A Domain-theoretic Approach to Integrating Functional and Logic Database Languages

Alexandra Poulovassilis
Department of Computer Science,
King's College London,
Strand, London WC2R 2LS

Abstract
The advantages of logic languages with respect to search-based computation are well-understood, while the advantages of functional languages with respect to deterministic computation are becoming increasingly recognised. It is therefore natural to investigate the development of languages which reconcile the two paradigms. As a contribution to this effort, we extend an existing functional database language called PFL with sets as first class objects. The resulting language subsumes Datalog in the sense that any set of Datalog rules can be translated into a set of PFL equations with the same semantics. Since functional and logic database languages can be considered as proper sub-languages of PFL, well-known optimisation techniques from both can usefully be employed (for example lazy evaluation for recursive functions and bottom-up evaluation techniques for recursive predicates).

We motivate our work by reviewing the respective advantages of functional and logic programming for computation, data manipulation and data modelling. An overview of the previous version of PFL is presented and the syntax of this language is then extended to incorporate sets. We show how the Plotkin powerdomain construction can be used to assign meaning to set expressions and we give a denotational semantics for the extended language. To illustrate its expressiveness, we show how Datalog rules can be expressed as PFL functions. We discuss the optimisation of these functions. We also show how integrity constraints can be defined, and describe how a particular constraint enforcement technique developed for logic databases can be adopted by PFL.

1. Introduction
Recently we have been investigating the implementation of deductive databases whose inference is based on functional programming rather than the more common approach of logic programming. In particular, we introduced in [29, 33] a functional database language called PFL which supports higher-order functions, recursive data types, and the persistence of factual and procedural information, both in the form of equations. A unique feature of PFL is that bulk data is stored using a class of extensionally-defined updateable functions called selectors. Selectors are invertible, in the sense that they simulate extensional predicates. We showed in [29] how functions can be written which draw inferences from selectors and which are also invertible. In particular, we examined the expressiveness of these invertible functions and showed how any stratified Datalog IDB predicate can be simulated by a such a function, subject to the proviso that there exists a pre-determined order of firing the rules which define the predicate.

The main aim of the present paper is to remove this restriction and hence to endow PFL's invertible functions with at least the full expressiveness of Datalog. The result is a language which combines the advantages of functional programming with respect to deterministic computations with the advantages of logic programming with respect to search-based computations. This was not achieved in the original language since in general rules can be defined for which no fixed firing sequence generates all the provable facts. An example of such a set of rules is the following (from [22]):

\[
\begin{align*}
p(a,b) & \leftarrow \\
p(c,b) & \leftarrow \\
p(X,Z) & \leftarrow p(X,Y), p(Y,Z) \\
p(X,Y) & \leftarrow p(Y,X) \\
p(a,c) \text{ is provable from these rules, but not by any depth-first search strategy with a fixed ordering of trying the rules, since the last two rules have equally general heads.}
\end{align*}
\]

In order to attain at least the expressiveness of Datalog, we introduce sets into PFL. This allows us to express rules without any implied order of firing. Our sets are first class objects: they can appear arbitrarily nested...
within expressions and can themselves contain arbitrarily
nested expressions; also, functions can be written which
take sets as arguments and return sets as results. In par-
cular, functions can be defined over sets to simulate all
Datalog_{fun+neg} predicates. We term such functions inten-
tional selectors. As we would expect, the new PFL is in
fact more expressive than Datalog since aggregation and
counting operations can be defined over sets.

A secondary aim of the paper is to show how semantic
integrity constraints can be defined and efficiently
enforced in the extended PFL. An immediate conse-
quence of supporting sets as first class objects is that
approaches developed for the efficient enforcement of
constraints in logic databases can be adopted. As an
example of this we illustrate how the method of Lloyd
and Topor [21] can be used. In fact, our integrity con-
straints are more expressive than those of logic databases
since cardinality and other aggregation constraints are
supported.

Since PFL aims to combine the respective advantages of
logic and functional programming, we briefly review
these in the remainder of this introductory section.

One major advantage of functional languages over logic
languages is that they are higher order and so all expres-
sions are first class objects. Higher-order functions can
be defined which abstract out recursion patterns over
recursive data structures, and which can then be used in
the definition of other functions which do not include
explicit recursion. A second advantage is that the deter-
nomistic semantics of functional evaluation can be
exploited for the representation of default rules (see
[28]). In contrast, logic languages are monotonic and
hence require the introduction of extra logical constructs
such as the negation as failure rule [14]. Thirdly, by
employing lazy evaluation, infinite processes and data
structures can be handled. Finally, deterministic computa-
tions are expressed more succinctly by nested function
applications than by joins of predicates over common
variables.

Conversely, the main advantage of logic languages is that
search-based computations are supported directly, while
in functional languages they must be simulated determin-
istically. Also, unlike functions, predicates are invertible
in the sense that they can, theoretically, be used with any
of their arguments uninstantiated, and thus predicates
may be more versatile than functions. Finally, in the con-
text of deductive databases both facts and derivation rules
are often expressed more naturally in terms of single
predicates than in terms of functions and their inverses.

The motivation behind the development of PFL has been
to combine within one uniform semantic and operational
framework the respective advantages of these two para-
digms for deductive databases. In particular, the
extended PFL exhibits all of the features of functional
languages described above. Furthermore, since sets are
now first class objects, functions can be defined which
play a role similar to intentional predicates. Any set of

Datalog rules (including those incorporating negation and
function symbols) can be translated into an equivalent set
of PFL equations. Finally, all of the query optimisation
strategies developed for logic languages - including
optimisation techniques developed for integrity constrains - can now be transferred to a functional context.

The outline of the remainder of the paper is as follows.
In section 2 we briefly describe the features of our original
language, and in particular its type system and the
way in which user defined functions are specified. Sec-
tion 3 introduces the syntax and semantics of the
extended PFL. We first define a kernel functional
language and give the denotational semantics of this
language. We then add set expressions and extend the
denotational semantics accordingly. Finally, we describe
how bulk data is stored in a class of extensionally-defined
set-valued functions. Section 4 then discusses the
evaluation of expressions in the extended PFL. Section 5
considers the expressiveness of the language. We give a
scheme for translating rules in Datalog_{fun+neg} into equa-
tions involving set expressions, and then give an optimi-
sation scheme for these equations. We also show how
higher order functions can be applied to sets, thus allow-
ing computations to be defined which would require
extra logical predicates such as "set-of" in a logic
language. Finally, we discuss the definition and efficient
enforcement of integrity constraints. Section 6 gives our
conclusions and a brief comparison with related work.

2. Overview of PFL

PFL has a Milner style polymorphic, strong, static type
system [10] that supports a number of built-in types and
provides facilities for the introduction of new user-
defined types and constants. Functions are defined incre-
mentally through the insertion and deletion of equations.
The types of functions are inferred incrementally in the
face of such updates. Function evaluation is lazy,
thereby permitting computation with unbounded data
structures. Types, constants and equations can be defined
and removed at any stage during the life-time of the data-
bases subject to certain obvious restrictions (e.g. a con-
stant cannot be introduced for an as-yet unspecified type).
All types, constants and equations are stored in the data-
bases. We briefly review the structure of the type system
in 2.1, the system defined functions in 2.2, and the
specification of user defined functions in 2.3.

2.1 The Type System

PFL's type system comprises three layers (c.f. [11]).
Firstly, there is the meta-level type Type which is the set
of all types; secondly, there are the object-level types,
both built-in and user-declared, which are regarded as
meta-level values of type Type; and finally there are the
values corresponding to each object-level type. The
built-in types are Bool, Num, Str, and Char which are
populated by booleans, strings, numbers and characters
respectively. In addition, for every type t there is a dis-
tistinguished constant Any,. We usually write this constant without its subscript since this can be inferred from context. Certain system-defined functions when applied to Any will result in a run-time error e.g. the arithmetic functions (c.f. division by zero).

New types and values are declared using commands of the form

```
"type" constant type-variables "::
"value" constant "::" type "::"
```

These constants are sometimes termed constructors (27) since they construct values of the indicated types. For example, Person, Product and List types together with constructors for these types are defined as follows, where identifiers starting with an uppercase letter are constructors (meta-level or object-level), whilst identifiers starting with a lowercase letter are variables:

```
type Person;
type Prod2 a b;
type List a;
value John :: Person;
value Tuple2 :: a-b->(Prod2 a b);
value [:i : : a->(List a)->(List a);
value [] :: List a;
```

Thus, Person is a type (or meta-level value), Prod2 t1 t2 is a type for all types t1 and t2, and List t is a type for all types t. Similarly, John is a value of type Person, and Tuple2 v1 v2 is a value of type Prod2 t1 t2, assuming v1 and v2 are of type t1 and t2 respectively. Zero argument types (e.g. Person) and values (e.g. John) can be regarded as object types and object values respectively. For syntactic ease we adopt a number of abbreviations for types and values:

```
List t = [t]
Prodn t1 ... tn = (t1,...,tn)
(:) v1 v2 = (v1:v2)
v1:...(vn:[])... = [v1,...,vn]
Tuple n v1 ... vn = (v1,...,vn)
```

In preparation for adding sets to PFL, we also assume the availability of a polymorphic set type Set a, and we allow the syntax [t] in lieu of Set t for any type expression t (analogously to [t] for List t).

2.2 System Defined Functions

PFL provides a number of system defined functions, including the usual arithmetic (+, -, *, div, mod), relational (=, !=, <, >, >=, <=, and, or, not) and list processing operators (+++, head, tail). In addition, the operator ?= (pronounced "matches") acts on values containing Any. In particular, ?= determines whether its left operand is identical to its right operand except with respect to occurrences of Any in the former. More precisely, for each n-ary constructor C we have that

(C x1 ... xn) ?= (C y1 ... yn) = True and
(x1 ?= y1) and ... and (xn ?= yn)

while for any other values x and y we have that

x ?= y = x = Any

(we observe here that the first = symbol in a PFL equation denotes definition while thereafter = denotes equality). For example, [Any,2,Any] ?= [1,2,3] and [1,2,3] ?= [Any,2,Any] evaluate to True and False respectively. We note that ?= is reflexive, anti-symmetric and transitive i.e. it is a partial ordering.

2.3 User Defined Functions

Functions are defined by means of equations, which are introduced by a command of the form

```
"define" lhs "=" expr ";"
```

For example the 3-ary function if and the 2-ary function map (which has the property that map f [x1,...,xn] yields [f x1,...,fxn]) can be defined as follows:

```
define if True x y = x;
define if False x y = y;
define map f [] = [];
define map f (x:y) = (f x):(map f y);
```

The types of these functions are inferred automatically by the system.

A useful query construct supported by PFL, as by most functional languages, is the list abstraction, which is simply syntactic sugar for ordinary function applications (27). An example of the use of the list abstraction is the quick-sort function below (where ++ is the infix append operation):

```
define qsort [] = [];
define qsort (h:t) = (qsort [x | x ← t ; x < h]) ++ [h] ++
(qsort [x | x ← t ; x ≥ h]);
```

3. Extending PFL with Sets

In this section we describe how PFL is extended with sets as first-class objects, and we specify formally the semantics of the resulting language. The formal foundation of any functional language is the λ calculus (of which a detailed account may be found in [19]). Computation in this calculus consists of rewriting expressions to a normal form by a series of syntactic transformations called β and η reductions. A formal theory can be constructed for βη reduction and models of this theory are triples <Λ, •, D>, where D is a domain of continuous functions (we define the terms domain and continuous in 3.1 below), Λ models the definition of functions by λ abstraction and • models function application. The semantics of functional languages are thus usually specified using a denotational
approach [31, 35] which assigns values in D to expressions in the syntax of the language. You can take this route for specifying the semantics of PFL. We begin by reviewing the prerequisite base domain theory in 3.1. In 3.2 we define the syntax and denotational semantics of a kernel functional language into which PFL expressions are translated before evaluation. In 3.3 we add set expressions to this syntax and in 3.4 we extend the denotational semantics accordingly. In 3.5 we extend the kernel further to allow pattern-matching. Finally, in 3.6 we discuss a class of set-valued functions called selectors which are used for the storage and update of bulk data.

Readers who are not familiar with denotational semantics can safely omit sections 3.2, 3.3, 3.4 and 3.5, and read only 3.3 and 3.6. In this case, two essential concepts that are required for 3.3 and 3.6 are that:

- \( 1_D \) is the value of type \( D \) denoting "no information" (resulting perhaps from a non-terminating computation or a run-time error), and

- continuous functions are monotonic and information-preserving (i.e. applying the function to the least upper bound of a sequence of better-defined arguments is equivalent to applying the function to each argument individually and then taking the least upper bound of the results).

3.1 Background on Domains

If \( D \) is a set and \( \sqsubseteq \) is a partial ordering on \( D \), then for any subset \( E \subseteq D \) there is at most one element \( d \in D \) such that \( \forall d' \in D \), \( d \sqsubseteq d' \) if and only if \( \exists d \in E \). If \( d \) exists, it is termed the least upper bound of \( E \), written \( \sqcap E \). The element \( \sqcap \emptyset \) (if it exists) is denoted by \( \bot \) and satisfies \( \sqcap \sqsubseteq d \), \( \forall d \in D \). \( D \) is said to be a domain if it has such a least element and if for every (possibly infinite) sequence of elements \( d_1 \sqsubseteq d_2 \sqsubseteq \ldots \), their least upper bound \( \squplus \{d_1, d_2 \ldots \} \) also exists.

Given two domains, \( D_1 \) and \( D_2 \), a function \( f:D_1 \rightarrow D_2 \) is said to be continuous if, for every sequence \( d_1 \sqsubseteq d_2 \sqsubseteq \ldots \) in \( D_1 \), \( f(\sqcup \{d_1, d_2 \ldots \}) \) is equal to \( \sqcup \{f(d_1), f(d_2), \ldots \} \). In other words, continuous functions preserve least upper bounds. Functions are required to be continuous in order to guarantee the existence of a least fixed point. In particular, given a continuous function \( f:D \rightarrow D \), the least fixed point of \( f \) exists and is given by

\[
\text{Fix}(f) = \bigcup \{ f^n(\bot_D) \mid n \geq 0 \}
\]

where \( f^0(x) = x \) and \( f^i(x) = f(f^{i-1}(x)) \) for \( i > 0 \).

The simplest domain is a flat domain which has the property that if \( d_1 \sqsubseteq d_2 \) then either \( d_1 = d_2 \) or \( d_1 = \bot_D \). It is possible to construct more complex domains from simpler domains. In particular, given domains \( D_1, \ldots, D_n \), four useful domains are the product, sum, coalesced sum and continuous function domains:

The product domain \( D = D_1 \times D_2 \times \ldots \times D_n \) contains tuples of the form \( <d_1, d_2, \ldots, d_n> \) where each \( d_i \in D_i \), and \( \forall i \leq n \) the bottom element is thus \( \langle \bot_{D_1}, \ldots, \bot_{D_n} \rangle \).

The (separated) sum domain \( D = D_1 + D_2 + \ldots + D_n \) is defined as a union:

\[
\{1 \times D_1 \} \cup \{2 \times D_2 \} \cup \ldots \cup \{n \times D_n \} \cup \{\bot_D\}
\]

where \( \bot_D \sqsubseteq d, \forall d \in D, \) and \( <1, d> \sqsubseteq <j, e> \) if the "tags" \( i \) and \( j \) are equal and \( d \sqsubseteq e \). The (coalesced) sum domain \( D = D_1 \oplus \ldots \oplus D_n \) is also a union:

\[
\{1 \times (D_1 - \{\bot_{D_1}\}) \} \cup \{2 \times (D_2 - \{\bot_{D_2}\}) \} \cup \ldots \cup \{n \times (D_n - \{\bot_{D_n}\}) \} \cup \{\bot_D\}
\]

where \( \bot_D \) is defined as for the separated sum. The (coalesced) sum is so called because the least element of each component space becomes identified with the least element of the sum space. This is in contrast to the separated sum where a new least element is introduced into the sum space.

The domain of continuous functions from a domain \( D_1 \) to a domain \( D_2 : D_1 \rightarrow D_2 \) has \( f:D_1 \rightarrow D_2 \), \( f(d) \sqsubseteq g(d) \), \( \forall d \in D_1 \). The least element of \( D \) is thus \( \lambda d.1_D \), the function which returns the least element of \( D \) for every element in \( D_1 \).

The equation defining a domain \( D \) may be recursive. An example is the domain of lists of integers, where \( \text{NIL}_{\text{INT}} \) is a distinguished constant:

\[
\text{LIST}_{\text{INT}} = (\text{INT} \times \text{LIST}_{\text{INT}}) + \{\text{NIL}_{\text{INT}}\}
\]

The solution of such recursive domain equations is discussed by Stoy [35].

3.2 Denotational Semantics of the PFL Kernel

Functional languages typically have a simple kernel into which expressions in the full syntax of the language are translated prior to evaluation. The semantics of the full language are thus derived from the semantics of the kernel. PFL follows this pattern and the syntax of its kernel is as follows:

\[
\text{expr} = \text{id} | \text{const} | \text{expr} \text{ expr} | \text{fun} (\text{expr}, \ldots, \text{expr}) | \lambda \text{id}. \text{expr} | \text{fix} \text{ expr} | \text{expr} \text{ where id = expr}
\]

In particular, the "fix" construct allows recursive functions to be defined. For example, the equation

\[
\text{fact} \ x = \text{if} \ (x = 0) \ 1 \ (x \times \text{fact} \ (x-1))
\]

is syntactically transformed into the following non-recursive definition:

\[
\text{fact} = \text{fix} \ \lambda f. \lambda x. \text{if} \ (x = 0) \ 1 \ (x \times \text{f} \ (x-1))
\]

Mutually recursive functions are handled by packaging them into a single non-recursive tuple similarly.
For our purposes, we assume that expressions are assigned values in the following semantic domain, $D$:

$$D = W + \text{Bool} + \text{Atom} + [D \rightarrow D] + (D \times D) + \ldots + \text{LIST}$$

Here, $\text{Bool}$ and $\text{Atom}$ are flat domains representing booleans and other constants respectively, while $\text{LIST}$ denotes an infinite number of domains such that each $\text{LIST}$ consists of lists whose elements are drawn from domain $I$, e.g.

$$\text{LIST}_{\text{Atom}} = (\text{Atom} \times \text{LIST}_{\text{Atom}}) + \{\text{NIL}_{\text{Atom}}\}$$

$$\text{LIST}_{\text{Atom} \times \text{Atom}} = ((\text{Atom} \times \text{Atom}) \times \text{LIST}_{\text{Atom} \times \text{Atom}}) + \{\text{NIL}_{\text{Atom} \times \text{Atom}}\}$$

In practice of course atoms are partitioned into separate domains in most functional languages (one exception is LISP) and lists are constrained to be homogeneous by the type checker. Type checking in PFL was discussed in [29] so we consider it no further here. We merely note that in the domain equation for $D$ above, $W$ contains a single element, "error", which is the value of any incorrectly typed expression.

Defining the denotational semantics of a language consists of identifying the syntactic categories of the language, and setting up semantic functions between these and the domain $D$ (see [31,35]). The three syntactic categories of our kernel are Con, Idc and Env = [Idc→D], from which constructors, variables and environments are drawn, respectively. In particular, an environment $p$ is a function that assigns to each variable in Idc a value in the semantic domain $D$. Two semantic functions are required to map expressions to values in $D$: the function $K:([\text{Con} \rightarrow D]$ maps each constructor to its predefined value, and the function $E:([\text{Exp} \rightarrow \text{Env} \rightarrow D]$ maps any expression to a value, for a given environment.

The definition of $E$ is given below where, by convention, all expressions enclosed in square brackets, $\llbracket$ and $\rrbracket$ are in the syntax of the language whose semantics are being defined i.e. the kernel. All other expressions are in the defining notation, in this case some model of the $\lambda$ calculus that supports $\lambda$ abstraction (whence the $\lambda$ on the right hand side of the third equation) and function application (whence the $\cdot$ on the right hand side of the fourth equation). The notation $p[d/id]$ denotes the environment obtained by extending $p$ to map the variable id to the value $d$. As we would expect, the semantic function $E$ mapping a kernel functional language to a model of the $\lambda$ calculus is almost trivial. Specifically, variables are mapped in the value assigned in them by $p$, constants are mapped to the value assigned to them by $K$, $\lambda$ abstractions are mapped to $\lambda$ abstractions, applications are mapped to applications, tuples are mapped to tuples, "fix" results in a fixpoint computation, and "where" clauses result in the extension of the environment with a new identifier:

$$E \llbracket \text{id} \rrbracket p = p[\text{id}]$$

$$E \llbracket \text{const} \rrbracket p = K[\text{const}]$$

$$E \llbracket \lambda \text{id} . \text{expr} \rrbracket p = \lambda d. E \llbracket \text{expr} \rrbracket p[d/id]$$

$$E \llbracket \text{expr} \text{expr}_2 \rrbracket p = (E \llbracket \text{expr} \rrbracket p) \bullet (E \llbracket \text{expr}_2 \rrbracket p)$$

$$E \llbracket (\text{expr}_1, \ldots, \text{expr}_n) \rrbracket p = <E \llbracket \text{expr}_1 \rrbracket p, \ldots, E \llbracket \text{expr}_n \rrbracket p>$$

$$E \llbracket \text{fix} \text{expr} \rrbracket p = \text{FIX}(E \llbracket \text{expr} \rrbracket p)$$

$$E \llbracket \text{expr}_1 \text{where id = expr}_2 \rrbracket p = E \llbracket \text{expr}_1 \rrbracket p[(E \llbracket \text{expr}_2 \rrbracket p)[d/id]]$$

### 3.3 Adding Set Abstractions

We are now ready to add set abstractions to the kernel functional language. We need to extend first the syntax of the language to allow set expressions, then the semantic domain to include values for set expressions, and finally the semantic function $E$.

Set abstractions are similar to list abstractions in that they construct new sets from existing sets just as list abstractions construct new lists from existing lists. As well as set abstractions, we also introduce empty sets, singleton sets and set unions. The resulting language syntax is as follows:

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr</td>
<td>id</td>
</tr>
<tr>
<td>\text{qualifier}</td>
<td>qualificr</td>
</tr>
<tr>
<td>\text{filter}</td>
<td>expr</td>
</tr>
<tr>
<td>\text{generator}</td>
<td>id</td>
</tr>
</tbody>
</table>

We also allow the notation $\{s_1, \ldots, s_n\}$ in lieu of $\{s_1\} \cup \ldots \cup \{s_n\}$. Two examples of set expressions are:

$$\{(\text{John}, \text{Jane}), (\text{Jack}, \text{Jane})\}$$

$$\{n \mid n \in \text{nat} \& (n \bmod 2) = 0\}$$

where the second expression denotes the set of even numbers in nat. In 3.5 below we extend this syntax to allow arbitrary patterns on the left-hand side of generator expressions, so that the following set abstractions respectively return (i) all the tuples in the set parents, (ii) all the tuples in the set parents which have Bill as their first component, and (iii) the singleton set \{Bill, Mary\} if this tuple is in the set parents, and the empty set otherwise:

$$\{(x, z) \mid (x, z) \in \text{parents}\}$$

$$\{(\text{Bill}, z) \mid (\text{Bill}, z) \in \text{parents}\}$$

$$\{(\text{Bill}, \text{Mary}) \mid (\text{Bill}, \text{Mary}) \in \text{parents}\}$$

The semantic domain $D$ of 3.2 is extended to include an infinite number of domains $\text{SET}_1$ such that each $\text{SET}_1$ consists of sets whose elements are drawn from domain $I$:

$$D = W + \text{Bool} + \text{Atom} + [D \rightarrow D] + (D \times D) + (D \times D) + \ldots + \text{LIST}_1 + \text{SET}_1$$

Three main orderings have been proposed for set
domains in the literature: the Hoare, Smyth and Plotkin orderings (the last of these is also known as the Egli-Milner ordering), and a comparison of them is given in [314]. Of these, the Hoare and Plotkin orderings are both candidates for our purposes since they consider larger sets to be better defined than smaller sets. The Hoare ordering permits sets to contain an infinite number of non-1 elements whereas under the Plotkin ordering sets are either finite or contain L. We have chosen to use the Plotkin construction rather than the Hoare one since, as we will see below, this allows us to map from sets to lists and thereby to fully integrate sets into PFL. This is not possible with the Hoare ordering (which was adopted by Silbermann and Jayaraman in their integration of functional and logic programming [32]) since such a mapping would not be continuous. More specifically, the Plotkin ordering, \( \subseteq \text{int} \), on two sets S and T whose elements are drawn from a flat domain I is defined as follows:

\[
S \subseteq \text{int} T \text{ iff } \forall s \in S \exists t \in T. s \subseteq t \text{ and } \forall t \in T. \exists s \in S. s \subseteq t
\]

(The definition for non-flat I is more intricate and can be found in [31].) Thus, the least element is \( \{I\} \) and the empty set is not part of the domain. This is because the Plotkin powerdomain is traditionally used to model non-determinism, where all computations have some result even if this result is non-termination. In our case we wish to model set-valued functions and so the empty set is meaningful. The structure of \( \text{SET}_I \) is thus the coalesced sum (see 3.1) of the Plotkin powerdomain with a singleton domain representing the empty set of type I.

For example, the left-hand part of Figure 1 below shows the structure of \( \text{SET}_{\text{bool}} \). Computation over this domain commences at the least element \( \{\text{false}\} \) and proceeds either to the empty set or to a non-empty set. In the latter case the set may contain the element \( \{\text{true}\} \), representing the possibility that more elements will be added to the set or that a non-terminating computation will occur. When no more elements can be added to a set, computation terminates with a set not containing \( \{\text{false}\} \).

To stay within a purely functional formalism (and thereby continue to benefit from the advantages of functional programming we discussed in the introduction), there are two key requirements for operations over sets: determinism and continuity. To retain determinism we assume a total ordering, \( < \), on all elements of Atom and Bool except \( \perp \) and \( \bot \), which in turn induces a total ordering on all finite elements of I (except the functions in \( [D \rightarrow D] \) of course). We then use an operator

\[
\text{set-to-list}_I : \text{SET}_I \rightarrow \text{LIST}_I
\]

which maps elements of \( \text{SET}_I \) to elements of \( \text{LIST}_I \) (we often write \( \text{set-to-list} \) without its subscript since this can be inferred from context). It does this by taking

- any set containing \( \perp \) to \( \perp \)-LIST, and
- any other set to the (unique) list whose components are precisely those of the set and appear in the order of

< without duplicates.

For example, Figure 1 below shows the \( \text{SET}_{\text{bool}} \) and \( \text{LIST}_{\text{bool}} \) domains and the \( \text{set-to-list}_{\text{bool}} \) mapping between them (which assumes that \( \text{Any}_{\text{bool}} < \text{False} < \text{True} \)). We note that \( \text{set-to-list} \) is continuous, as required.

### 3.4 Denotational Semantics of Set Expressions

We now extend the semantic function \( E \) of 3.2 to map set expressions into sets. Before doing so we require three standard (and continuous) operators from powerdomain theory [31]. The \( \text{singleton} \) operator constructs singleton sets in \( \text{SET}_I \) from elements in I:

\[
\{\_\} : I \rightarrow \text{SET}_I
\]

The \( \text{union} \) operator constructs new sets from pairs of sets:

\[
_\_ : (\text{SET}_I \times \text{SET}_I) \rightarrow \text{SET}_I
\]

Finally the \( \text{+} \) operator distributes functions of type \( I \rightarrow \text{SET}_I \) over sets of type \( \text{SET}_I \) and takes the union:

\[
_+ : (I \rightarrow \text{SET}_I) \rightarrow (\text{SET}_I \rightarrow \text{SET}_I)
\]

In particular, for any function \( F : I \rightarrow \text{SET}_I \), the function \( F_+ \) defined as follows:

\[
F_+ S = \{d \mid d = \bot \text{ or } (F d) \mid d \in S\}
\]

The extended semantic function \( E \) assigns meanings to set expressions as follows, where we have explicitly subscripted set expressions with the type of their components where necessary:

\[
E[I] = \emptyset
\]

\[
E[\text{expr}] = (E[\text{expr}])_I
\]

\[
E[\text{expr} \cup \text{expr}_2] = (E[\text{expr}_1])_I \cup (E[\text{expr}_2])_I
\]

\[
E[\text{expr} \mid \text{filter} \text{-expr} \text{ & rest}]_I = (\lambda d. E[(\text{expr} \mid \text{rest})]_I \mid d \notin \text{filter}) + (E[\text{expr} \mid \text{rest}])_I
\]

Thus, singleton expressions map to singleton sets, union expressions map to unions of sets, filters map to conditions, and generators \( id \in \text{set-expr} \) map to functions which generate bindings for the variable id by being distributed over set-expr.

We conclude this section by noting that the \( = \) operator of section 2.2 can be extended to match set terms in a number of ways. Three possibilities are \( = \) and \( = \), which simulates the Hoare ordering on sets (i.e. every element in the left-hand operand of \( = \) must match some element of the right-hand operand):

\[
S_1 =_H S_2 = \{p \mid p \in S_1 \land v \in S_2 \land p =_H v\} = S_1
\]

\( =_S \), which simulates the Smyth ordering (i.e. every element of the right-hand operand must be matched by some element of the left-hand operand):

\[
S_1 =_S S_2 = \{v \mid v \in S_2 \land p \in S_1 \land p =_S v\} = S_2
\]
and $\equiv$, which simulates the Plotkin ordering (i.e., every element of the left-hand operand must match some element of the right-hand operand, and every element of the right-hand operand must be matched by some element of the left-hand operand):

$$S_1 \equiv S_2 \iff \{p \mid p \in S_1 \land 
\forall v \in S_2 \land p \equiv v \} = S_1 \quad \text{and} \quad \{v \mid v \in S_2 \land p \in S_1 \land p \equiv v \} = S_2$$

### 3.5 Extending the Kernel to the Full Language

In common with most functional programming languages, PFL allows patterns (i.e., expressions in constants and variables) in several places where the kernel syntax allows only variables. These are: lambda abstractions, left hand sides of equations, and left hand sides of generators in list and set abstractions. Peyton Jones [27] discusses how such features can be translated into a kernel language, although not for patterns in set abstractions. So for completeness we give the semantics of this feature. To do so we need to add one more equation to the definition of $E$ in order to take care of generators with patterns i.e., generators of the form $c \, \text{pat}_1 \ldots \text{pat}_n \, \in \text{set-expr}$, where $c$ is a constructor with $n \geq 0$ arguments which are themselves variables or further patterns. The effect of such generators is to iterate over only those elements of set-expr that match the pattern $c \, \text{pat}_1 \ldots \text{pat}_n$

In our semantic equation for this feature below we assume the availability of pattern-matching lambda abstractions [27] i.e., expressions of the form $\lambda \, \text{pat} \, \text{expr}$ which when applied to an argument, $\text{arg}$, substitute in $\text{expr}$ the variables of $\text{pat}$ by constants from $\text{arg}$ if $\text{pat}$ matches $\text{arg}$ and otherwise return the constant error $E$ (of section 3.2):

$$E \{ \text{expr} \mid c \, \text{pat}_1 \ldots \text{pat}_n \, \in \text{set-expr} \land \text{rest} \} \l p = (E \{ E \mid \lambda \, \text{pat}_1 \ldots \text{pat}_n \, \text{expr} \land \text{rest} \} \l p) \uplus \uplus \text{error} \quad \text{then} \quad (E \{ E \mid \lambda \, \text{pat}_1 \ldots \text{pat}_n \, \text{expr} \land \text{rest} \} \l p) \uplus \uplus \text{error}$$

### 3.6 Bulk Data

PFL’s bulk data is stored in a class of functions called selectors. In [29, 33] we defined selectors as list-valued functions defined by an equation of the form...
where the relation was a list without duplicates and where updates to the selector could change the order of its elements. Having now incorporated sets into PFL we can more properly formalise selectors as set-valued functions. In particular, a selector is now declared by a statement of the form

"selector" name "::" type ";"

where the type is required to be of the form $\tau \rightarrow \{1\}$ for some first-order monomorphic type $\tau$. For example, the following statement declares a selector that can be used to store details of books - consisting of their ISBNs, titles, authors' names and year of publication:

```
selector books ::
  (Num, Str, [Str], Num) \rightarrow
  | (Num, Str, [Str], Num)|;
```

A newly declared selector, $f$, may be assumed to be defined by an equation of the form

```
f p = (v \mid v \in \text{relation } \&\ p \neq v)
```

where $\text{relation} = \{\}$

Updates to a selector result in the relation expanding or contracting. In particular, the command

"include" selector value ";"

adds a new value to the relation, while the command

"exclude" selector pattern ";"

removes from the relation all the values that match the pattern. For example, the inclusions

```
include books (071678158, "Principles of Database and Knowledge-Base Systems", ["J.D.Ullman"], 1988);
include books (052126896, "Introduction to Combinators and Lambda Calculus", ["J.R.Hindley", "J.P.Seldin"], 1986);
include books (020508974, "Denotational Semantics", ["D.A.Schmidt"], 1986);
```

result in the following definition for the books selector:

```
books p = (v \mid v \in \text{relation } \&\ p \neq v)
```

where $\text{relation} =$

```
{(071678158,"Principles...",...),
(052126896,"Introduction...",...),
(020508974,"Denotational...",...)}
```

and subsequently the exclusion

```
exclude books (Any,Any,Any,1986)
```

removes the Hindley & Seldin and Schmidt books.

The application of a selector to an argument returns the set of the values in the relation that match the argument. For example, books Any returns the entire relation while books (Any, Any, Any, 1986) returns the details of books published in 1986.

4. Implementation

According to the definition of the fixpoint operator FIX in 3.1, functional languages can evaluate functions by commencing with least elements everywhere and continuing until no more information is inferred. For example, the factorial function, fact, of 3.2 is the least fixed point of the following higher-order function

```
H = \lambda f.\lambda x.\text{if } (x = 0) I (x * (f(x - 1)))
```

By the definition of FIX the successive approximations to $H$ are as follows:

- $H^0$ maps all numbers to $\bot_{\text{Num}}$.
- $H^1$ maps all numbers to $\bot_{\text{Num}}$ except 0 which is mapped to 1.
- $H^2$ maps all numbers to $\bot_{\text{Num}}$ except 0 and 1 which are both mapped to 1.
- $H^3$ maps all numbers to $\bot_{\text{Num}}$ except 0 and 1 which are both mapped to 1, and 2 which is mapped to 2, and so on, obtaining the factorial of a (positive) number $i$ on the $i+1$th approximation.

However, the expression $Y \in \text{hl.} (\lambda x.\text{hl}(x \times)) (\lambda x.\text{h}(x \times))$ has the property that for any expression $H$, $(Y H) = H (Y H)$ i.e. $(Y H)$ is a fixed point of $H$. In fact it can be shown [35] that $(Y H)$ is the least fixed point of any continuous function $H$ i.e. $(Y H)$ denotes $\text{FIX}(H)$. Thus, functional languages typically evaluate functions top-down using $Y$ rather than bottom-up using $\text{FIX}$. For example, to evaluate the factorial of 2 we start off with the expression $(Y H) 2$, and this is simplified through the following reductions:

```
(Y H) 2
→ H (Y H) 2
→ (\lambda x.\text{if } (x = 0) I (x * (Y H)(x - 1))) 2
→ 2 * ((Y H)(2 - 1))
→ 2 * ((Y H) 1)
→ 2 * (H (Y H) 1)
→ 2 * ((\lambda x.\text{if } (x = 0) I (x * (Y H)(x - 1)))) 1
→ 2 * (1 * ((Y H) (1 - 1)))
→ 2 * (1 * ((Y H) 0))
→ 2 * (1 * (H (Y H) 0))
→ 2 * (1 * ((\lambda x.\text{if } (x = 0) I (x * ((Y H)(x - 1)))))) 0
→ 2 * (1 * (1))
→ 2
```

Operationally then, $Y$ has the affect of producing a new copy of $(Y H)$ (the "meaning" of the factorial function) upon each recursive call. Notice that we are choosing to simplify the left-most, outermost redex at each step.
Evaluation in the presence of sets can proceed in a similar fashion. Sets are implemented as binary trees with the \( \cup \) operator at the inner nodes and singletons at the leaves. The operator + applies a function \( F \) to a set by distributing \( F \) to all the leaves. The application of \( F \) to singletons can occur asynchronously e.g. in parallel. However, the operator + violates lazy evaluation since it evaluates the argument to \( F \) before applying \( F \) (set the definition of \( F+ \)).

Optimisation to utilise semi-naive evaluation techniques is an ongoing area of work.

5. Expressiveness

Selectors are the analogue of the EDB predicates of a Datalog database, and from now on we term them extensional selectors. In 5.1 below we introduce the concept of an intentional selector and show how any Datalog IDB predicate - including those whose rules include negation and function symbols - can be translated into a PFL intentional selector. In 5.2 we discuss the optimisation of equations defining intentional selectors. In 5.3 we show how PFL with sets supersedes Datalog by supporting aggregation and counting operations. Lastly, in 5.4 we show how integrity constraints over selectors can be defined and efficiently enforced.

5.1 Intentional Selectors

We define an intentional selector to be a function \( f \) defined by an equation of the form

\[
\begin{align*}
f \mathbf{p} = s_1 \, U \ldots \, U \, s_n
\end{align*}
\]

where each \( s_i \) is either an application of a selector to the pattern \( p \), or a set abstraction \( \{ \ldots \} \) all of whose generators iterate over selectors. For example, the following function is an intentional selector:

\[
\begin{align*}
\text{anc} \, (x,z) &= \\
&\text{parent} \, (x,z) \cup \\
\{(x,z) \mid (x,y) \in \text{parent} \, (x,\text{Any}) \land \land \text{y} \in \text{y} \}
\end{align*}
\]

In our translation of Datalog rules into PFL, we first consider the case of IDB predicates defined by a single rule of the following form, where each \( q_i \) is either an EDB or IDB predicate, and each \( r_j \) is a built-in predicate:

\[
\forall (x,y) \in \text{ant} \, (x,2) \cup \text{ant} \, (y,2) \in \text{parent} \, (x,\text{Any}) \land \land \text{X} \in \text{X} \land \text{Y} \in \text{Y}
\]

Without loss of generality we assume that the variables of \( x \) are distinct. We also make the usual assumption of range-restriction \cite{range-restriction} for the \( r_j \), i.e. any variable appearing as an argument to an \( r_j \) is instantiated by some \( q_i \). We translate such a rule into a PFL equation of the form

\[
\begin{align*}
p \mathbf{x} = \{ \mathbf{x} \mid a_1 & \ldots \land a_n \land b_1 & \ldots \land b_m \land \\
c_1 & \ldots \land c_p \}
\end{align*}
\]

where:

(i) each literal \( q_i(y_1, y_2, \ldots) \) in the rule results in a generator \( a_i \) of the form \( \{ y_1, y_2, \ldots \} \in q_i(w_1, w_2, \ldots) \) such that:

- a constant or function symbol in \( y_{i,j} \) maps to the same constructor in \( w_{i,j} \);
- a universally quantified variable in \( y_{i,j} \) maps to the same variable in \( w_{i,j} \), and
- an existentially quantified variable in \( y_{i,j} \) maps to a new variable in \( w_{i,j} \) and the constant \( \text{Any} \) in \( w_{i,j} \).

(ii) each literal \( r_j(y_{n+1}) \) in the rule results in a semantically identical filter \( b_i \) in the equation; and

(iii) each pair of new variables \( z_{i,j} \) and \( z_{i,j'} \) arising from the same existentially quantified variable in (i) result in a filter \( c_{i,j} \) of the form \( z_{i,j} = z_{i,j'} \).

We note that the equation resulting from this translation satisfies the criteria for an intentional selector. For example, the following Datalog rule finds proper siblings:

\[
\begin{align*}
\text{siblings} \, (X,Y) &\leftarrow \text{parent} \, (Z,X), \text{parent} \, (Z,Y), X \neq Y
\end{align*}
\]

and the equivalent PFL equation obtained by our translation scheme is

\[
\begin{align*}
\text{siblings} \, (x,y) &= \\
\{ (x,y) \mid (z_1, x) \in \text{parent} \, (\text{Any}, x) \land \\
& (z_2, y) \in \text{parent} \, (\text{Any}, y) \land \\
& x \neq y \land z_1 \neq z_2 \}
\end{align*}
\]

We note that the repeated occurrences of the variables \( x \) and \( y \) in this equation are not ambiguous since, by the semantics of set 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 then overridden by subsequent occurrences in the heads of generators. This process of inheriting bindings from one qualifier to the next is, of course, the precise analogue of sideways information passing \cite{sideways} in the evaluation of logic rules.

The above translation scheme is clearly inefficient: for example, rules containing existentially quantified variables translate into equations which take cartesian products of selectors and then perform selections. More sophisticated translation schemes which overcome these problems are possible (e.g. the one given in \cite{optimisation} for the
previous version of PFL). However, we have used this particular scheme here because it simplifies our discussion of selector optimisation in 5.2 below.

We can extend the translation scheme to IDB predicates that are defined by more than one rule. Without loss of generality we assume that these rules are rectified [36] i.e. that they all have the same head. Each rule is first separately translated according to the above scheme, giving a number of equations with the same left hand side. These equations are then combined into a single equation by forming a union from their right hand sides. For example, the standard rules for the ancestor predicate

\[
\text{anc}(X, Z) \leftarrow \text{parent}(X, Z) \\
\text{anc}(X, Z) \leftarrow \text{anc}(X, Y), \text{anc}(Y, Z)
\]

translate into the PFL equation

\[
\text{anc}(x, z) = \\
\text{parent}(x, z) \cup \\
\{ (x, z) \mid (x, y_1) \in \text{anc}(x, \text{Any}) \land \\
y_2, z) \in \text{anc}(\text{Any}, z) \land \\
y_1 = y_2 \}
\]

Similarly, the rules

\[
\text{nat}(X) \leftarrow \text{is-number}(X) \\
\text{nat}(X) \leftarrow \text{nat}(Y), X \leftarrow \text{succ}(Y)
\]

translate into the equation

\[
\text{nat}_x = \\
\{ x \mid x \in \text{is-number} x \} \cup \\
\{ x \mid y \in \text{nat}, \text{Any} \land x \in \{ \text{succ} y \} \}
\]

assuming that \text{is-number} is a selector of type \text{Nat} !\rightarrow \text{Nat} and \text{succ} a constructor of type \text{Nat} !\rightarrow \text{Nat}.

We now extend our translation scheme to cover the case of negative literals. We first make the usual assumption that every variable appearing in a negative literal also appears in some positive literal which precedes it in the rule. We then translate a negative literal, \(-p(Z)\), into a filter (p Z) = \{\}. For example, the following Datalog rule assumes that people are male if they are not female

\[
\text{male}(X) \leftarrow \text{person}(X) \land \neg \text{female}(X)
\]

and the equivalent PFL equation is

\[
\text{male}_x = \\
\{ x \mid x \in \text{person} x \land (\text{female} x) = \{\} \}
\]

The translation of Datalog rules which are not stratified with respect to recursion through negation results in functions which yield only the least element of the set domain. For example, this occurs if we define males as persons who are not female and vice versa:

\[
\text{male}_x = \\
\{ x \mid x \in \text{person} x \land (\text{female} x) = \{\} \}
\]

\[
\text{female}_x = \\
\{ x \mid x \in \text{person} x \land (\text{male} x) = \{\} \}
\]

A similar source of no information being inferable is recursion through the set-to-list function of 3.3 (recall that set-to-list requires its argument to be fully evaluated before it can be applied).

5.2 Transformation of Intentional Selectors

It is possible to apply several syntactic transformations to the set abstracts defining intentional selectors. In particular, we use the following transformations:

Tr1: Reordering the qualifiers so that each filter appears as early as possible and each generator as late as possible, subject to the constraint that each qualifier must follow all of the generators which provide instantiations for its free variables (this is equivalent to undertaking selections as early as possible in relational queries).

Tr2: Switching pairs of generators over the same selector, \(p_a \in a\) and \(p_b \in b\), whenever \(a \neq b\) (subject to the same constraint as for Tr1).

Tr3: Replacing inequality by pattern-matching (this is equivalent to pushing selections through joins) i.e. given a sequence of qualifiers

\[
\ldots \land q_{n-1} \land (\ldots x_{i-1}, x_i, x_{i+1}, \ldots) \in s(\ldots y_{j-1}, \text{Any}, y_{j+1}, \ldots) \land q_{n+1} \land \ldots \land q_{n+m-1} \land E = x_i \land q_{n+m+1} \land \ldots
\]

where \(E\) is an expression all of whose variables are bound by qualifiers up to \(q_{n-1}\), we can remove the qualifier \(E = x_i\) and replace \(\text{Any}\) by \(E\) in \(q_n\):

\[
\ldots \land q_{n-1} \land (\ldots x_{i-1}, x_i, x_{i+1}, \ldots) \in s(\ldots y_{j-1}, E, y_{j+1}, \ldots) \land q_{n+1} \land \ldots \land q_{n+m-1} \land q_{n+m+1} \land \ldots
\]

For example, applying the ant function of 5.1 above to an argument (Any,B), where \(B\) is a constant other than Any, results in the equation:

\[
\text{ant}(\text{Any}, B) = \\
\text{parent}(\text{Any}, B) \cup \\
\{ (x, z) \mid (x, y_1) \in \text{ant}(\text{Any}, \text{Any}) \land \\
y_2, z) \in \text{ant}(\text{Any}, B) \land \\
y_1 = y_2 \}
\]

By applying Tr2 we obtain

\[
\text{ant}(\text{Any}, B) = \\
\text{parent}(\text{Any}, B) \cup \\
\{ (x, z) \mid (y_2, z) \in \text{ant}(\text{Any}, B) \land \\
y_2, z) \in \text{ant}(\text{Any}, \text{Any}) \land \\
y_1 = y_2 \}
\]

and by then applying Tr3 we obtain

\[
\text{ant}(\text{Any}, B) = \\
\text{parent}(\text{Any}, B) \cup \\
\{ (x, z) \mid (y_2, z) \in \text{ant}(\text{Any}, B) \land \\
y_2, z) \in \text{ant}(\text{Any}, \text{Any}) \land \\
y_1 = y_2 \}
\]
The process we have followed resembles the magic sets [6] method of rewriting Datalog rules, which generates a different set of rules for a predicate depending upon the binding pattern. We are currently investigating adapting the full magic sets method for the bottom-up evaluation of intentional selectors. A further optimisation would be to store evaluated intentional selectors in the database until an update to a dependent function (see 5.4 below) invalidates them.

5.3 Beyond Datalog

As we would expect, PFL is more expressive than Datalog since set-manipulation functions such as nesting, counting, and unnesting can be defined. We give three examples below: "nest" nests the anc relation, yielding a set of pairs \((x,\text{ancs})\) such that \(\text{ancs}\) is the set of all the ancestors of person \(x\); "generation" returns a set of pairs \((x,g)\) such that \(g\) is the generation of person \(x\) (which is calculated by counting the number of female ancestors of \(x\)); and "unnest" yields a set of pairs \((x,a)\) such that \(a\) is an ancestor of \(x\):

\[
\text{nest} = \{ (x, \{a | (a, x) \in \text{anc}(\text{Any}, x) \}) | (y, x) \in \text{parent}(\text{Any}, \text{Any}) \}
\]

\[
\text{generation} = \{ (x, \text{length}\{y | y \in z \& \text{female } y \neq 0 \}) | (x, z) \in \text{nest} \}
\]

\[
\text{unnest} = \{ (x, \{a | (x, \text{ancs}) \in \text{nest} \& a \in \text{ancs} \}) \}
\]

The second of these functions uses an infix operator \(\&\) which allows a function expecting a list as an argument to be applied to a set; \(\&\) is defined as

\[f \& s = f(\text{set-to-list } s)\]

5.4 Integrity Constraints over Selectors

We define an integrity constraint to be an intentional selector, \(f\), such that the database state is incorrect whenever the expression \((f \text{ Any})\) denotes anything other than the empty set. For example, we may define a constraint ic1 which states that no-one is both male and female and a constraint ic2 which states that no-one is their own ancestor:

\[
ic1 x = \{ y_1 \mid y_1 \in \text{male } x \& y_2 \in \text{female } x \& y_1 = y_2 \}
\]

\[
ic2 x = \{ x \mid (x, y) \in \text{anc}(x, \text{Any}) \& x = y \}
\]

Since the filters of set abstractions in an intentional selector can contain arbitrary functions, we can also enforce cardinality constraints. For example, ic3 ensures that anyone who is recorded as having parents (the first generator) has precisely two parents (the second generator):

\[
ic3 x = \{ x \mid (y, x) \in \text{parent}(\text{Any}, x) \& \text{length}\{ \text{parent}(\text{Any}, x) \} = 2 \}
\]

It is clearly desirable to use the updates to the database to restrict the amount of computation required to validate constraints. A number of constraint optimisation techniques have been developed for logic languages (see [15] for a review of them) any of which could be adapted for our integrity constraints. Here we show how one such method - that of Lloyd and Topor [21] - can be used. Our account roughly follows that of [30] for the handling of integrity constraints in the previous version of PFL, modified to utilise set-valued rather than list-valued constraints.

We say that a function \(f_1\) directly depends upon a function \(f_2\) if \(f_2\) occurs in an equation that defines \(f_1\); we use the term depends upon for the transitive closure of the directly depends upon relationship. This "call graph" is already automatically maintained by PFL as equations are inserted and deleted since we need it to infer the types of functions. We can now put it to further use for the enforcement of integrity constraints.

We recall from 5.1 that an intentional selector (and hence an integrity constraint) is a function of the form:

\[f \circ \{ \text{p} \mid (c_1 \& \ldots \& c_m) \} \circ (\text{c} \& \ldots \& \text{c}_m)\]

where, without loss of generality, we have converted summands of the form \(\text{p p} = \{\}\) to the syntax of 5.1. We distinguish between three different forms of qualifier occurring in such an intentional selector:

Type 1: \(c_{ij}\) is of the form \(p_1 \in s p_2\),

Type 2: \(c_{ij}\) is of the form \(s p = \{\}\), and

Type 3: any other form of qualifier,

where \(s\) is a selector and \(p, p_1\) and \(p_2\) are patterns. Note that the three types of qualifier correspond to positive, negative and built-in predicates of Datalog rules respectively. Inclusion or exclusion of tuples in extensional selectors trigger implicit inclusions and exclusions from intentional selectors. In particular, the (explicit or implicit) inclusion of a value \(t_1 = (u_1, \ldots, u_r)\) into one selector, \(s_1\), may affect another selector, \(s_2\), in one of three ways:

(i) Inclusion of values into \(s_2\): If \(s_2\) directly depends upon \(s_1\) via a Type 1 qualifier i.e.

\[s_2 p = \ldots \cup \{ c \mid \ldots \& (v_1, \ldots, v_t) \in s_1 (w_1, \ldots, w_r) \& \ldots \} \cup \ldots \]

let \(t\) be the value obtained from \(c\) by replacing \(v_j\) by \(u_j\), for all \(1 \leq j \leq r\), and any other free variables of \(c\) by \(\text{Any}\). Then for any value \(t_2\) implicitly included into \(s_2\) as a result of the insertion of \(t_1\) into \(s_1\), we have \(t = t_2\) holds.

(ii) Exclusion of values from \(s_2\): If \(s_2\) directly depends upon \(s_1\) via a Type 2 qualifier i.e.
s2 = \ldots \cup \{ e \mid \ldots \} \cup s1 \{ v_1, \ldots, v_r \} = \{ \} \cup \ldots \cup \ldots

Again, let \( t \) be the value obtained from \( c \) by replacing

\( v_j \) by \( u_j \) for all \( 1 \leq j \leq r \) and any other free variables of \( c \) by Any. Then for any tuple \( t' \) implicitly excluded from \( s2 \) as a result of the insertion of \( t1 \) into \( s1 \), we have \( t' \neq t2 \).

(iii) Both inclusion into and exclusion from \( s2 \) : If \( s2 \) depends upon \( s1 \) via a Type 3 qualifier then arbitrary variables may explicitly have been included and excluded. Let \( t \) be the value obtained from \( c \) by replacing any free variables by Any. Then for any tuple \( t2 \) implicitly included or excluded from \( s2 \), we have \( t' \neq t2 \).

The exclusion of a value from one selector may similarly affect another; in this case the roles of Type 1 and Type 2 qualifiers are interchanged. Thus, in our description of the constraint enforcement algorithm below, we assume the availability of two functions: "includes(s,v,u)" takes a selector \( s \), a value \( v \) and a flag \( u \) indicating whether \( v \) represents a possible inclusion or a possible exclusion from \( s \), and returns a set of pairs \((s',v')\) such that \( s' \) is a selector either directly depending upon \( s \) via a Type 1 or Type 2 qualifier or indirectly via a Type 3 qualifier, and the \( v' \) match all the inclusions into the \( s' \); similarly, "excludes(s,v,u)" returns a set of pairs \((s',v')\) such that the \( v' \) match all exclusions from the \( s' \).

To optimise the enforcement of constraints, the system automatically associates with each selector, \( s \), affected by an update two sets, \( s_i \) and \( s_e \), which respectively contain tuples matching all tuples included into (or excluded from) \( s \). These sets are built recursively using the function "update" given below. In particular, after the inclusion (or exclusion) of a value into an extensional selector \( s \), \( update(s,v,"i") \) (or \( update(s,v,"e") \)) is called to maintain these sets:

```pseudocode
update(s,v,"i_or_e")
{
  if \( v \in s_i \) or \( e \)
    return;
  \( s_i \) or \( e \) = \( s_i \) or \( e \) \cup \{ v \};
  for ((s',v') \in includes(s,v,"i_or_e"))
    update(s',v',"i");
  for ((s',v') \in excludes(s,v,"i_or_e"))
    update(s',v',"e");
}
```

The database is validated with respect to the constraints when a commit point is reached. Validity is ascertained by evaluating the expression \( ic(t) \) for each constraint \( ic \) and each value \( t \) in \( ic \). In general, a set \( ic \) may contain redundant tuples: whenever there are tuples \( t1, t2 \in ic \), such that \( t1 \neq t2 \), then \( t2 \) may safely be removed from the set since \( ic(t2) \subseteq ic(t1) \). Thus constraints need only be validated with respect to these reduced sets of tuples.

6. Conclusions

We have described how the functional database language PFL can be extended with sets as first class objects. Our support of the value Any and the \( \neq \) operation then allows any Datalog\textsuperscript{fun+neg} predicate to be expressed as a PFL function. We thus combine the respective advantages of functional and logic database languages within one semantic and operational framework. Our work can also be considered as contributing to the formalisation of database concepts using powerdomain theory, as exemplified by Buneman et al [8].

More specifically, in common with functional database languages such as [3, 4, 5, 7, 16, 18, 23, 28] we support deterministic computations over large volumes of data. We also support the storage of all types and functions in the database, a feature found only in [28]. In common with logic database languages [12, 13, 25] we also support search-based computations over large volumes of data. Several functional languages provide relational processing by incorporating records [2, 9, 23, 24, 26] but it is not clear how full deductive capabilities can be achieved in these languages. FAD [4] does add sets to a functional computation model, but relies upon the ability to "call out" to an external, computationally complete, language in order to define arbitrary functions. Several logic-based languages also incorporate sets [20]. In particular, COL [1, 17] integrates both functions and sets into a logic framework and thus has similar expressiveness to our language. However, it too achieves this by assuming the ability to call out to an external language to define arbitrary functions. In contrast we use one language and one database to store all information.

We have indicated how optimisation techniques developed for both functional and logic languages can be transferred to PFL, for example for recursive query processing and for integrity constraint enforcement. These are areas of ongoing research. We are also investigating suitable bulk data structures to efficiently support the \( \neq \) operation. Finally, PFL is currently being used to analyse road traffic accident data, which requires both search-based computation, e.g. to find the nearest site (junction, roundabout etc.) to a given accident location, and deterministic computation, e.g. to group accidents by site and to produce accident statistics.

Acknowledgements

We are grateful to Swarup Reddi for discussions on incorporating integrity constraints into PFL. The work described in this paper has been supported by the U.K. Science and Engineering Research Council (grant no. GR/G 19596).

References


