A Functional Programming Approach to Deductive Databases

Alexandra Poulovassilis
Department of Computer Science,
University College London,
Gower St., London WC1E 6BT

Carol Small
Department of Computer Science,
Birkbeck College,
Malet St., London WC1E 7HX

Abstract

We introduce a persistent functional language called PFL which adapts functional programming to the area of deductive databases, much as logic-based deductive database languages adapt logic programming. PFL inherits the advantages of functional programming languages, including higher-order functions, static type checking, lazy evaluation, and support for user-defined types and constants. However, PFL allows functions to be defined incrementally by the insertion and deletion of equations, and stores these equations in the database. PFL also supports a class of extensionally defined updateable functions called selectors which allow the storage of arbitrarily nested values. Further functions can be written over selectors which act as derivation rules and which are "invertible" in the sense that they simulate predicates. We begin the paper by motivating the development of PFL. We review the respective advantages of functional and logic programming, particularly with respect to database modelling and manipulation, and we compare PFL with other functional database languages. We describe the salient features of the language and examine its expressiveness with respect to data modelling, computation and updates. We also describe the implementation of PFL concentrating in particular on the storage and retrieval of its persistent data.

1. Introduction

Deductive databases have been the focus of much research during the past decade or so. Broadly speaking, deductive databases couple a conventional database with a knowledge base, the knowledge base consisting of a set of rules and an inference engine. Generally, deductive database researchers have assumed that rules are expressed as logic formulae and that the inference engine is a first order logic theorem prover [Gal84, Gra88a, Cer90, Chi90, Hor90]. More recently, the limitations of this approach with respect to both data modelling and computation have led to a number of extensions to first order logic being proposed, including sets [Abi88a], types and object identifiers [Abi89, DBP89], higher-order syntactic features [Che89] and functions [Gru89]. None the less, knowledge representation techniques other than first order logic are possible, for example frames or production rules, and we believe they deserve further consideration as candidate paradigms for deductive databases. In particular we are investigating the use of rules expressed as *equations* and an inference engine which is a *functional evaluator* [Pey87]. Our knowledge base can thus be viewed as a production rule system with goal-driven problem solving.

We describe in this paper a functional language called PFL which adapts functional programming to the area of deductive databases, much as logic-based deductive database languages adapt logic programming. Thus, PFL has features in common both with logic-based deductive database languages, and with functional programming languages.

In common with logic-based deductive database languages, PFL supports

- the persistence of both factual and procedural information,
- recursive data structures such as trees and graphs,
- the definition and storage of rules which enable the derivation of further information from factual data.

In addition, PFL supports default rules, which allow the graceful handling of missing and incomplete information, and update procedures which maintain the semantic integrity of the factual data.

In common with functional programming languages, PFL supports higher-order functions and user-defined types. However, unlike conventional functional languages, PFL allows functions to be defined incrementally by the insertion and deletion of equations, and maintains these equations in a repository on secondary storage. Thus, much larger volumes of information can be handled while achieving comparable query performance by caching the most frequently used equations ([Sma90] gives performance figures).

PFL supersedes functional query languages such as FQL [Bun82] and GENESIS [Bat88] in which only extensionally defined functions can be stored. PFL also improves on the implementations of DAPLEX [Smi83, Kul86, Gra88b] since these have extended the functional data model [Shi81] with non-functional computation, thereby introducing an impedance mismatch, whereas in PFL all querying is purely functional. Also, languages based on the functional data model can represent only binary relationships (by single-argument functions and their inverses) and an n-ary relationship must be modelled by an entity type participating in n binary relationships. Reconstructing the n-ary relationship for the purposes of querying requires these binary relationships to be joined and can be prohibitively expensive. In contrast, PFL supports n-ary relationships directly.

A forerunner of PFL was the functional database language FDL [Pou90] which integrates functional programming with the functional data model. PFL improves upon FDL by supporting n-ary relations rather than just binary ones. PFL also infers the types of functions incrementally in the face of equation insertions and deletions whereas in FDL function types are declared by the user and are static. Finally, PFL's query evaluator has several optimisations which give PFL enhanced query performance, comparable to that of contemporary functional programming languages.

A number of other database systems have also incorporated functions [Ban87, Day87, Bee88, Hei88] but they have all assumed that intentionally defined functions are coded in some external programming language rather than stored in the database. Finally, several database programming languages do support the persistence of functions on secondary storage [Alb85, Car85, Mor89]. However, unlike PFL, these languages do not allow the incremental update of functions, only their deletion and complete redefinition.

Since a key feature of PFL is its foundation on functional as opposed to logic programming, we conclude this introductory section by reviewing the advantages of each paradigm, particularly with respect to database modelling and manipulation.

The main advantage of logic programming is that, unlike functions, predicates can (at least theoretically) be used with any number of their arguments uninstantiated. This makes logic programs more versatile than functional ones. Also, in the context of deductive databases, facts and derivation rules are often expressed more naturally in terms of single predicates than in terms of functions and their inverses. Despite this fact, our premise is that the advantages of functional programming which we outline below make it worth evaluating as a foundation for deductive databases. Also, as we will see in Sections 2 and 3, PFL supports a class of extensionally defined updateable functions which simulate extensional predicates and which go much of the way to restoring the flexibility of relational programming.

The most obvious advantage of functional programming is that it is higher order and so all expressions are first class objects. Higher order functions can be written which "abstract out" recursion patterns over recursive data structures. These functions can then be used to write more concise programs which do not include explicit recursion. Consider, for example, the higher-order functions `map` and `fold` defined by the following equations[1] :

```
map f [ ]              = [ ]
map f (x:y)            = (f x):(map f y)
fold f end  [ ]        = end
fold f end  (x:y)      = f x (fold f end y)
```

`map` and `fold` can be used to define further functions which increment a list of numbers by a constant, sum a list of numbers, append two lists, concatenate a list of lists, test whether a list of booleans contains at least one True value, and test for the membership of an element in a list[1] :

```
incr  c  x             = map ((+) c) x
sum  x                 = fold (+) 0 x
append  x  y           = fold (:) y x
concat  x              = fold (append) [ ] x
or_list  x             = fold (or) False x
member  x  c           = or_list (map ((==) c) x)
```

A very useful high-level query construct supported by most functional languages is the *list abstraction* [Pey87], [e | q_1; q_2; ... ;q_n], which may be read as "the list of values e such that q_1 and q_2 and ... and q_n". Each q_i may be either a *generator* $p_i \leftarrow l_i$, read as "the pattern p_i is matched against each element in the list l_i in turn", or a boolean-valued expression which must be satisfied. List abstractions are only syntactic sugar and can be translated into a series of higher-order function applications (see [Pey87] for details). For example, the expression

[salary x | x ← employees; (age x) > 50]

is equivalent to the expression

map salary (filter (λx.((age x) > 50)) employees)

A second feature of functional languages is their use of pattern-matching rather than unification for parameter passing. Deterministic computations can thus be expressed more succinctly since there is no need to communicate intermediate results via common variables. Also, the deterministic semantics of functional evaluation can be exploited for the representation of *default rules*. For example, given the following function :

```
tax_code Jim           = "336A"
tax_code x             = "307L"
```

the query `tax_code Sue` evaluates to "307L" while the query `tax_code Jim` evaluates to "336A". In contrast, first order logic is monotonic and so default knowledge must be represented extra-logically in logic-based languages.

Lastly, most modern functional languages are typed. They provide a set of built-in types, and facilities for the definition of new types and constants. Any expression has a unique type which can be inferred at compile-time by a unification-based type checker [Car84], provided the type is finite. Such a type checker is especially useful in a database environment since it can be used both as a special-purpose integrity enforcer (equations are only inserted if they are correctly typed) and also to avoid running potentially expensive queries which are incorrectly typed and which may ultimately yield no useful information. Cardelli [Car88a] describes how type checking can be extended to support multiple inheritance of functions via type inclusion, and his work forms a promising foundation for integrating functional and object-oriented database languages. In fairness, we note that a number of type checking techniques have also been proposed for logic programming languages, for example [Myc84, Xu89], although these have yet to find their way into working systems.

The layout of the rest of this paper is as follows. In Section 2 we introduce PFL. We describe its type system and its support of user-defined types and constants.

---

[1] We use PFL's syntax in all our examples of functions. In this syntax, [ ] is an empty list, (x:y) is a list with head x and tail y, = is used for definitions and == for equality . Whenever binary infix operators such as +, : and == need to be used prefix, they are enclosed in round brackets.

We discuss the incremental update and type-checking of function definitions. We introduce a class of extensionally defined updateable functions called *selectors* which are "invertible" in the sense that they simulate predicates. We show how functions can be written which draw inferences from selectors and which are also invertible. Finally, we discuss the update of selectors. In Section 3 we address PFL's expressiveness with respect to data modelling, computation, and updates. In particular, in 3.2 we compare PFL's class of invertible functions with Datalog and we show how any stratified Datalog IDB predicate can be simulated by a PFL function, subject to the proviso that there exists a predetermined order of firing the rules which define the predicate. In Section 4 we discuss PFL's implementation, showing how all of PFL's persistent data is stored in a database consisting of two files, a B^+ tree and an extendible hash file. We conclude in Section 5 with a summary of our contributions and directions of further research.

2. Overview of PFL

PFL is a polymorphic, statically typed, persistent functional language. It provides a set of built-in types, and facilities for the definition of new types and constants. Its functions are defined incrementally by the insertion and deletion of equations. The types of functions are inferred incrementally in the face of such updates. PFL's bulk data is stored in a class of functions which we call *selector functions* or *selectors*. Selectors store and return lists of values. Although these values may be arbitrarily nested, by convention we call them "tuples". Similarly, we call the list of values stored in a selector a "relation". As we will see below, selectors go much of the way to providing the flexibility of relational programming.

Selectors are updated by two built-in functions, `include` and `exclude`, which insert and delete tuples into their associated relations. Both functions operate by side effect although between updates selectors remain deterministic. Hence, queries which do not invoke `include` or `exclude` are purely functional and have no side effects. Conversely, queries which do invoke `include` or `exclude` will have side effects and so will tend to be written in a procedural style.

We describe selectors in greater detail in 2.3. In 2.4 we discuss PFL's evaluation semantics. In 2.5 we discuss functions which draw inferences from selectors and in 2.6 we discuss update procedures.

2.1 PFL's Type System

PFL's type system comprises three layers c.f. [Car88b]:

- The metalevel type Type which is the set of all types.
- The object-level types, both built-in and user-declared. These are also regarded as metalevel values of type Type.
- The values of each object-level type.

The built-in types are Str, Num and Char which are initially populated by strings, integers and characters. New constants, whether object-level or metalevel, are declared using the command :

declare <constant> :: <type>

These constants are termed *constructor functions* or *constructors* in functional programming parlance [Pey87] since they construct values of the indicated type. For example, we can declare person, boolean, list, and product types by the following statements[2] :

declare Person :: Type
declare Bool :: Type
declare List :: a→Type
declare Prod1 :: a→Type
declare Prod2 :: a→b→Type
declare Prod3 :: a→b→c→Type ...

and we can declare (object-level) constructors for these types[2] :

declare Jim :: Person
declare Sue :: Person
declare True :: Bool
declare False :: Bool
declare Nil :: List a
declare Cons :: a→(List a)→(List a)
declare Single :: a→(Prod1 a)
declare Pair :: a→b→(Prod2 a b)
declare Triple :: a→b→c→(Prod3 a b c) ...

Thus, Person is a type, Prod2 s t is a type, for any types s and t, Jim is a value of type Person, and Pair u v is a value of type Prod2 s t, where u is a value of type s and v is a value of type t. Zero-argument types such as Person above can be viewed as *object types* and constructors such as Jim and Sue above can be viewed as *object identifiers*. It is also possible to declare constructors for use as *null values*. For example, the following constructors can be used in the place of any value :

declare Any :: a
declare None :: a

New constructors can be declared at any time during the lifetime of a database. Only one declaration can exist for a given constructor, although constructors can be deleted and re-declared. A constructor can only be deleted if there are no references to it (we describe how this is verified in Section 4).

2.2 Functions

Functions are defined by means of equations which are inserted using the syntax

define <lhs> = <expr>

and deleted using the syntax

delete <lhs>

Equations have unique left hand sides so if the right hand side of an equation needs to be modified the equation must be deleted and re-inserted. For example, the following statements result in the expected definition for the 3-ary function `if` :

define if True x y = x
define if False x y = x
delete if False x y
define if False x y = y

Unlike constructors, functions do not have to be declared before they are used : if no equations exist for an identifier it is assumed to have an unconstrained

[2] In PFL, identifiers starting with an uppercase letter are constructors (either metalevel or object-level) and identifiers starting with a lowercase letter are variables (again, either metalevel or object-level).

type. Thus, programs can be developed "top-down" by using identifiers before they are defined. However, the user is free to predeclare the type of a function if he so wishes and this acts as an extra aid to writing correct programs.

Whenever a new equation, e, is specified for a function, f, the type checker infers a type, τ(e), from e for f. The type τ(e) is unified with the existing type for f, τ(f). If this unification fails, the equation is rejected. Otherwise if τ(f) has become more specific, a message to that effect is displayed and any equations which contain f in their right hand side are also type checked again (we explain in Section 4 how these equations are located). A cascade of type checking is thus set off. Such a cascade always terminates since PFL functions have finite types and a finite type can only become more specific a finite number of times. For example, given the built-in function + of type `Num→Num→Num` and the hitherto unused identifiers `incr` and `map`, the statement

define incr c = map ((+) c)

gives the message `incr::Num→a`. Subsequently, the statement

define map f [ ] = [ ]

gives the message

```
map :: a→[b]→[c]
incr :: Num→[a]→[b]
```

and finally the statement

define map f (x:y) = (f x):(map f y)

gives the message

```
map :: (a→b)→[a]→[b]
incr :: Num→[Num]→[Num]
```
[3].

Whenever an equation e defining a function f is deleted, the type of f is re-inferred from its remaining equations (since the deletion of e may have made this type less specific). If τ(f) has indeed changed, any equations which reference f are also type checked again similarly. For example, given the functions incr and map above, the statement

delete map f [ ]

gives no message. Subsequently, the statement

delete map f (x:y)

gives the message `map::a, incr::Num→a`, and finally the statement

delete incr c

gives the message `incr::a`. When all the equations defining a function have been deleted, the function is left with an unconstrained type (the metalevel variable a in the above messages) and so can be redefined as a completely different function.

Our incremental type-checking of functions is similar to that proposed for FQL [Nik85] in that we maintain a dependency graph. However, our approach differs from that of [Nik85] in that our dependency graph records which *equations* reference each function as opposed to which *functions* reference each function, and our type-checking occurs on an equation as opposed to on a function level. This is in keeping with the incremental nature of PFL, where functions are updated by the insertion and deletion of equations rather than by their complete redefinition.

[3] Lists and tuples occur so commonly in PFL that we provide a shorthand notation for them. We use [ ] instead of Nil, (:) instead of Cons, [a1,...,aN] instead of a1:(... :(aN:[ ]), {a} instead of Single a, {a,b} instead of Pair a b, {a,b,c} instead of Triple a b c, and so on. Similarly at the meta level we use [a] instead of List a, {a} instead of Prod1 a, {a,b} instead of Prod2 a b, and so on.

2.3 Selector Functions

Selector functions are declared by a statement of the form :

selector <name> :: <type> → [<type>]

where <type> is a first-order type. A newly declared selector, f, may be assumed to be defined by the following equation :

f x = [y | y ← relation; matches x y]
where relation = [ ]

Here, the argument x acts as a search pattern. Each of its components is either a constant or the "wildcard" constructor Any (we declared it in 2.1 above). The function `matches` compares x with each element y drawn from the list `relation` and returns True or False according to whether x matches y. Thus, the selector returns a list of the tuples which match x, in the order they are encountered in `relation`. As we describe in greater detail in Section 4, the relations associated with selectors are stored and retrieved using an extendible hashing scheme [Enb88]. Hence, the efficiency of evaluating a query `f x` is directly related to the proportion of specified components appearing in the search pattern x.

Updates to a selector result in `relation` expanding or contracting. The inclusion of a tuple is achieved by the built-in polymorphic function include :: (a→[a]) →a→Bool which takes a selector and a tuple and returns False if the tuple is already present in `relation`, otherwise it returns True and updates `relation` by side effect. The new tuple is inserted into an arbitrary position. However, this position remains fixed and so the updated selector will be deterministic. The exclusion of tuples is accomplished similarly by the built-in function exclude :: (a→[a])→a→Bool which takes as arguments a selector and a search pattern, deletes the tuples matching the pattern from `relation`, and returns True or False according to whether any tuples have been deleted. If no tuples remain in `relation`, the selector is left with an unconstrained type and can be redefined as a completely new function.

Consider, for example, the following selectors which record the class of Person objects, and the personal details of each person (address and date of birth), respectively :

selector people :: Person → [Person]
selector pdetails :: {Person,Str,{Num,Num,Num}}
→ [{Person,Str,{Num,Num,Num}}]

Assuming that Jim, Sue and Bob are constructors of type Person, the following queries all return True :

include people Jim
include people Sue
include people Bob
include pdetails {Jim,"10 Sunset Bvd",{12,10,55}}
include pdetails {Sue,"10 Sunset Bvd",{25,10,23}}
include pdetails {Bob,"10 Sunset Bvd",{18,8,25}}

and the selector definitions which result are as follows, up to a permutation of the elements in their associated relations :

people x = [y | y ← relation; matches x y]
where relation = [Bob,Sue,Jim]

pdetails x = [y | y ← relation; matches x y]
where relation =
[{Jim,"10 Sunset Bvd",{12,10,55}},
{Sue,"10 Sunset Bvd",{25,10,23}},
{Bob,"10 Sunset Bvd",{18,8,25}}]

The following query then returns False since it is looking to delete the details of everyone born in September and there are no tuples which match :

exclude pdetails {Any,Any,{Any,9,Any}}

Finally, the following queries return the lists [Bob, Sue, Jim], [ ] and [{Jim, "10 Sunset Bvd", {12,10,55}}, {Sue, "10 Sunset Bvd", {25,10,23}}], respectively :

people Any
people Fred
pdetails {Any,Any,{Any,10,Any}}

2.4 Semantics of Query Evaluation

PFL queries are evaluated by the standard technique of *graph reduction* [Pey87] employed by most functional programming languages. Essentially, this repeatedly selects and rewrites applications of functions or constructors until no further simplification of the query is possible. We diverge somewhat from implementations of other functional languages since our functions are stored and retrieved as individual equations, rather than as whole definitions, and since we incorporate selectors. So we outline PFL's evaluation semantics below.

In PFL, an application of an n-ary function or constructor to n arguments, f a_1 ... a_n, is evaluated as follows :

- If f is a constructor, the arguments a_1 ... a_n are recursively evaluated in turn, to the expressions a'_1 ... a'_n say, and the application is replaced by f a'_1 ... a'_n.
- If f is a built-in function, the built-in code for f is executed (this may call for the evaluation of one or more of the a_i) and the application is replaced by the result.
- If f is a selector, the application is of the form f a_1. In this case, a_1 is evaluated, to a'_1 say, and the application is replaced by a list of the tuples from the relation associated with f which match a'_1.
- Finally, if f is any other function, the application is replaced by the right hand side of an equation, after any variables in the equation have been substituted by the corresponding arguments. The equation is selected by applying a *left-to-right, best-fit* pattern-matching algorithm [Pou90] : the equations defining the function f are compared with each of the arguments a_1,...,a_n and at each a_i only those equations which contain the most specific match for this argument are considered for a_{i+1}. This algorithm guarantees that at most one equation is left after a_n has been matched. There may of course be no equations left, in which case the query is aborted with an error message. We describe the implementation of this pattern-matching algorithm in Section 4.

As to the order in which function applications are selected for rewriting, we chose the *left-most, outermost* application each time. This is called *normal order* reduction [Pey87] and has the desirable effect of delaying the evaluation of function arguments until such time as their value needs to be known, either to match against a function definition or a relation, or to compute a built-in function.

2.5 Making Inferences from Selectors

Selectors can be used just as any other function when specifying equations and queries. In particular, functions can be written which make inferences from selectors. To illustrate, we consider a "parts" database which contains two selectors recording (a) base parts and their unit cost, and (b) composite parts and their immediate sub-parts, and the quantity thereof:

selector base :: {Part,Num} → [{Part,Num}]
selector composite :: {Part,Part,Num}
→ [{Part,Part,Num}]

In Figure 1 below we give some example functions for this database. In these examples, we assume the usual `head` and `tail` list functions and also the following functions for projecting into tuples :

first_of_two	{x,y}	= x
second_of_two	{x,y}	= y
first_of_three	{x,y,z}	= x
second_of_three	{x,y,z}	= y ...

The first function in Figure 1 is a recursive function cost::Part→Num which computes the cost of any part : if x is a base part its cost is obtained from the `base` selector, otherwise its cost is obtained by recursively summing the costs of its immediate sub-parts.

The second function in Figure 1 is bom::[{Part,Part,Num}] which returns a bill of materials i.e. it computes the transitive closure of the `composite` selector, creating at most one entry for each super-part/sub-part pair, and appends the base parts to the result. We note the use of the keyword **where** to introduce some local equations in this definition. The functions defined by these equations are accessible only by each other and by the main equation. This facility to hide auxiliary functions is a useful modularisation technique supported by most functional languages.

In the definition of `bom`, `close` starts off the transitive closure computation; `tc` repeatedly infers new tuples and adds them to the relation `total` until no more new tuples can be inferred; `infer` performs one inference step by joining the last increment to `total` with `composite`; `add` merges the new tuples into the `total` relation; and finally, `add_one` merges a single tuple into `total` making sure that there is only one entry for each super-part/sub-part pair.

The third equation in Figure 1 defines the function sub_parts::{Part,Part}→[{Part,Part}] which computes an intentional relation containing the transitive closure of the sub-parts relationship (ignoring quantities). As in

```
cost x =     if      (base {x,Any} == [ ])
                     (sum [(cost sp) * q | {p,sp,q} ← composite {x,Any,Any}])
                     (second_of_two(head(base {x,Any})))
bom =        append  (close (composite Any))  [{p,None,None} | {p,c} ← base_part Any]
             where
             close start    = tc start start;
             tc  [ ] total  = total;
             tc incr total  = tc (infer incr) (add total (infer incr));
             infer incr     = [{p,sp2,q * q2} | {p,sp,q} ← incr; {p2,sp2,q2} ← composite {sp,Any,Any}];
             add total [ ] = total;
             add total (x:y)      = add (add_one total x) y;
             add_one [ ]  {p,sp,q}       = [{p,sp,q}];
             add_one ({p,sp,q}:y) {p2,sp2,q2}= if ((p == p2) and (sp == sp2))
                                                   ({p,sp,q+q2}:y)
                                                   ({p,sp,q}:(add_one y {p2,sp2,q2}))
sub_parts {x,y}  =    append    [{p,sp} | {p,sp,q} ← composite {x,y,Any}]
                                [{p,sp2} | {p,sp,q} ← composite {x,Any,Any};{p2,sp2} ← sub_parts {sp,y}]
sub_parts {Any,y} = append      [{p,sp} | {p,sp,q} ← composite {Any,y,Any}]
                                [{p2,sp}| {p,sp,q} ← composite {Any,y,Any}; {p2,sp2} ← sub_parts {Any,p}]
```

Figure 1 : Examples for Section 2.5

logic-based languages, `sub_parts` is "fully invertible" in that it can be used to find

(a)
 the transitive closure of the sub-parts relationship :
 sub_parts {Any,Any}

(b)
 the sub-parts of a given part p :
 map (second_of_two) (sub_parts {p,Any})

(c)
 the super-parts of a given part q :
 map (first_of_two) (sub_parts {Any,q})

(d)
 whether a part p is a super-part of a part q :
 sub_parts {p,q} != [ ].

Of course, the query `sub_parts {Any,q}` in (c) is unnecessarily expensive because it results in the enumeration of `composite {Any,Any,Any}`. A better definition of sub_parts would also contain the last equation in Figure 1 which closes `composite` "left-wards" rather than "right-wards". The left hand side of this equation is a more specific match for the query in case (c), and also in case (a) incidentally, and so this definition will be selected when evaluating these queries.

We can draw some important conclusions from the above examples. Firstly, PFL shares the advantages of functional programming languages with respect to deterministic computations such `cost` and `bom`. Secondly, PFL is more expressive than other functional languages since it supports selectors and "invertible" functions over them. We examine the expressiveness of this class of PFL functions in Section 3 below. Lastly, since PFL is list-based rather than set-based, the queries in (a) - (d) may return lists which contain duplicate elements. In general, duplicates can be removed by wrapping the `make_set` function round list-valued expressions :

```
make_set [ ] =       [ ]
make_set (h:t) =     h:[y | y ← (make_set t); y != h]
```

2.6 Updating Selectors

Since bulk data is stored using selectors, it is important that PFL provide tools to maintain the semantic integrity of this data. In fact, our support of the `include` and `exclude` operations as built-in functions means that they can be be embedded within PFL expressions and can be used to write functions which serve as *update procedures*. The query evaluation semantics of 2.4 determine the order in which `include` and `exclude` are evaluated and thus sequence their update effects. PFL update procedures can enforce a wide variety of integrity constraints, including transition, uniqueness and cardinality constraints. We illustrate in Figure 2 three update procedures for the `composite` selector of 2.5. These procedures all have type {Part,Part,Num}→[Bool]. We note that these update procedures rely on the elements of lists being evaluated from left to right, in accordance with the semantics of constructor applications in 2.4. `add_subpart` ensures that there is only one tuple for each part/sub-part pair; `update_quantity` checks that the given part/sub-part pair pair exists and updates its quantity field; and `increment_quantity` checks that the increment passed to it is positive and that a single tuple remains for the given part/sub-part pair.

We examine the expressiveness of PFL's updates further in Section 3 below.

3. Expressiveness Issues

3.1 Data Modelling

The ability to declare new types and populate them with new constants means that object types and object identifiers are inherent in PFL. Furthermore, arbitrary composite objects, including recursive objects, can be modelled. Selectors can be used to represent both nested object values and n-ary relationships. Functional properties of objects can be defined by means of functions defined by one or more equations. Finally, default rules can be formulated as equations which are overidden by more specific equations.

3.2 Computation

PFL is computationally complete by virtue of the fact

```
add_subpart {p,sp,q} =
        if ((composite {p,sp,Any}) == [ ]) [include composite {p,sp,q}] [False]
update_quantity {p,sp,q} =
        if ((composite {p,sp,Any}) != [ ]) [exclude composite {p,sp,Any},include composite {p,sp,q}] [False]
increment_quantity {p,sp,i} =
        if (((composite {p,sp,Any}) != [ ]) and (i > 0)) (update {p,sp,i} (old_q p sp)) [False]
        where
        old_q p sp = third_of_three(head(composite {p,sp,Any}));
        update {p,sp,i} q = [exclude composite {p,sp,q}, include composite {p,sp,q+i}]
```

Figure 2 : Examples for Section 2.6

that functional programming is so. Moreover, a significant class of PFL functions is the class of invertible functions i.e. the class of functions which represent relations. It is useful to compare this sub-language of PFL with Datalog. Clearly, any Datalog EDB predicate can be represented by a PFL selector. Also, we give below a scheme for translating a Datalog rule defining an IDB predicate to a PFL equation defining an invertible function. We then extend this scheme to cater for negative literals. Finally, we indicate when an IDB predicate defined by a set of Datalog rules can be simulated by a single PFL function (up to duplicate elimination).

We consider first an IDB predicate, p, which is defined by a Datalog rule

$p(\bar{x}) \leftarrow q_1(\bar{y}_1), ..., q_n(\bar{y}_n)$

where each q_i is an EDB, IDB or built-in predicate. Without loss of generality we can assume that $\bar{x}$ contains only distinct variables, and we also make the usual assumption that any variables appearing as arguments to a built-in predicate will be instantiated by preceding EDB or IDB predicates [Ull88]. We can express the above rule by a PFL equation of the form

$p\ \{\bar{x}\} = \quad [\{\bar{x}\} \mid w_1 \leftarrow [Any]; ...; w_m \leftarrow [Any]; Q_1; ... ; Q_n]$

where the w_i are the variables of the rule which do not appear in the head, and each Q_i is the generator $\{\bar{y}_i\} \leftarrow q_i\ \{\bar{y}_i\}$ if q_i is an EDB or IDB predicate, or the boolean expression $q_i\ \{\bar{y}_i\}$ if q_i is a built-in predicate. Thus, for example, the Datalog rules

```
anc(X,Y) ← par(X,Z), anc(Z,Y)
p (X,Y)  ← r(X,W), s(W,"u"), t("u",Y), < (W,Y)
```

are expressed by the following PFL equations :

```
anc {x,y} = [{x,y} | z ← [Any]; {x,z} ← par {x,z};
                    {z,y} ← anc {z,y}]
p {x,y}   =  [{x,y} | w ← [Any]; {x,w} ← r {x,w};
                    {w,"u"} ← s {w,"u"};
                    {"u",y} ← t {"u",y}; w < y]
```

We note that the repeated occurrences of variables in these equations imply no ambiguity of meaning since, by the semantics of list abstractions, variable bindings are inherited initially from the left hand side of the equation or from the first generator with the variable in its head, and are then overidden by subsequent occurrences in the heads of generators. We also note that the `anc` function may return duplicate elements, unlike the `anc` IDB predicate.

We now extend our translation scheme to Datalog rules with negative literals in their body, on the assumption that any variable appearing in a negative literal also appears in a positive literal and that positive literals precede negative literals in the body. A negative literal, $\neg q_i(\bar{y}_i)$, translates then into the following expression :

$(q_i\ \{\bar{y}_i\}) == [\ ]$

For example, the Datalog rules

```
flies(X) ← bird(X), ¬ostrich(X)
p(X,Y)  ← r(X,W), s(W,"u"), ¬t("u",X)
```

are expressed by the following PFL equations :

```
flies {x} =   [{x}|{x}←bird {x}; ostrich {x} == [ ]]
p {x,y}   =   [{x,y}|w←[Any]; {x,w}←r {x,w};
                {w,"u"}←s {w,"u"}; t {"u",x} == [ ]]
```

We observe that PFL equations corresponding to non-stratified Datalog rules may give rise to non-terminating computations (c.f. multiple models in Datalog), for example the PFL equations

```
ostrich{x}= [{x}|{x}←bird {x}; flies {x} == [ ]]
flies {x} =   [{x}|{x}←bird {x}; ostrich {x} == [ ]]
```

corresponding to

```
ostrich(x) ← bird(x), ¬flies(x)
flies(x)   ← bird(x), ¬ostrich(x)
```

Thus, our translation scheme is suitable only for stratified Datalog rules.

Finally, we extend our translation scheme to IDB predicates which are defined by a number of Datalog rules. Without loss of generality, we can assume that these rules are *rectified* [Ull88] i.e. all have the same head. We first use the above translation scheme to obtain one PFL equation per rule, ending up with a set of PFL equations with the same left hand side. These equations are then combined into a single equation by appending the lists on their right hand side. For example, the Datalog rules

```
anc(X,Y) ← par(X,Y)
anc(X,Y) ← anc(X,Z), anc(Z,Y)
```

can be represented by the single PFL equation

```
anc {x,y} = append  [{x,y} | {x,y} ← par {x,y}]
                    [{x,y} | z ← [Any]; {x,z} ← anc {x,z};
                     {z,y} ← anc {z,y}]
```

We can make a number of observations here. Firstly, we note the syntactic similarity between the Datalog and PFL definitions. Secondly, we note that the order in which the lists are appended on the right hand side of the PFL equation is significant : we know from the definition of `append` in Section 1 and from the semantics of query evaluation in 2.4 that the first argument to `append` will be evaluated before the second argument and that therefore the first argument represents the base case of the recursion. Thus, the alternative equation

```
anc {x,y} = append  [{x,y} | z ← [Any];
                     {x,z} ← anc {x,z}; {z,y} ← anc {z,y}]
                    [{x,y} | {x,y} ← par {x,y}]
```

would always yield a non-terminating computation. A third observation immediately follows : PFL can simulate only IDB predicates whose defining rules can be fired in a pre-determined sequence. This limitation is analogous to Prolog's top-to-bottom scanning of rules. We conjecture that if PFL were extended with a non-deterministic function `or` [Hen80] which arbitrarily returns one of a number of alternative expressions, it could simulate all stratified Datalog programs.

3.3 Updates

PFL's bulk data is stored in selectors, so thus far we have concentrated on the update of selectors in our design of PFL. We support the `include` and `exclude` operations as built-in functions which can be embedded within PFL expressions whereas PFL's other update operations (declare identifier, insert or delete equation) are not so supported. Thus, in our discussion below we assume a fixed set of types, constants, functions and selectors, and we allow only selectors to be updated.

We define a *database schema* to be a set of selector names, and an *instance* of a database schema to be a mapping of each selector name to a definition. Given a database schema S, we define an *update* to be a partial recursive function from I(S) to I(S), where I(S) is the set of all possible instances of S. It is easy to see that any such update can be expressed in PFL :

Let s_1 ... s_n be the relation names in S. PFL is computationally complete so given any update, U, a function, f, can be written which takes the list [s_1, ... s_n] and

(a)
retrieves the old relation associated with each selector,

(b)
computes the new relations to be associated with the selectors as a list of tuples,

(c)
returns the list [{s_1,old-relation$_1$,new-relation$_1$}, ..., {s_n,old-relation$_n$,new-relation$_n$}].

The function f is invoked from a second function g which effects the update :

```
g  sel-list =
      [{map (exclude s) o, map (include s) n} |
       {s,o,n} ← f sel-list]
```

We observe that, for simplicity, we have not included the requirement for *C-genericity* in our definition of an update above, unlike the treatment in [Abi88b] say. However, our definition could be tightened up accordingly.

4. Implementation

PFL has been implemented in C and runs on a variety of Unix platforms. PFL's parser translates each input statement into a graph in main memory before it is evaluated, in the case of a query, or stored or deleted otherwise. As we stated in 2.4, PFL's evaluator uses the standard technique of graph reduction to evaluate queries and details of this can be found in [Pey87]. So in this section we concentrate on the storage of functions, meta data, and selectors. We also describe how extra "dependency" information is maintained for use during type checking and deletions.

PFL's functions are stored in a B^+ tree [Com79]. This stores records of three fields, of which the first two fields are key fields. Conceptually, the equations defining each function are merged into a tree, after uniformly renaming the variables of each equation to v0, v1, v2 ... in the order of their appearance. These trees are themselves merged into one *equations tree* which represents all equations. For example, the equations :

```
map f [ ]     = [ ]
map f (x:y)   = (f x):(map f y)
```

are stored as shown below, where f is renamed to v0, x to v1, and y to v2 :

```
                        []
                       ----n2
      map      v0     /
root------n0------n1
                     \\
                       ----n3----n4----n5
                          :     v1    v2
```

Here, `root` is the root of the equations tree and each n_i is a unique internal identifier. The leaf nodes n2 and n5 identify the right hand sides of the equations (see below). Each arc $n \xrightarrow{label} m$ in the equations tree is stored as one record <n,label,m> in the B^+ tree.

When evaluating a function application, the equations tree is traversed from left to right and at each node the arc selected is the most specific match for the current argument, if any. For example, when evaluating:

map ((+) 1) [ ]

the rhs identified by n2 is selected, with v0 bound to ((+) 1), and when evaluating:

map ((+) 1) [1,2,3]

the rhs identified by n5 is selected, with v0 bound to ((+) 1), v1 to 1 and v2 to [2,3]. With respect to the right hand sides of equations, these are stored as a number of records of the form <leaf-node, count, instruction>, where `instruction` recreates a part of the right hand side and `count` indicates the sequencing of these instructions. The right hand side is thus reconstructed as a main-memory graph by retrieving all records of the form <leaf-node,*,*>, ordering these records by their second component, and executing their third components in that order.

The types of all identifiers (whether constructors, selectors, or other functions) are stored identically to equations, in this case in a *declarations tree*. This tree is only one arc deep. Each arc is labelled with an identifier and each right hand side stores the type of that identifier. These types are retrieved in the same way as equation right hand sides.

The relations associated with selectors are stored in a separate extendible hashing file [Enb88]. A key is calculated for each tuple which identifies the page in which that tuple is to be stored. The key is calculated by hashing the atomic sub-components of the tuple to integer values and splicing the results. For example, the key for the tuple {Jim,10,Bob} is calculated as follows, under the assumption that an integer comprises 32 bits :

hash(Jim) = $x_1 \ldots x_{32}$
hash(10) = $y_1 \ldots y_{32}$
hash(Bob) = $z_1 \ldots z_{32}$
----
key({Jim,10,Bob}) = $x_1 y_1 z_1 \ldots x_{32} y_{32} z_{32}$

At any point in time, the hash file will comprise 2^i pages, for some i, and the page in which a tuple is stored is identified by the first i bits of its key. For example, if i = 4 then the tuple {Jim,10, Bob} is stored in page $x_1 y_1 z_1 x_2$.

When retrieving tuples, the search pattern may include the wildcard value Any. In this case, the search pattern is hashed to a *set* of keys which identify a set of pages to be searched. The specified components of the search pattern are hashed to integers, as before, but now a key is generated for every possible integer to which the unspecified components might have hashed. For example, the keys identified by the search pattern {Jim,Any,Bob} are found as follows :

hash(Jim) = $x_1 \ldots x_{32}$
hash(Bob) = $z_1 \ldots z_{32}$
----
keys({Jim,Any,Bob}) = $x_1 0 z_1 \ldots x_{32} 0 z_{32}$
$x_1 0 z_1 \ldots x_{32} 1 z_{32}$
: : :
$x_1 1 z_1 \ldots x_{32} 1 z_{32}$

Thus, if i = 4, two pages are searched, $x_1 0 z_1 x_2$ and $x_1 1 z_1 x_2$.

Our dependency information is also maintained in the hash file and records the appearance of user-declared identifiers (functions and constructors) in declarations, equations and relations.

With respect to functions, for each equation e which defines a function f, we store a record of the form <f,defined-by,e>, and for each equation e which references a function f, we store a record of the form <f,referenced-by,e>. For example, the second equation for `map` gives rise to two dependency records, <map,defined-by,n5> and <map,referenced-by,n5>. We note that dependency records do not conflict with selector tuples in the hash file since the former contain the internal identifiers `defined-by` or `referenced-by`, and the latter do not.

Whenever the insertion or deletion of an equation for a function f causes the type of f to change, the equations which reference f are located, and the types of the functions which *they* define are reinferred. This process may cause further cascading of type inference, but will in any case always terminate (since the database contains only a finite number of functions and only supports finite types). In our experience, little cascading occurs in practice. We note that since selectors are first order, and since tuples are fully evaluated before being inserted into selectors, no functions appear in selectors and so selectors do not participate in such cascades.

As with functions, reference information is also maintained in the hash file which records the appearance of *constructors* in equations, relations, and declarations (for metalevel constructors i.e. types). For each equation or declaration e which references a constructor c, a record of the form <c,referenced-by,e> is stored, and for each selector s which references a constructor c, a record of the form <c,referenced-by,s,count>. In this second record, the `count` field is updated according to tuple insertions and deletions. A constructor can be deleted only if it appears in no dependency records. An attempt to delete a constructor which is currently "in use" causes a message to be displayed detailing the equations and selectors which reference that constructor.

5. Conclusions

In this paper we have introduced a persistent functional language called PFL which adopts a functional as opposed to logic-based approach to deductive databases. The key contributions of the language are its adaptation of functional programming to the needs of deductive databases, its incremental updates and type checking, and its support of selectors.

PFL has all the expressiveness of a functional programming language : higher-order functions, static type checking, lazy evaluation, and support for user-defined types and constants. However, PFL also supports extensionally and intentionally defined relations. Despite this increased expressiveness, querying remains purely functional, unless the update functions `include` and `exclude` are invoked.

Our approach to reconciling the advantages of deterministic computations on the one hand and relational manipulation of data on the other, is completely in contrast to the alternative approach of extending Datalog with rewrite rules for function symbols, in [Gru89] for example. Our aim in this paper has been to argue the case for our functional approach which we believe is at least as promising as the Datalog-based alternative.

There are of course aspects of PFL which need further attention. For a start, our update functions operate by side effect - of necessity since they act at the object level - which makes them quite cumbersome to specify. An alternative, possibly more attractive, approach would be to provide a purely functional update language at the meta level. Secondly, we do not at the moment support system-generated identifiers so the user has to think of a new name for every new object he introduces : a `generate_new` built-in function could easily be provided for this purpose. This function would also be a useful aid in database restructuring, c.f. the generation of object identifiers in [Abi89], and its invocation could be functionally determined, c.f. the use of function symbols for this purpose in [Kif89]. Thirdly, encoding semantic integrity constraints within update methods has the disadvantage that it is possible to specify mutually inconsistent constraints, so we are currently exploring the alternative of global semantic integrity constraints. Lastly, we are also considering equipping PFL with a higher-level "object-oriented" interface.

Acknowledgements

The work described in this paper is supported by the U.K. Science and Engineering Research Council (grant no. GR/G 19596). Alex Poulovassilis also holds a Postdoctoral Fellowship from the Council.

References

[Abi88a] Abiteboul S. and Grumbach S. *COL : A Logic-Based Language for Complex Objects*, in Advances in Database Technology (EDBT 88), LNCS 303, Springer-Verlag, 1988.

[Abi88b] Abiteboul S. and Vianu V. *Procedural and Declarative Database Update Languages*, Proc. ACM PODS Conference, 1988.

[Abi89] Abiteboul S. and Kanellakis P.C. *Object Identity as a Query Language Primitive*, Proc. ACM SIGMOD Conference, 1989.

[Alb85] Albano A., Cardelli L. and Orsini R. *Galileo: A Strongly-Typed, Interactive Conceptual Language*, ACM TODS 10(2), 1985.

[Ban87] Bancilhon F. *et al. FAD, A Powerful and Simple Database Language*, Proc. 13th VLDB Conference, 1987.

[Bat88] Batory D.S., Leung T.Y. and Wise T.E. *Implementation Concepts for an Extensible Data Model and Data Language*, ACM TODS 13(3), 1988.

[Bee88] Beech D. *A Foundation of Evolution from Relational to Object Databases*, in Advances in Database Technology (EDBT 88), LNCS 303, Springer-Verlag, 1988.

[Bun82] Buneman P., Frankel R.E. and Nikhil R. *An Implementation Technique for Database Query Languages*, ACM TODS 7(2), 1982.

[Car84] Cardelli L. *Basic polymorphic type checking*, Science of Computer Programming, 8(2), 1987.

[Car85] Cardelli L. *Amber* in Combinators and Functional Programming Languages, G.Cousineau *et al.* (eds.), LNCS 242, Springer-Verlag, 1985.

[Car88a] Cardelli L. *A Semantics of Multiple Inheritance*, Information and Computation, 76, pp 138-164, 1988.

[Car88b] Cardelli L. *Types for Data-Oriented Languages*, in Advances in Database Technology (EDBT 88), LNCS 303, Springer-Verlag, 1988.

[Che89] Chen W., Kifer M. and Warren D.S., *Hilog as a platform for database languages* in [DBP89]

[Cer90] Ceri S., Gottlob G. and Tanca L. *Logic Programming and Databases*, Surveys in Computer Science, Springer-Verlag, 1990.

[Chi90] Chimenti D. *et al. The LDL System Prototype*, IEEE Trans. on Knowledge and Data Engineering, 2(1), 1990.

[Com79] Comer, D. *The Ubiquitous B-tree.* ACM Comp. Surveys 11(2) 1979.

[Day87] Dayal U. *et al.*, *Simplifying Complex Objects: The PROBE Approach to Modelling and Querying Them*, Workshop on the Theory and Applications of Nested Relations and Complex Objects, Darmstadt, April 1987.

[DBP89] Proc. 2nd International Workshop on Database Programming Languages, Oregon.

[Enb88] Embody R.J. and Du H.C. *Dynamic Hashing Schemes*, ACM Comp. Surveys, 20(2), 1988.

[Gal84] Gallaire H., Minker J. and Nicolas J-M. *Logic and Databases : a Deductive Approach,* ACM Comp. Surveys, 16, 1984.

[Gra88a] Gray P.M.D. and Lucas R.J. (eds.) *Prolog and Databases : Implementations and New Directions*, Ellis Horwood, 1988.

[Gra88b] Gray P.M.D., Moffat D.S. and Paton N.W. (eds.) *A Prolog Interface to a Functional Data Model Database*, in Advances in Database Technology (EDBT 88), LNCS 303, Springer-Verlag, 1988.

[Gru89] Grumbach S. *Integration of functions defined with rewriting rules in Datalog*, Proc. DOOD 89.

[Hei88] Heiler S. and Zdonik S. *Views, Data Abstraction and Inheritance in the FUGUE Data Model*, in Advances in Object-Oriented Database Systems, LNCS 334, Springer-Verlag, 1988.

[Hen80] Henderson P. *Functional Programming*, Prentice Hall, 1980.

[Hor90] Horsfield T., Bocca J. and Dahmen M. *Megalog User Guide*. ECRC, 1990.

[Kif89] Kifer M. and Wu J. *A logic for object-oriented logic programming*, Proc. ACM PODS Conference, 1989.

[Kul86] Kulkarni K.G. and Atkinson M.P. *EFDM : Extended Functional Data Model,* Computer Journal, 29(1), 1986.

[Mor89] Morrison R. *et al.*, *The Napier88 Reference Manual*, Universities of Glasgow and St.Andrews, PPRR-77-89.

[Myc84] Mycroft A. and O'Keefe R.A. *A Polymorphic Type System for Prolog*, Artificial Intelligence, 23, 1984

[Nik85] Nikhil R.S., *Practical Polymorphism*, in Functional Programming Languages and Computer Architectures, LNCS 201, Springer-Verlag, 1985.

[Pey87] Peyton-Jones, S.L. *The Implementation of Functional Programming Languages,* Prentice Hall, 1987

[Pou90] Poulovassilis A. and King P. *Extending the Functional Data Model to Computational Completeness*, in Advances in Database Technology (EDBT 90), LNCS 416, Springer-Verlag, 1990.

[Shi81] Shipman D.W. *The Functional Data Model and the Data Language DAPLEX*, ACM TODS 6(1), 1981.

[Sma90] Small C. *A Persistent Functional Programming Environment*, Research Report PFL/2, Birkbeck College. To appear.

[Smi83] Smith J.M, Fox S. and Landers T. *ADAPLEX Rationale and Reference Manual*, CCA, CCA-83-08.

[Ull88] Ullman J.D. *Principles of Database and Knowledge-Base Systems*, Computer Science Press, 1988.

[Xu89] Xu J. and Warren D.S. *A Type Inference System for Prolog*, Proc. 5th International Logic Programming Conference, 1989.