Database Programming Languages: A Functional Approach

Jurgen Annevelink
H.P. Labs, Mailstop 3U-4, P.O. Box 10490, Palo Alto, 94303-0969

Abstract

This paper introduces the Iris Programming Language (IPL), a functional embedding of OSQL, the query language of Iris, an object-oriented database system prototype developed by Hewlett-Packard Laboratories. The main contribution of this paper is to show how a declarative query language can be extended into a general purpose database programming language by embedding it into a functional progamming language. The key to the approach is the capability of invoking an embedded interpreter from within the context of a query evaluation. The paper discusses the mechanism by which the Iris query interpreter was extended to provide this capability and discusses how such extensions can be added to relational database systems.

IPL is a full-fledged programming language, that includes control abstractions and is compiled into an intermediate, lambda calculus based language. IPL provides direct access to the operators of the underlying intermediate language, thus allowing it to be extended and specialized for a particular application domain.

IPL is succesfully implemented and is in active use.

1 Introduction

Many database system research projects address the need for database system extensibility [5, 6, 9, 10, 11, 12, 14]. This direction is motivated by the desire to make the database an integral component of an information system and to simplify both the development and maintenance of an ever growing set of application programs. In addition, new requirements on database systems, e.g. for the database to play an active role in a distributed information system [18], pose additional demands on extensibility.

Today, operations on database objects are mostly expressed in application code. This is due to the limited computational power of database languages, such as SQL. As a result, because different applications have to implement similar operations, redundant code is introduced, and a sufficient level of integrity is difficult to establish. In an extensible DBMS, application-specific operations can be added to the system.

The advantage of making what otherwise would be application code into operations stored in the database (together with the data) are many, including: (1) increased application programmer productivity, through reuse of code and an increased level of data-independence, (2) increased consistency of applications, by allowing database functions to enforce arbitrary consistency constraints and (3) increased performance, e.g. by compiling out redundant type checking and by eliminating the networking overhead.

The system we used for developing IPL was Iris [9], an object-oriented database system prototype developed at Hewlett-Packard Laboratories.

The rest of the paper is organized as follows. Section 2 discusses related research. Section 3 provides a brief overview of the Iris data model. The next section, Section 4, contains an introductory example, showing the main IPL constructs and their usage. Section 5 discusses the main issues underlying the design of the IPL language. Next, Section 6, discusses the implementation of IPL as well as the generalization to SQL based systems. Finally, Section 7 contains an extensive example, based on that discussed in [3]. Section 8 contains some concluding remarks.

2 Related Research

There are at least four major directions into database programming language research at this time. First there are the embedded language approaches, e.g. SQL embedded in C or COBOL. The embedded language approach is characterized by the fact that it uses a preprocessor that translates the embedded language into e.g. a C program. The C program includes calls to subroutines to encapsulate the interaction with the database (server). Translation to and from database specific storage structures is done in these interface routines. This approach is well established and is what is used for most database application development today. The second type of approach is the persistent language approach. The goal here is to make interaction with the database an integral part of the semantics of the program, typically by providing the capability to make programming language data structures capable of being stored and retrieved from the database. Most of these efforts have concentrated on either (1) providing persistent extensions to C++, requiring the introduction of new syntax, in effect creating a new language or (2) providing a base class that allows persistence to be added to another class by (multiple) inheritance. The third major area of research is that of functional languages. Examples in this class include [19, 9, 13]. The functional

datamodel is very well suited to model extensibility because of its basis in the lambda calculus [8]. The question of how to provide the kind of extensibility that allows the user/application developer to define and store arbitrary operations, thus allowing them to be shared across all applications accessing the database, is of critical importance for managing the complexity and consistency of developing large suites of applications and dealing with the inevitable change. The fourth major area of research is that of logic programming languages. Logic languages can be distinguished from functional languages by the fact that they use resolution as the computational paradigm. The resolution process will produce all the possible bindings of variables to values for a given goal. Functional programming on the other hand is based on a reduction or rewriting process, guided by the implicit directionality (distinction between inputs and outputs) of a functional definition. Functional and logic languages fall in the category of so-called declarative languages. As such, they share the advantages of a high-level of abstraction and freedom of detail, making them more amenable to formal analysis and optimization then other languages. Relational query languages [2], can be seen as a combination of logic and functional languages. The notion of a relation resembles that of a set of ground facts in that there is no directionality. The relational algebra operators that are applied to these relations resemble much more of a functional notion however, distinguishing as they do between inputs and outputs.

There are a number of (database) languages that have been based on the functional (applicative) paradigm. For example, FAD [4] is a functional language built on top of a lower-level layer of abstract datatypes. FAD operators, that are used for combining the lower-level datatype operations, were selected for their parallel implementation opportunities, e.g. operations like pump and filter. FAD provides extensibility on top of a base layer of abstract datatypes, by providing a calculus to combine (aggregation) atomic values into sets and tuples and then define functions and expressions to operate on these. The FAD language is different from IPL mainly in terms of the base datamodel (set of abstract datatypes vs. OSQL) and the set of predefined operators. IPL provides syntactic constructs only for a few operators and in that sense is more limited then FAD. At the same time however, IPL provides access to its lambda calculus based intermediate language, thus guaranteeing extensibility. IPL, contrary to FAD, uses a built-in embedded query language to provide the necessary (declarative) access to data stored in the database. Similarly, IPL provides access to the database systems update operator to change the state of the database. O2FDL [13] is a language similar to FAD in many aspects. It adds to FAD however in that it combines a notion of classes, used to characterize real-world entities and define the operations applicable to them, with that of a type-system.

In the area of logic database systems, IPL is most related to GLUE[17], a procedural language for deductive databases, designed to complement NAIL![15]. GLUE and IPL share the same design philosophy, in that they both extend a declarative language with procedural constructs. An important aspect of this is to avoid the so-called impedance mismatch problem, by allowing the two languages to share the same type system and set of datatypes.

Another approach to include arbitrary procedures, and thus provide a measure of extensibility in a database language is the POSTGRES system[20]. POSTGRES contains a mechanism for storing values of type procedure in an attribute of a relation. These procedures, however, are limited to sequences of retrieve statements. This capability has some similarity with the derived function mechanism implemented in Iris and is considerably less powerful then the IPL functions discussed in this paper.

3 The Iris Data Model

The Iris data model supports abstract data types. Its roots can be found in previous work on Daplex [19] and Taxis [16]. The Iris data model contains three important constructs: *objects*, *types* and *functions*. These are briefly described below. The Iris data model has recently been described elsewhere [9, 21].

The language developed in conjunction with the Iris database system is called OSQL. OSQL adds syntactic sugar and a notion of host-variables to the simple function call interface supported by the database system. The syntactic format of OSQL was developed to closely correspond to SQL. Since OSQL supports different kinds of abstractions, OSQL is not a superset of SQL.

3.1 Objects and Types

Objects in Iris represent entities and concepts from the application domain being modeled.

Types have unique names and are used to categorize objects into sets that are capable of participating in a specific set of functions. Objects serve as arguments to functions and may be returned as results of functions. A function may only be applied to objects that have the types required by the signature of the function. Types are organized in an acyclic type graph that models inheritance in Iris.

3.2 Functions

Attributes of objects, relationships among objects, and computations on objects are modeled by functions. Iris functions are defined over types, they may be multi-valued and, unlike mathematical functions, they may have side-effects. In Iris, the declaration of a function is separated from its implementation. This provides a degree of data independence. A new function is *declared* by specifying its name, the types of its argument and result parameters, and whether it is single or many

valued. Before a function may be invoked, an *implementation* must be specified.

The *implementation* or *body* of a function is a specification of its behavior. The implementation of the function is compiled into an internal format that is stored in the system catalog. When the function is later invoked, the compiled representation is retrieved and interpreted. It is important to note here that the compiled representation of the function can be very different from the original specification, based on the operators available to the run-time interpreter. Typical Iris functions are compiled into an (extended) relational algebra format and are optimized to reduce resource usage, e.g. by making use of indices when appropriate.

4 Examples of IPL functions

In this section we will begin to introduce the IPL language, and motivate its constructs by means of a series of examples. In general, the body of an IPL function consists of an expression; the function returns the value computed by the expression. The syntax of IPL was kept as close as possible to that of typical application languages such as C and Pascal.

The first example shows an IPL function body that consists of a sequence of statements, enclosed between `begin` and `end` keywords. It also introduces the use of conditional constructs. The function, `NameOfObject`, returns the name of an object, if a name function is defined for it, otherwise it returns the string "NoName". It is important to realize that IPL functions such as the ones shown here can be called from within OSQL queries [1], in a like manner that we can call OSQL derived or stored functions from within the body of an IPL function.

```
create function NameOfObject(Object o) ->
                                  String as ipl
begin
   declare f, n;
   f := ResolveFunctionByName("Name",o);
   if( oidp(f) ) then n := f(o);
   else n := "NoName";
   return n;
end
```

The example shows how we can define the body of a function as a compound expression. A compound expression is a sequence of expressions enclosed between a `begin` and an `end` keyword. In general, the value returned by a compound expression is the value returned by evaluating the last expression. In this case however, we use a return expression. A return expression returns its argument as the value of the function. The formal parameters, i.e. the names given to the arguments of the function, are used as variables throughout the body of the function. In addition to these, local variables can be declared at the beginning of a compound expression, i.e. immediately following the begin. The scope of these variable is limited to the compound expression, with the usual rule that a variable defined in an inner block hides a variable of the same name defined in an outer block. In addition to the use of block and conditional statements, this functions also shows the use of function variables. IPL treats functions as first class objects; in this example, the "Name" function is resolved, and, if a "Name" function is found, it is applied to the object given as the argument [2].

The second example shows the use of iteration. In this example, we are given a list of objects, of type Component, and are asked to compute the sum of the weights of these objects.

```
create function Weight(List of Component l)
                                    -> Real as ipl
begin
   declare w;
   w := 0;
   while(l != []) begin
      w := w + Weight(first(l));
      l := rest(l);
   end;
   return w;
end
```

The above example shows the use of the `while` construct to create an iterative loop. The introduction of iteration is not strictly necessary. For example, the last example could also have been written as a recursive function, as follows:

```
create function Weight(List of Component l)
        -> Real as ipl
   return if( l == []) then 0 else
       (weight(first(l)) + Weight(rest(l)));
```

In addition to the control constructs shown above, IPL defines type constructor functions to create lists, tuples and bags and corresponding accessor functions and constants [3]. This allows database programmers to construct more or less complex aggregate data values whose type is given by a type expression consisting of type constructors applied to atomic types and other, less complex, types.

[1] Note that IPL procedures, i.e. functions that update the database, can not be called this way.

[2] The function `ResolveFunctionByName()` is an example of a (built-in) system function. System functions can be called similar to user-defined functions

[3] To create a list one insert elements starting wthn an empty list. The empty list is denoted with `[]`. For example, the list containing 1, 2 and 3 is created by the following expression: `insert(1, insert(2, insert(3, [])))`. The insert function can also be replaced by a binary infix insert operator: `::`, allowing us to rewrite the above expression as: `1 :: 2 :: 3 :: []`. Finally, one can also use the list brackets `[` and `]` as implicit list constructors, allowing us to construct a list directly as: `[1, 2, 3]` . Similarly to create a bag, we can start from an empty bag {}, or use the notation: { b1, b2, ... }. To create a tuple, start from an empty tuple `[< >]`, or use the notation: `[< t1, t2, ... >]`. We also have a pair of generic accessor functions, `first` and `rest`, which return respectively the first element of a list, tuple or bag, or a list, tuple or bag containing all but the first element. Note that returning the first element of a bag amounts to randomly selecting one of its elements.

5 IPL: Language Definition

The design of the IPL language was motivated primarily by the desire to design a simple language (extension) that would allow the database user to define arbitrary computable functions as an integral part of the database schema design. The approach that was chosen was to design a simple functional language that would embed the capabilities of the underlying database system, primarily the capability to define (and compile) declarative queries and the capability to update stored functions (relations), and to complement these with notions of variables, sequencing, conditionals and iteration/recursion. In addition, it was decided to provide a simple syntax resembling that of imperative languages such as C and Pascal.

5.1 Variables and Function Evaluation

In IPL, we choose to implement a simple model for variables, based on that of traditional imperative languages. Variables can be declared at the beginning of a compound expression. A value can be assigned to a variable anywhere inside the compound expression in which it is declared. One of our original goals in the design of the language was to adopt a substitution model for procedure evaluation. This has been proven difficult to do in practice, since it would rule out iterative control structures that rely on assignment to update the values of the variables that appear in the predicate controlling the iteration. Our model is limited in expressive power, in that we can not define functions that have internal state. In IPL this is not a real limitation, since such functions can be defined by including calls to OSQL procedures that update the state of the database. In Iris we distinguish between functions that have side-effects, e.g. updating the database, and functions that do not. When an IPL function has side-effects this is marked in the system catalogs by the IPL function compiler.

To apply a function, the evaluator will first evaluate the arguments of the function (applicative-order evaluation or call-by-value). This implies that constructs that require different evaluation strategies need to be implemented explicitly, i.e. as syntactic constructs in the language. This restriction may be relaxed in later versions of the language. Specifically, we may want to introduce a simple notation that allows us to quote an expression, thus allowing us to pass the expression as a value to the function. The additional functionality was not provided in the prototype that we implemented, for two reasons: (1) to not complicate the language, e.g. requiring the users to specify the evaluation strategy, implicitly requiring that they understand the issues involved and (2) because for all the constructs that we wanted to implement, we provided special syntax anyhow, thus allowing us to explicitly implement a certain evaluation strategy for these [4].

[4] Note that these constructs map more or less directly to special forms in LISP like languages

5.2 Type System and Typechecking

The type system of IPL is based on that of OSQL. Types are organized in a type hierarchy that allows multiple inheritance. The aggregate types exposed in IPL are List, Tuple and Bag. A tuple is represented by a non-homogeneous list of values, i.e. each value can be of a different type. A list on the other hand is a homogeneous list of values, i.e. all values share a common supertype. Bags are essentially lists with a non-stable ordering. Since ultimately all objects are instances of type Object, the main difference between a tuple and a list is that a tuple has a fixed number of elements, whereas the number of elements in a list may vary. Another way of putting this is that a tuple can also be viewed as a list (of type List of Object). Subtypes of aggregate types can be defined implicitly when creating a function, by specifying an appropriate type declaration for specific arguments and/or results. For example, we can specify the type of a list of integers, or the type of a tuple consisting of a person (a Person), a name (a String) and an age (an Integer), as follows:

```
List of Integer
[<Person, String, Integer>]
```

Note that the argument type of a function with multiple arguments can be viewed as a tuple where the type of an element of the tuple is the type of the corresponding argument of the function. Similarly, we can view the result type of a single-valued function as a tuple, where the type of an element of the tuple is the type of the corresponding result. The type of a multi-valued function can similarly be viewed as a bag of tuples, where the type of the tuple is determined by the types of the result as in the case of a single-valued function.

Typechecking the body of an IPL function is done both at compile time, i.e. when the body is translated into an interpretable expression, and at run-time, i.e. when the body is actually being interpreted. It is the job of the compiler/translator to insert code to do run-time type checking if and where needed. In particular, it is verified that functions are called with arguments of the appropriate types, and that the result returned by the function being compiled conforms with the declared result type. All typechecking is done using the partial ordering defined by the OSQL type hierarchy. In particular, to determine that a function call is type safe, it is determined that the types of the actual arguments are the same as, or subtypes of, the type of the formal argument. Therefore, to do typechecking, the translation module has to determine the type of the variables used in the body of the function. For the variables representing the function arguments, this is simple. We can assume that the actual type of the value is the same as, or a subtype of the type of the input argument. Next, whenever we assign a value to a variable, the type of the variable becomes the type of the value. If we can not (at least not easily) determine the type of the value at compile time, this means that we get a so-called dynamically typed variable. In that case, the compiler will insert additional code that will do the appropriate type checking at run-time, i.e. when the expression is actually evaluated.

Run-time type checking also becomes necessary when we have function variables, as for example in the first example in the previous section. Another case in which we may have to do run-time type checking is when functions are overloaded, in particular, when the result type of a function depends on the type of the argument it is called with.

Run-time type checking is done by calling the appropriate typing function (for non-literal) types or inspecting the object (i.e. the object tag) to determine whether it is an instance of a particular type. For aggregate objects, the process is applied recursively, according to the pattern specified by the aggregates declaration.

5.3 Extensibility

The extensibility of the IPL language is provided by giving access to the underlying intermediate language. This provides an easy and straightforward way of extending the set of functions and functional operators that are available in IPL. If necessary, new syntactic constructs can be provided as well, by extending the parser and defining the implementation of the construct in terms of the intermediate language (see also Section 6).

6 Implementation of IPL

In this section we will discuss the implementation of IPL. After discussing the implementation in the Iris prototype, we will also discuss a generalization of the approach applicable to relational database systems.

6.1 Generalized Foreign Function Node

As said before, the implementation of IPL is based primarily on a feature in the Iris prototype known as a foreign function node [7]. In this section, we will provide some background on foreign function nodes and in particular describe a feature to turn foreign function nodes into what is called a *generalized foreign function node.*

A foreign function node allows the query interpreter to invoke, at run-time, an arbitrary piece of (C) code and have the code compute a sequence of tuples, similar to the sequence of tuples returned by a TABLE node in a relational query interpreter. To achieve this kind of behavior, a foreign function module, i.e. the code that implements a given class of foreign functions, provides six different entry points, resp. xxx_init(), xxx_alloc(), xxx_dealloc(), xxx_open(), xxx_next() and xxx_close(). These entry points, and in particular the open, next and close entry points, provide the capability of implementing foreign functions that return a bag of tuples, where each call to next returns one more tuple to be added to the bag. From a user point of view, the behavior of these entry points, and the order in which they are called, is as follows:

xxx_init The init entry point is called at the time the function is implemented. It is used to compute data to be stored as part of the definition of the function body. This data, referred to as environment data, is made available when the function is called, passing it as an argument to the alloc entry point. [5]

xxx_alloc The alloc entry point is called as the first step of evaluating a function call. Its main purpose is to allocate data structures needed by the open, next and close entry points, as well as to copy the environment data into these data structures. The reason for having a separate alloc entry point is that in the course of interpreting a query, one can have multiple sequences of open, next and close calls, e.g. when the foreign function is called repeatably with different sets of arguments.

xxx_open The open entry point is called to create a scan and serves to initialize the computation. The arguments of the call to the function are passed in to the open in addition to a pointer to the structure allocated by the alloc entry point.

xxx_next The next entry point is called once for single valued functions and repeatedly for multi-valued functions, i.e. until it returns a special status code NOMORE. The xxx_next entry point computes a result tuple each time it is called. For multi-valued functions, these tuples are collected into a bag. Arguments passed to the next entry point are the arguments to the function call, and a pointer to the structure allocated by the alloc entry point.

xxx_close The close entry point is called to close a scan.

xxx_dealloc The dealloc entry point is called as the final step of evaluating a function. It is used to deallocate the data structures allocated by the alloc entry point.

The capability of a foreign function to include, as part of its stored function body, so-called environment data, that can be computed at the time the function is implemented, leads to the notion of so-called generalized foreign function nodes. In particular, we will call a foreign function node a generalized foreign function node if the environment data defines both (an interpretable version of) the body of the function and the (name of the) interpreter to be used for doing that. To see why this is such an important capability, one can consider the possibility of implementing e.g. a (relational) database system completely by 'calling' foreign functions and then interpreting the body of these functions (at run-time) using the appropriate interpreter. Also, looking ahead a little bit, IPL is implemented as a single foreign function module, where the interpreter is a (small, portable) LISP interpreter.

6.2 IPL Foreign Function Module

The generalized foreign function node described above is the basis for defining the IPL foreign function mod-

[5] The init entry point is not strictly necessary; it is however crucial to defining a generalized foreign function node.

ule. This module is used to implement IPL functions. The prototype implementation was implemented by (slightly) modifying the OSQL parser to parse IPL function bodies into a (literal) nested list structure, that is passed in in the environment argument of the IPL foreign function module used to implement the function. The init entry point, which is passed the parsed representation of the IPL function, translates the body into LISP code[6], does type checking (to the extent possible) and generally tries to optimize the LISP code based on its knowledge of invoking database functions. The LISP code (either in string format or in compiled (bytecode) form) is then stored in the database as part of the foreign function node that is the body of the IPL function.

Thus, an IPL function body is translated into a LISP lambda expression. The arguments of the function become the arguments of the lambda expression. Any variable declarations are translated into (nested) let constructs. An if-then-else expression is translated into a similar kind of LISP expression and the while expression is translated into a LISP do loop. All of these translations are straightforward and simple, and are examples of the translation of IPL language constructs into intermediate language (LISP) expressions that can be directly interpreted. Based on the flexibility and extensibility of the intermediate language, it is easy to add other kinds of syntactic constructs to IPL and provide the associated mapping to a LISP expression.

For example, the function NameOfFunction introduced in the examples in Section 4, is translated as follows:

```
(lambda ( o)
 (call-with-current-continuation
  (lambda (return)
   (let ( f n)
    (set! f (iris_eval
      "ResolveFunctionByName"
      (aggr_list "Name" o)))
    (if (oidp f)
        (set! n (iris_eval "f" (aggr_list o)))
        (set! n  "NoName"))
   (return  n)))))
```

The example above shows some more details, including the LISP function iris_eval, that we implemented as part of extending the LISP interpreter. The iris_eval procedure provides a direct interface to the database kernel, allowing us to pass in the name of a LISP variable or the name (or oid) of an (OSQL or IPL) function and a list of arguments and then evaluate either the function that the LISP variable is bound to, or the IPL or OSQL function named as the argument. One possibility not shown is to provide additional entry points into the query interpreter that allow us to bypass succesively more of the typechecking, authorization and function resolution implemented by the iris_eval entry point.

The main reason for having the translation from IPL into LISP is to shield the database user from the details of the underlying LISP implementation. Another reason is that, over time, we plan to develop a specialized intermediate language which is specifically suited to be efficiently interpreted and to allow the procedure compiler to optimize the function body.

[6]Note that in the prototype system we used LISP as the intermediate language.

6.3 Generalization to SQL/relational database systems

In this section we will discuss a (minimal) set of extensions necessary to allow SQL[2], a standard query language for relational database systems, to be extended with IPL constructs. The key to our approach here is the introduction of two kinds of views, respectively (1) a derived view and (2) a computed view [7], similar to the definition of derived and foreign functions in OSQL. Computed and derived views can be implemented by a generalized foreign function node, similar to the Iris foreign function node; for computed views the interpreter is a LISP like interpreter, similar to IPL and for derived views, the interpreter is the SQL system itself.

To give an example of how the SQL language could be extended with derived and computed view statements, look at the following example:

```
create derived view AgeOfPerson(Integer ssno)
    -> Integer age as
  select Person.age from Person
                    where Person.ssno = ssno;

create derived view SalaryOfPerson
            (Integer ssno) -> Real dollars as
  select Person.salary from Person
                    where Person.ssno = ssno;

create computed view IncomeOfPerson
           (Integer ssno) -> Real dollars as
begin
  dollars := SalaryOfPerson(ssno);
  if AgeOfPerson(ssno) >= 65 then
      dollars := dollars +
                  SocialSecurityIncome(ssno);
  return dollars;
end;
```

There are two main differences between the views defined above and SQL views. The first is that views defined above have input arguments and compute a result value. SQL views are more like logic predicates in that any attribute can be an input or output argument. Secondly, the derived view defined above can actually be compiled, e.g.into an SQL section, when it is defined. This allows it to be executed directly, without embedding it into a select statement that has to be compiled before it can be interpreted. The views defined above can be called in a manner similar to OSQL functions and SQL sections[8].

[7]We will assume, for ease of exposure, that we also make the necessary extensions to the SQL parser, to parse the necessary statements into run-time datastructures

[8]Another way of looking to this is to think of a derived view as a named SQL section that can take input parameters.

Note also that, as in the previous examples, concerns regarding the optimization of the computed views can be addressed in two ways: (1) defining the appropriate derived views or (2) make the code computing the body of the view, i.e. the init entry point of the foreign function code module, smarter. In the example above, we would like to be able to prevent the computed view from accessing the person table twice and instead have it fetch both the salary and age of the person. This could be done by defining the appropriate derived view and accessing the elements of the tuple returned by the view using IPL accessor functions.

7 Comprehensive Example

In this section we will give a comprehensive example showing the facilities provided by the IPL language. The example is based on that used in [3]. We have extended it to include active database functionality.

The first task defined in [3] is to define the schema of the example application. In IPL, schema definition consists of defining types and functions on these types. As such, any of the four tasks discussed below can be viewed as defining (part of) the schema.

```
create type Supplier properties (
  Name Charstring);

create type Part properties (
  Name Charstring);

create type BasePart subtype of Part
properties (
  Cost Integer,
  Mass Integer,
  SuppliedBy Supplier
);

create type CompPart subtype of Part
properties (
  AssemblyCost Integer,
  MassIncrement Integer,
);

create function Uses(CompPart p) ->>
       <Part , Integer > as stored;

create function Quantity(CompPart p, Part sp)
        ->> <Integer >
   as select q for each Integer q where
                          Uses(p) = <sp,q>;

create function MadeFrom(CompPart p) ->> Part
   as select p for each Part sp, Integer
     quantity where Uses(p) = <sp,quantity>;

create function UsedIn(Part p) ->> CompPart
   as select sp for each CompPart sp
                    where MadeFrom(sp) = p;
```

The schema consists of a number of type creation statements. All types have one or more properties, that are implemented as single-valued functions, that return a value of the appropriate type. In the definitions above, we defined multi-valued functions using a double arrow (->>). We also define a number of derived functions, implemented by OSQL select statements. In this case, the derived functions are the inverse functions of a property.

The second task is to print the names, cost and mass of all base parts that cost more then a specified amount:

```
create function ExpensiveParts(Integer mcost)
    ->> <Charstring, Integer, Integer> as
select Name(p), Cost(p), Mass(p)
  for each BasePart(p) where Cost(p) > mcost;
```

This task shows the use of OSQL as a true, declarative query language. The OSQL select statement has the same expressive power as the SQL select statement, with the advantages of the functional model, e.g. allowing composition of functions etc.

The function above can be used in an application, or another function. For example, to compute the names, cost and mass of all the base parts costing more then $100. one can call the function, as shown below.

```
call ExpensiveParts(100)
```

The third task is to compute the total mass and total cost of a part. We first create an auxiliary function, to help locate a part:

```
create function PartNamed(Charstring n)
      -> Part as
   select p for each Part p where Name(p) = n;
```

Next we create a function computing the cost and mass for a BasePart and a similar function for computing the cost and mass of a CompPart.

```
create function costAndMass(BasePart p) ->
          <Integer, Integer> as
   select Cost(p), Mass(p);

create function costAndMass(CompPart p) ->
  <Integer, Integer > as
begin
  declare resultCost, resultMass, subparts;

  resultCost := AssemblyCost(p);
  resultMass := MassIncrement(p);
  subparts   := MadeFrom(p);
  while(subparts != {}) begin
     declare imResultTuple, subpart, quantity;

     subpart := first(subparts);
     subparts := rest(subparts);
     quantity := Quantity(p, subpart);
     imResultTuple := costAndMass(subpart);
     resultCost := resultCost + quantity *
    first(imResultTuple) ;
     resultMass := resultMass + quantity *
    second(imResultTuple) ;
   end
   return [<resultCost, resultMass>];
```

```
end
```

The two functions above, i.e. PartNamed and costAndMass, can be used in an application to return the cost and mass of a part called CPU., e.g.:

```
call costAndMass(PartNamed('CPU'));
```

Several things are noteworthy here:

- use of overloading and late binding: the function costAndMass is overloaded on the types BasePart and CompPart. Combined with the use of late binding, this allows us to call it with either a BasePart or a CompPart and have the system choose the appropriate implementation, based on the most specific type of the argument.
- use of (iterative) control construct and recursion: in the case of a composite part, we need to iterate over the bag of subparts of the composite part, recursively compute the cost and mass of the part and sum them, in order to obtain the cost and mass of the composite part.
- use of aggregate data values (bags, tuples) and accessing them (functions first, rest, second)
- explicit control flow: note that we implemented the iteration by means of a while loop, rather then some form of transitive closure operation.
- the compactness and readability of the code, (1) 'look and feel' of a simple PASCAL/C like language, (2) selective use of syntactic sugar, (3) operations on aggregate data values.

The fourth task is to create a new composite part, enforcing one or more integrity constraints.

```
create function newCompPart(Charstring
   name, Integer massIncr, Integer
   assCost, Bag of <Parts, Integer>
   subparts) -> CompPart p as
begin
   declare props, vals;

   if(size(subparts) > 10) then
      RaiseError(1001,
  "Max. number of subparts (10) exceeded");
   if(PartNamed(name) != null)
      RaiseError(1002,
    "Similarly named part exists already");
   props := ["Name","MassIncrement",
                    "AssemblyCost","Uses"];
   vals = mkpropvaluelist(name,massIncr,
                         assCost, subparts);
   p := first(
     ObjCreateInit("CompPart",props,vals));
end
```

The function above creates a new composite part, provided a similarly named part does not already exist, and the number of subparts is not greater then 10. The function `RaiseError` is used to allow a user to return their own error messages. Its effect is to raise an error and, if the function was a procedure, to undo any side-effects (updates) that were made as part of the request that invoked it. The function `mkpropvaluelist` is another example of an aggregate constructor function. In this case it is special, since its only purpose is to construct a list in the format expected for the third argument of the function `ObjCreateInit`.

Finally, we will discuss an additional tasks, that is not part of the example in [3]. This task is included to show the potential of being able to define and store arbitrary computable functions in the database.

The task that we added is to order new basic parts when the inventory drops below an established threshold. It requires us to extend the schema in a number of ways. First, we create a number of new stored functions, to keep track of ordering information.

```
create function PreferedSupplier(BasePart p)
                -> Supplier s as stored;
create function QuantityToOrder(BasePart p)
                -> Integer quantity as stored;
create function QuantityOnHand(BasePart p)
                -> Integer quantity as stored;
create function MinQuantityOnHand(BasePart p)
                -> Integer quantity as stored;
```

Next, we define a function to actually order a new supply of BasePart's. This function formats a message, that then needs to be send to the supplier of the base part. This is something that requires that the database interact with some process in the outside world [9]. In the prototype, we use HP's Softbench broadcast message server for implementing the necessary communication. That way, the database must only include the functionality required to send a Softbench message and need not directly interact with the outside process. The functionality is encapsulated by the function `broadcast_msg`, which takes an argument, a string, and forwards it to the Softbench message server for further distribution.

```
create function OrderBasePart(BasePart p) ->
Integer as
begin
   declare flag, msg,partSupplier, quantity;

   partSupplier := Name(PreferedSupplier(p));
   quantity := QuantityToOrder(p);
   msg := stringAppend("ORDER-SUPPLIES",
                       partSupplier,quantity);
   flag := broadcast_msg(msg);
   return flag;
end
```

The next function `lowerQuantityOnHand` does two things. First it checks to see the quantity of parts available for a given part. If this, minus the amount to give

[9] Such a process is typically available in the form of a service in the distributed system that the database is part of, e.g. in the network computing architecture (NCS) of HP/Apollo

out, goes below a threshold, a (trigger) function will be enabled to fire on commit of the transaction, i.e. when the change has become permanent. Note also that we need a helper function to do the actual update, i.e. setting the quantity on hand.

```
create function setQuantityOnHand(BasePart p,
                        Integer q) -> Boolean as
   set QuantityOnHand(p) = q;

create function lowerQuantityOnHand
      (BasePart p, Integer q) -> Integer qoh as
begin
   qoh := QuantityOnHand(p) - q;
   if(qoh < MinQuantityOnHand(p)) then
      callOnCommit("OrderBasePart",[p]);
   setQuantityOnHand(p,qoh);
end;
```

The callOnCommit() function abstracts the event/trigger functionality. It causes the function named in its first argument to be called with the list of arguments that is its second argument, when the transaction commits.

The above functions can now be used in a straightforward fashion by application programmers, implementing e.g. an inventory control system.

For example,

```
call lowerQuantityOnHand(
        PartNamed('LAN card'), 2);
commit;
```

could trigger the ordering of new LAN cards if the `lowerQuantityOnHand` function would reduce the quantityOnHand below the set threshold.

It is easy to see that much more complicated ordering strategies could be provided as well. Note also that a crucial aspect for this to work is that the authorization mechanisms should allow us to specify that the function `setQuantityOnHand` can only be called by the function `lowerQuantityOnHand`.

8 Conclusions

In this paper we have described a first prototype of a simple, extensible language to allow database users to define arbitrary computable functions and procedures, store them as part of the database schema and invoke them as part of a database transaction, under control of the database system.

The method by which this language was implemented was discussed in some detail as well. It shows how similar kinds of extensions can be made to relational database systems and how in fact IPL is just one example of a database programming language developed using these principles. In the paper, we argue that the mapping of an external, syntactic form into an intermediate language that is computationally complete and easily extensible is a powerful way of implementing application specific end-user languages. It is our believe that such languages will allow more reliable, faster, and thus less costly development of database applications, as well increase the range of applications using database technology, e.g. by the inclusion of active database functionality.

There are many areas that can be worked on to further extend this approach, including the development of more specialized intermediate languages, that (1) can be optimized to take advantage of the capabilities of the underlying system, e.g. parallel hardware, and (2) are better suited as the implementation language for increasingly expressive high-level declarative database programming languages.

Acknowledgements

The author would like to acknowledge the contributions of his colleagues in the database technology department of HP labs and the object-oriented database section of the commercial systems division of HP. In particular, the author acknowledges the contributions and feedback received from Rafiul Ahad, Stefan Gower, Waqar Hasan, Bill Kent, Mohammed Ketabchi, Ravi Krishnamurthy and Peter Lyngbaek.

References

[1] H. Abelson and G.J. Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, Mc. Graw Hill, 1985.

[2] American National Standards Institute. *Database Language SQL.* ANSI X3.135-1986.

[3] M.P. Atkinson and P. Buneman. Types and persistence in database programming languages. *ACM Computing Surveys*, 19(2), June 1987.

[4] F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez. Fad, a powerful and simple database language. In *Proceedings of the Thirteenth International Conference on Very Large Data Bases*, 1987.

[5] D. Batory. Principles of Database Management System Extensibility. *IEEE Database Engineering Bulletin*, June 1987.

[6] M. Carey and D. DeWitt. An Overview of EXODUS. *IEEE Database Engineering Bulletin*, June 1987.

[7] T. Connors and P. Lyngbaek. Providing Uninform Access to Heterogeneous Information Bases. In Klaus Dittrich, editor, *Lecture Notes in Computer Science 334, Advances in Object-Oriented Database Systems.* Springer-Verlag, September 1988.

[8] Anthony J. Field and Peter G. Harrison. *Functional Programming.* Addison-Wesley, 1988.

[9] D. H. Fishman, J. Annevelink, D. Beech, E. C. Chow, T. Connors, J. W. Davis, C. G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M. A. Neimat, T. Risch, T. A. Ryan, M. C. Shan, and W. K. Wilkinson. Overview of the Iris DBMS. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Languages, Applications, and Databases.* Addison-Wesley Publishing Company, 1989.

[10] G. Gardarin, J. P. Cheiney, G. Kiernan, D. Pastre, and H. Stora. Managing Complex Objects in an Extensible Relational DBMS. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, Amsterdam, The Netherlands, August 1989.

[11] L. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. Extensible Query Processing in Starburst. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, Portland, Oregon, June 1989.

[12] V. Linnemann, K. Kuspert, P. Dadam, P. Pistor, R. Erba, A. Kemper, N. Sudkamp, G. Walch, and M. Wallrath. Design and Implementation of an Extensible Database Management System Supporting User Defined Data Types and Functions. In *Proceedings of the Fourteenth International Conference on Very Large Data Bases*, Los Angeles, California, August 1988.

[13] M. Mannino, I Choi, and D. Batory. An overview of an object-oriented functional data language. In *Proceedings of IEEE Data Engineering Conference*, 1989.

[14] F. Manola and U. Dayal. PDM: An Object-Oriented Data Model. In *Proceedings of 1986 International Workshop on Object-Oriented Database Systems*, Pacific Grove, California, September 1986.

[15] K. Morris, J. Ullman, and A. vanGelder. Design overview of the nail! system. In *Proceedings 3rd International Conference on Logic Programming*, 1986.

[16] J. Mylopoulos, P. A. Bernstein, and H. K. T. Wong. A Language Facility for Designing Database-Intensive Applications. *ACM Transactions on Database Systems*, 5(2), June 1980.

[17] G. Phipps. Glue: A deductive database programming language. Technical report, Dept. of Computer Science, Stanford University, 1990.

[18] T. Risch. Monitoring Database Objects. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, Amsterdam, The Netherlands, August 1989.

[19] D. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 6(1), September 1981.

[20] M. Stonebraker, J. Anton, and E. Hanson. Extending a Database System with Procedures. *ACM Transactions on Database Systems*, 12(3):350–376, September 1987.

[21] K. Wilkinson, P. Lyngbaek, and W. Hasan. The Iris Architecture and Implementation. *IEEE Transactions on Knowledge and Data Engineering*, December 1989.