one.

Type inferring in O’FDL is based on the Milner [1978] type algorithm (also see [Peyton-Jones 1987]). It was originally proposed by Curry and Feys [1958] and rediscovered by Hindley [1969] and Milner. O’FDL extends Milner’s technique for type variables, most generalized types, and restrictions. This increases the complexity of the type checker, but it provides for safer expressions. The extension of type inferring in the O’FDL is similar to that made in Fun [Cardelli and Wegner 1985]. A formal treatment of type inferring including the underlying inference rules is described by Mazzino and Choi [1988].

5. O’FDL FUNCTIONS

The O’FDL includes two kinds of computed functions (user-defined and system-defined) and one type of stored function (data-defined) which are primarily distinguished by their implementation methods. User-defined functions are defined by expressions in the O’FDL. Data-defined functions are evaluated by searching the internal structure of object collections, and their implementation is hidden from class definers. Data-defined functions are further distinguished because the values they return (either class or instance components) are modifiable. System-defined functions are computed functions whose implementation is hidden from class definers. They are used for primitive operations such as adding two numbers and for common database operations such as filtering, aggregating, and updating objects. In this section, we present the general features of O’FDL functions along with examples of their use.

5.1. User-Defined Functions

User-defined functions include a public part known as the type declaration and a private part known as the implementation. The type declaration restricts the objects to which a function can be applied. Since the type declaration can be inferred as discussed in the last section, it is not necessary for the user to explicitly state it. The implementation is based on guarded recursive equations of functional programming languages especially Miranda [Turner 1985]. Functions of arbitrary complexity can be defined in the implementation through the use of pattern matching and recursion. Due to the principal of encapsulation, the implementation is hidden from users of the function. However, class definers can access the implementation of user-defined functions, while they cannot access the implementation of system- and data-defined functions.

An implementation of a user-defined function is a collection of rules of the following forms:

\[ \text{<head> } = \text{ <body>; or } \]

\[ \text{<head> & <guard> } = \text{ <body> } \{ & <guard> = \text{ <body>), } \}

\[ & <guard> ] = \text{ <body>; } \]

where the head specifies a pattern for the function’s domain and argument list, the guard specifies a Boolean expression, and the body specifies a message expression that is performed if the rule’s head is matched and the guard is true. In the second form, a list of (guard, body) pairs can be given for one head. The final guard can be omitted when the last body is the default.

The head of a rule has the form \( \beta \cdot F \cdot \alpha \), where \( F \) is the name of a user-defined function, \( \beta \) is the domain, and \( \alpha \) is the optional argument list. \( \beta \) and each element of \( \alpha \) specify pattern expressions. The simplest pattern expressions are constants and variables. These simple forms can be augmented with arithmetic operators for numeric objects and sequence constructors. Curried definitions can be given by omitting the domain and argument list. This is especially useful for defining specialized higher-order functions.

The collection of a function’s rules specify cases which may be overlapping. If not disjoint, the rules should be listed in the order of most restrictive to least restrictive because they are matched in the order defined. Typically, the empty sequence is the most restrictive and is therefore listed first. Recent research on rule analysis in logic database languages [Tsar and Zaniolo 1986] may allow relaxation of this dependency on the order of rules.

The body of a rule defines a message expression that provides the mapping of the function possibility in terms of variables used in the patterns of \( \beta \cdot \alpha \). A message expression is the application of a function to its domain and other arguments. For example, \( F: (A) D \rightarrow R \) is a function with domain \( D \), argument \( A \), and range \( R \). Application of the function can be done as \( F \cdot A \) where \( A \) is a member of \( D \) and \( F \) is a member of \( A \). This expression can be interpreted as sending the message \( F \cdot a \) to the object \( d \).

As in other functional languages, O’FDL expressions can use composition and recursion. Functions may be combined by composition. The expression \( (x \cdot F) \cdot G \) represents the application of \( F \) to the object \( x \) followed by the application of \( G \) to the output of \( F \). To reduce the need for expressions with nested parentheses, we use the dot notation. The previous expression is equivalent to \( x \cdot F \cdot G \) where the dot represents function composition. An O’FDL expression can make a recursive reference to itself, either directly or indirectly. Recursion can be useful for list manipulations and querying databases with graph structured instances.

---

1 The name is due to Curry [1958] who formalized this form of abbreviated function. In a Curried function, missing arguments are inferred from the context of the definition and other function definitions.
We depict these concepts in the following examples. The aggregate functions \( \text{COUNT1} \) and \( \text{SUM1} \) can be defined as:

\[
\begin{align*}
\text{COUNT1} & : (\text{OBJECT}) \rightarrow \text{INT}; \\
& \begin{cases} 
\text{COUNT1} \mapsto 0; \\
(h: T) \mapsto 1 + (T \text{COUNT1}); 
\end{cases} \\
\text{SUM1} & : (\text{NUMBER}) \rightarrow \text{FLOAT}; \\
& \begin{cases} 
\text{SUM1} \mapsto 0; \\
(h: T) \mapsto h + (T \text{SUM1}); 
\end{cases}
\end{align*}
\]

The first rule of \( \text{COUNT1} \) means that the number of elements in an empty sequence is 0. The second rule means that the number of elements in a nonempty sequence is one more than the number of elements in the tail of the sequence. The parentheses around \( T \text{COUNT1} \) can be omitted because regular function application has higher precedence than the + function.

The \( \text{AVE1} \) function uses both \( \text{COUNT1} \) and \( \text{SUM1} \) to define the arguments for the \( / \) function:

\[
\text{AVE1} : (\text{NUMBER}) \rightarrow \text{FLOAT}; \\
& \begin{cases} 
\text{AVE1} \mapsto ; \\
\text{d} \mapsto \text{d SUM1} / \text{d COUNT1}; 
\end{cases}
\]

The first rule handles the average of an empty sequence. For nonempty sequences, \( \text{AVE1} \) is computed as the sequence applied to the functions \( \text{SUM1} \) and \( \text{COUNT1} \).

To apply these functions to a doubly nested sequence, they must be mapped as in:

\[
\begin{align*}
[[1, 2, 3, 4], [5, 6, 7]] & \mapsto \text{MAP AVE1}; \\
\text{MAP} & \text{ is a higher order function that applies its argument (a function) to each member of its domain (a sequence). In Section 5.4.3, system-defined versions of the aggregate functions are described which do not require the MAP function to handle nested sequences.}
\end{align*}
\]

Guards can be used to further restrict the application of a function. A classic example of a function requiring guards is the greatest common divisor using Euclid's algorithm:

\[
\text{GCD} : \text{INT} \rightarrow \text{INT}; \\
& \begin{cases} 
\text{GCD} b & \text{ if } a > b \Rightarrow (a-b) \text{GCD} b, \\
& \text{ if } a < b \Rightarrow (a-b) \text{GCD} b, \\
& \text{ if } a = b \Rightarrow a.
\end{cases}
\]

In general, pattern matching and guards are not mutually exclusive. Guards are more general than patterns (i.e., every pattern can be expressed as a guard), but patterns are often easier to write and read.

We have seen from previous examples that user-defined functions can have recursive implementations. This is particularly useful in querying databases that have graph-structured extensions. As a simple example, consider the all_prereq function which produces the prerequisites for a sequence of courses.

\[
\begin{align*}
\text{all_prereq} & : \text{[course]} \rightarrow \text{[course]}; \\
& \begin{cases} 
\text{all_prereq} \mapsto []; \\
(h: T) \mapsto \text{all_prereq} \mapsto (\text{prereq}) \mapsto \text{all_prereq} \mapsto (T \text{all_prereq});
\end{cases}
\end{align*}
\]

In the second rule, \( \text{all_prereq} \) produces the immediate prerequisites of \( h \), which is used as the input to \( \text{all_prereq} \) in the second term. In the third term, \( T \text{all_prereq} \) recursively processes the remainder of the list.

### 5.2. Data-Defined Functions

Data-defined functions are distinguished by indefinite nesting levels, the generating function, and path expressions. Indefinite nesting levels are incorporated into the general form of data-defined functions which is

\[
\begin{align*}
\text{F} & : (\text{OBJECT})^n \rightarrow (\text{OBJECT})^m \text{ RESTRICTED}_n \text{ BY } n \geq 0;
\end{align*}
\]

The superscripted variable denotes an indefinite nesting level. In effect, data-defined functions are defined from a sequence (possibly nested) to a sequence (possibly nested). For example, if \( a_1 \) maps to \( b_1, b_2 \) and \( a_2 \) maps to \( b_3, b_4 \), then

\[
\begin{align*}
[a_1, a_2] & \mapsto \{b_1, b_2, b_3, b_4\} \\
\{a_1, a_2\} & \mapsto \{b_1, b_2, b_3, b_4\} \text{ II denotes evaluates to}
\end{align*}
\]

Indefinite nesting levels offer two advantages. First, expressions involving data-defined functions are simplified because increasingly nested sequences are not generated. Second, as demonstrated by Batory, Leung, and Wise [1988], functions with indefinite nesting levels have an efficient, iterative implementation. This is in contrast to the recursive processing of lists in functional programming languages which can be inefficient for the long and deeply-nested lists which would be generated by a traditional functional approach.

The second distinguishing element of data-defined functions is the generating function which is a system-defined function that materializes a class. The function \( \text{F} : \text{CLASS} \rightarrow (\text{OBJECT}) \) returns the sequence of stored objects of the class. Each class has a class function with the name of "!". If \( A \) is a class name, \( F : \{A^k \rightarrow [B]^m \} \) and \( G : \{B^k \rightarrow [C]^n \} \), the expression \( A!F.G \) returns a list of \( C \) objects.

Typically, expressions to query a database begin with a generating function. As examples from the university database, consider the expressions:

1. \( \text{student? .enrolled_in.grade_of} \);
2. \( \text{offering! .teaches? .name_of} \);

Query 1 returns the list of grades obtained by any student. Query 2 returns the list of names of instructors teaching any course offering.

The function \( \text{teaches?} \) is the inverse of teaches as the prime (') denotes the inverse of a function.

Constructing a query in this manner may be a little difficult because users must know the names and types of intermediate functions. To alleviate this problem, we introduce the path expression which gives O'FDL expressions a more flexible graphical interpretation. A path expression is any unambiguous path in a schema diagram. A path can be given in the traditional manner as a start node followed by a collection of connected edges. Subpaths can also be given as a sequence of connected classes when no ambiguity results. In cases where two classes are connected by more than one function, the function name must be used. Using a path expression, queries 1 and 2 can be rewritten as:

1. \( \text{student? .enrolled_in.grade_of} \);
2. \( \text{offering? .faculty.name} \);

Path expressions are not message expressions. They are converted into message expressions by a preprocessor which translates the first class name into a generating function and substitutes function names for following class names. Translation rules for path expressions including their use in the system-defined functions described in the next section can be found in the complete description of O'FDL [Maninno, Choi, and Batory 1988].

23
5.3. System-Defined Functions

In this subsection, we highlight the system-defined functions which accommodate the database query user. These functions were carefully designed so that they lead to concise, easy to read expressions. Most notably, like data-defined functions, system-defined functions can be applied to sequences with indefinite nesting levels. The O^FDL includes a filter function that allows traditional infix appearance of Boolean expressions and alternative combining forms such as the join to accommodate users more familiar with relational languages. The O^FDL also provides data manipulation functions to insert and delete persistent objects. Their description can be found in the complete definition of the O^FDL. [McNinn, Choi, and Batory 1988].

5.3.1. The Tuple Function

The system-defined tuple function generates a sequence of tuples by applying a tuple of message expressions (F_i) to a sequence of objects. It is declared as:

TUPLE: \( \langle F_1, F_2, \ldots, F_m \rangle \rightarrow [\text{OBJECT}]^n \)

The range of the tuple function is a sequence of tuples in which the nesting level of the domain is preserved. If D is a sequence of objects and \( \langle F_1, \ldots, F_m \rangle \) is a tuple of functions, the expression \( D \cdot \langle F_1, \ldots, F_m \rangle \) produces a sequence of n-tuples of the form \( \langle F_1(d), \ldots, F_m(d) \rangle \) for each object d in D. As an example, consider an extension to Query 2a which produces both faculty and student tuples by applying a tuple of message expressions (F_i) to a sequence of objects. It is declared as:

\[ \text{TUPLE: } (\text{TUPLE}) \rightarrow \text{[OBJECT]} \]

To demonstrate the use of the tuple function, nested sequences can be easily generated as in Query 3a.

3a. offering!.teaches!.TUPLE <name, address>;

3b. faculty!.teaches!.TUPLE <name, address>;


5.3.2. The WHERE Function

Unwanted objects can be eliminated from a sequence by the system-defined WHERE function. It takes a Boolean function, applies it to a sequence, and returns a subsequence whose elements satisfy the Boolean function. If no element of the sequence satisfies the predicate, the resulting subsequence will be empty. The type of WHERE is:

WHERE :: (BEXPR) [OBJECT x]^n \rightarrow [OBJECT x]^n;

As an example, consider the following O^FDL message expressions where EVEN is a Boolean function which returns True if the argument is even and (>4) is higher-order function which returns True if the argument is greater than 4.

\[ [1,2,3,4] \text{ WHERE(EVEN) = [2,4]} \]
\[ [1,2,3,4] \text{ WHERE(>4) = [4]} \]

The argument of the WHERE function can be an arbitrary expression which returns a Boolean value. To generalize from the case of a simple unary Boolean operator, we need to indicate how objects of the domain are applied to elements of the Boolean expression argument. When we consider combinations of Boolean expressions using AND and OR, and non-unary Boolean functions, we need to indicate which elements of the Boolean expression should be applied to the domain objects. The symbol '$' indicates the application of the domain to the following function. It serves the same purpose as the 'self' variable in Smalltalk [Goldberg and Robson 1983]. Consider the following query:

6a. student!.WHERE(grade > 3.5).ssno;

The $ symbol is also convenient for expressions with Boolean connectives and distribution of the domain to both sides of a Boolean function. These types of expressions are frequently used when querying databases. Consider the following two examples:

7a. student!.WHERE(grade > 3.5 AND classof = 'FR').nameof;
7b. student!.WHERE(grade > 3.5).classof = 'FR'.nameof;

8a. faculty!.WHERE(rank = 'ASST' AND salary > suprofsalary).nameof;
8b. faculty!.WHERE($rank = 'ASST' AND $salary > suprofsalary).nameof;

Query 7 lists the names of freshmen with GPA higher than 3.5. Query 9 demonstrates nesting of the WHERE function to select faculty teaching offerings in Fall 1987.

9a. WHERE(AND ($rank = 'ASST' AND $salary > suprofsalary).nameof;
9b. WHERE($rank = 'ASST' AND $salary > suprofsalary).nameof;

Like the tuple function, the WHERE function can be nested and path expressions can be converted inside the WHERE argument. To demonstrate the latter, Query 8b is equivalent to Query 9. Query 9 demonstrates nesting of the WHERE function to select faculty teaching offerings in Fall 1987.

5.3.3. Aggregation Functions

The O^FDL features the standard collection of aggregation operators as system-defined functions: COUNT, SUM, AVE, MAX, and MIN. They are system-defined to increase their efficiency and to utilize indefinite nesting levels. Each standard aggregation function reduces the nesting level by one. For example,
COUNT has the form:

\[
\text{COUNT} : \{\text{OBJECT}\}^n \rightarrow \{\text{INTEGER}\}^n;
\]

The other functions have similar definitions except that their domain and ranges differ. The meaning and usage of the aggregate functions is suggested by their names. To depict the usage of these functions, we provide Queries 10 and 11.

10. faculty.<name, dept.faculty.salary.AVE>;
11. offering.WHERE($enrollment.COUNT) =
   (offering.NEST studentsof.COUNT.MIN)).unique

Query 10 lists the name of each faculty member and the average salary of the faculty in his or her department. Notice that for each faculty f, a tuple <f.name, f.dept.faculty.salary.AVE> is generated. (f.dept.faculty) will return all the faculty members of the department that f belongs to, and hence (f.dept.faculty.salary.AVE) is the average salary of its faculty members. This query also illustrates the usefulness of the path expression; without it, the above query should have been

\[
\text{faculty!.<name_of, dept.of.dept.of_salary.of.AVE>}
\]

Query 11 lists the unique object of the course offering with smallest number of students. Note how offering! is used inside the WHERE function. Since the expression does not start with `$`, it is disconnected with the offering of the outside. Second, in order to compute the offering with the minimum number of students, the NEST function was necessary to add a layer of nesting to the result of offering!. The functions following NEST correctly process the doubly nested result without a MAP function because they are defined on indefinite nesting levels.

5.3.4. Cartesian Product and Alternative Combining Forms

In database applications, there is sometimes a need to explore relationships that are not explicitly represented in the database. It is always possible to define a new function and use it in a composition, but this requires knowledge about the details of function creation. As a more convenient solution, the O'FDL features the $\times$ function (cross product) which maps two sequences into the sequence of pairs which represent all possible combinations of the two input sequences. Its type is defined as:

\[
\times: \{\text{OBJECT a}\}^n \rightarrow \{\text{OBJECT a, OBJECT b}\}^n
\]

The $\times$ function is convenient in a query such as retrieving the pairs of students who have the same address:

\[
\text{student!} \times \text{student!}.\text{WHERE}($\$\text{first.addrof} = \$\text{second.addrof})
\]

Here the functions first and second return the first and second components from the pairs in the sequence. This type of referencing may inconvenient. Hence, the O'FDL permits a symbolic naming capability called a reference variable list. The O'FDL preprocessor translates reference variable lists into proper message expressions as part of the analysis of path expressions. Using a reference variable list, the previous expression can be rewritten as:

\[
\text{student!} \times \text{student!}.@<s1, s2>.\text{WHERE}($\$\text{s1.addrof} = \$\text{s2.addrof})
\]

Composition is the minimal way to combine functions and usually the most convenient. To accommodate users of relational database systems, the O'FDL supports three alternative combining forms: JOIN, OUTER, and OS_OUTER which correspond to the natural join, outer-join, and one-sided of Relational Algebra. JOIN takes two compatible functions and generates a new function which produces a sequence of normalized tuples in which the first component is a result object from the first function and the second component is a result object from the second function. JOIN has identical behavior to composition except that it preserves the output of its first function. The outer-join functions, which are based on [Rosenthal and Reiner 1984], preserve the objects which do not map to an object in the range of a second function. For the non-matched domain objects, the output contains a tuple with the object itself and a null value for the range object. The non-matched range objects are handled similarly. The one sided outer-join (OS_OUTER) corresponds to a left outer-join as the non-matched range objects are not preserved in the output. A right outer-join can be performed by reversing the positions of the input functions. More details about the alternative combining forms can be found in [Mannino, Choi, and Batory 1988].

6. COMPARISON TO OTHER LANGUAGES

As a database programming language, the O'FDL is related to a number of recent languages on the basis of its type system and programming style. The type system of the O'FDL is based on the Milner algorithm [1978] for inferring the type of an expression. This algorithm is incorporated in functional programming languages such as ML [Milner 1983] and Miranda [Turner 1985]. Recent database programming languages including Galileo [Albanese, Cardelli, and Orsini 1985], Vbase [Ontolog 1986], and Q2 [Lecluse et al. 1988], as well as the O'FDL are based on the Milner type algorithm. Galileo made extensions to the algorithm for inheritance in a class lattice. Q2 made revisions for inference of function types, but it has a relaxed strong typing for compatibility with programming languages that are not strongly typed. The O'FDL makes extensions for function restrictions. The O'FDL employs an applicative programming style derived from functional programming languages, especially Miranda [Turner 1985]. This is in contrast to the imperative style used in Galileo, Q2, Vbase, and Gemstone [Maier and Stein 1986]. The Probe Data Model (PDM) [Manola and Dayal 1986] is also based on a functional programming style but it differs in its use of normalization in function results and it was not designed as a complete programming language. The data-defined and system-defined functions of the O'FDL extend other functional query languages such as the Functional Query Language (FQL) [Buneman and Frankel 1979] and Daplex [Shipman 1981] in terms of nested sequences, conciseness, and readability. In particular, the O'FDL does not require multiple applications of the MAP function as in FQL, nor make arbitrary restrictions on nested sets as in Daplex, nor require explicit iteration as in Daplex. In addition, it uses an infix notation for Boolean expressions in contrast to the prefix notation used in FQL which is less familiar to users of traditional relational systems.

7. CONCLUSION

The Object-Oriented Functional Data Language (O'FDL) combines the object-oriented and functional programming paradigms for both database and general computations. It adopts inheritance and encapsulation from the object-oriented paradigm and uses a functional notation to express methods. The type system incorporates parametric and inclusion polymorphism, function restrictions, and type inferencing. Database expressions are supported through functions with indefinite nesting levels, path expressions, and convenient system-defined functions. Indefinite nesting level functions simplify nested list processing and form the basis of an efficient computational model. Path expressions provide a flexible, graphical counterpart for database queries. System-defined functions provide a concise and convenient notation for filtering, aggregating, and combining objects.
The definition of the O'FDL is a first step towards integrating the functional and object-oriented programming paradigms for conceptual modeling. We have demonstrated that it is possible to produce a simple, concise, and powerful language that preserves the major benefits of both approaches. Our current interest is in implementing a complete prototype and applying the language and type system to the domain of mathematical programming applications, especially to recent work on mathematical libraries [Mannino, Greenberg, and Hong 1988] and strongly typed algebraic languages [Bradley and Clemence 1988].

REFERENCES


26