PIL : An Optimizable Functional Language for Data Intensive Applications

Waqar Hasan Ravi Krishnamurthy
Hewlett-Packard Laboratories
1501, Page Mill Road, Palo Alto, CA 94305, U.S.A
email: lastname@hpl.hp.com

Abstract

The Papyrus Interface Language (PIL) has the design goal of providing *optimizable and parallelizable* language features. Analogous to the design philosophy of RISC instruction sets, the design of PIL is motivated by the desire to exploit query optimization and parallelization techniques. In contrast, most proposals of database programming language provide features to directly match user needs irrespective of the implementation problems, analogous to the CISC instruction set proposals. We have combined a functional model of computation with data types suitable for data intensive applications. A functional model gives a *declarative semantics* to all expressions including "procedural" constructs such as if-then-else, while and function calls provided the expression is without side-effects. We also provide specialized constructs for iteration over bags and sequences in order to facilitate optimization. The semantics of data types and computational abstractions are carefully chosen to retain the capability to parallelize programs. We have chosen to only *partially define the order of evaluation* for programs. This opens up more opportunities for reordering and for parallel execution. Just as in RISC instruction sets — what is not included will be a factor in dictating the performance of the system — we argue the need to exclude features such as object identity, semantic types and inheritance that are popularly included in most database programming languages.

1 Introduction

Database programming languages (DBPLs) provide increased functionality by the provision of features such as if-then-else, while loops, object-identity, type inheritance and overloading. These designs have been traditionally motivated by the goal of directly reflecting the needs of an end-user as features in the language. This is much like the design of instruction sets for CISC [1] architectures which had the similar intention of removing impedance mismatch. In contrast the design of instruction sets for RISC[2] architectures is primarily motivated by what can be efficiently implemented on the underlying hardware.

The design goal of PIL (Papyrus Interface Language) is to provide a set of *optimizable and parallelizable* language features. Analogous to the RISC design

[1] Complex Instruction Set Computers
[2] Reduced Instruction Set Computers

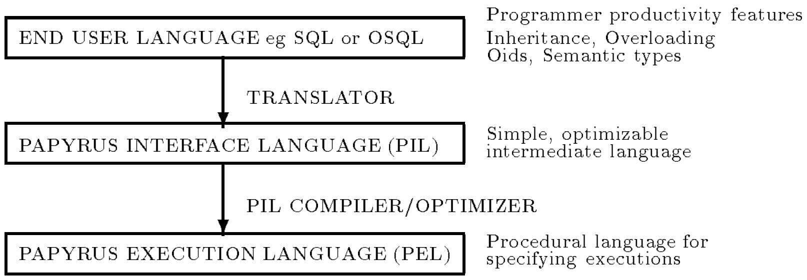

Figure 1: Language Levels in Papyrus

philosophy, the design of PIL is motivated by the desire to exploit query optimization techniques, and parallelization capabilities such as cloning — wherein clones of code are executed in parallel on disjoint partitions of data.

PIL is designed to be an *intermediate* language. Unlike most DBPLs such as OSQL[FAC+89, WLH90] (the language supported by Iris/OpenOdb), FAD[BBKV87], Napier[MBCD89], Galileo[ACO85] and Algres[CCRLZ89] PIL is *not* object-oriented. However, as shown in Figure 1, it is intended that several end-user database languages such as OSQL and SQL be supported on top of PIL. This reflects the belief that while object-oriented features such as object-identity, semantic types, inheritance and overloading enhance programmer productivity, the picture for optimization should be simplified by "pre-processing" away such features.

In keeping with the RISC design philosophy, we believe that gains from excluding features that hinder optimization will offset losses stemming from losing some of the semantics (and therefore some possible optimizations) of the features that are pre-processed away.

PIL combines a functional model of computation[BW88, AS85] with data types suitable for data intensive applications. A functional model gives a *declarative semantics* to all expressions including "procedural" constructs such as if-then-else, while and function calls provided the expression is without side-effects. A pure functional model can be extended to incorporate assignment without destroying the declarative reading of side-effect free (sub)expressions. We have adopted the environment model of expression evaluation as defined in the Scheme language[A+91].

We have chosen to only *partially define the order of evaluation* for programs. This opens up more opportunities for reordering and for parallel execution. For example we provide two constructs for iteration over bags, one with a parallel semantics (simple iteration) and the other with a serial-commutative semantics (reduction). Such constructs provide more degrees of freedom to the compiler in optimizing and parallelizing programs. Such constructs also make it possible to express that the order of eveluation is unimportant. This is important since compilers cannot always make such inferences. These and other such judicious choices for language semantics are elaborated in this paper.

This work is part of the Papyrus project[CHK+91] at HP Labs which is investigating ways of integrating highly-tuned, customized data managers while preserving their requirements for performance. Sources of high performance include optimization and parallel execution.

The next section gives a brief overview of PIL, Section 3 discusses the model of data and computation. Section 4 discusses the abstractions for iteration and opportunities for their parallelization.

2 PIL in a Nutshell

PIL provides several abstractions for iteration at different levels of generality. The *while* abstraction provides a very general form of iteration with a serial semantics. Reduction is a special case of while in which the domain of iteration is a bag and the body of the loop is executed once per bag element. Variables may be shared across iterations. Reduction is defined to have a serial-commutative semantics. In other words the execution must be equivalent to some serial ordering of the iterations. Simple iteration is a special case of reduction in which each iteration is independent (no variables are shared). Simple iteration has a parallel semantics.

Simple iteration is a generalization of Select-Project-Join queries while reduction is a generalization of queries with aggregation. The pre-existing body of knowledge on relational query optimization and execution may be applied to these abstractions.

Arguments to functions are passed by value. The only restriction on the order of evaluation of arguments is that it must be equivalent to some serial order of evaluation of arguments. This freedom allows a compiler to choose the best amongst alternate execution strategies.

Types are associated with values or in other words data is self describing. Types are also be stated as constraints on variables. Since PIL is an intermediate language, there is no type checking. However types serve the role of providing useful information to the compiler. The PIL compiler uses type inference to facilitate the generation of efficient code.

PIL provides the *type constructors* bag, tuple, labeled tuple and sequence for modeling aggregate data common in database applications. Direct support for sequences not only makes certain queries such as the k-day moving average of HP stock easier to express but also facilitates their efficient implementation. A disadvantage of not providing sequences in SQL is that queries with *order by* clauses do not produce relations and therefore have to be disallowed in SQL views.

PIL permits aggregate objects to be arbitrarily nested. A nested model is not as simple as a flat model. However, it provides modeling and processing advantages for domains where nested objects occur naturally.

PIL programs are block-structured like those in Algol or Pascal. Each use of a variable is associated with a lexically apparent binding of that variable.

A *database schema* is a named set of declarations of types and variables. Note that in our model functions are thought of as variables. An *environment* is a mapping of variables to values.

A *database* is a named persistent environment. A database is an instance of a database schema if it has the same name and provides bindings for exactly the variables and functions in the schema. A variable is persistent if it is defined in a database schema. A value becomes persistent if it is assigned to a persistent variable.

type *Cell*; /* A foreign type */
type *RoadSeg* $= \langle\langle name : String, segment : Cell \rangle\rangle$;
var *Roads*: $\{RoadSeg\}$;
function *shape*$(Cell) \longrightarrow [\langle\langle latitude : Float, longitude : Float \rangle\rangle]$);
function *length*$([\langle\langle latitude : Float, longitude : Float \rangle\rangle]) \longrightarrow$ *Float*;
function *plus(Float, Float)*$\longrightarrow$*Float*;

Figure 2: Logical Schema for the SimpleGIS database

Roads = { ⟨⟨ "El Camino", cell1 ⟩⟩
⟨⟨ "El Camino", cell2 ⟩⟩
⟨⟨ "Page Mill Road", cell3 ⟩⟩};
shape = `a function object`;
length = `a function object`;
plus = `a function object`;

Figure 3: The SimpleGIS database

2.1 Examples

Figure 2 shows the schema for a database of geographic information. The logical schema defines the names and types of the functions and variables in the database. Figure 3 shows the contents of the GIS database.

A road has a name and consists of a number of *segments*. A new segment starts each time the road intersects with a cross street or crosses a political boundary such as a county boundary. A segment is a continuous, but not necessarily straight, line. Segments are stored as geometrical objects of type *cell*. *Roads* is a bag of tuples with one tuple per segment of a road. The function *shape(cell)* returns the shape of the segment as a sequence of two or more points with each point represented as a tuple consisting of the *latitude* and *longitude* of the point. The *length* function computes the length of a sequence of points and *plus* adds two floating point numbers.

Example 1 The following query returns the shape of all segments that make up the road named "El Camino". The result of the query is a bag of sequences and may be used to graphically display the shape of El Camino on a map. The *simple iteration* construct used below is a generalization of select-project-join queries. One way of processing the query is to first select all road segments named "El Camino", then pipe the selected road segments to *m clones* of the shape function. The decision of which clone a particular road segments goes to may be made based on a hash function. The stream of results produced by each clone may then be merged to form the result.
for r:RoadSeg **in** Roads
where r.name="El Camino"
apply shape(r.segment) ∎

Example 2 The following query computes the length of El Camino. It uses a *reduction* construct which has a serial-commutative semantics. First the variable *len* is initialized to zero and then we iterate over all tuples in *Roads* which satisfy *r.name = "El Camino"* in an *arbitrarily* chosen order. For each such tuple the body of the loop, which accumulates the length of the road in the variable *len*, is executed.

with len:Float := 0
for r: RoadSeg **in** Roads
where r.name="El Camino"
reduceby len := plus(len, length(shape(r.segment)))

One way of executing the query is to partition the road segments amongst several clones. Each clone can compute the sum of the lengths of its segments and then the lengths produced by each clone may be summed up.

■

3 Model of Data and Computation

Our model consists of a functional model of computation and some chosen set of *built-in* data types and *built-in* functions defined on the types.

Persistence is provided through the notion of a database as a named persistent environment.

The choice of data types is orthogonal to the model of computation and may be easily extended to incorporate *foreign* types and functions in addition to the those that are built-in. A discussion of how foreign types and functions are implemented is beyond the scope of this paper.

Section 3.1 describes the built-in data types and Section 3.2 discusses the primitives out of which expressions are constructed and the model of expression evaluation. Lastly, in Section 3.3 we describe how type declarations (viewed as constraints on values for variables) may be added to the notion of expressions.

3.1 Data Types

All data objects in PIL are literal objects[FAC+89]. A literal object is a *value*. For example integers, bags and functions are literals but pointers and object identifiers are not.

Both built-in and foreign data types can be categorized into atomic and aggregate types. Data types such as *Integer*, *String*, and *Float* are *atomic*. Atomic objects have no structure visible in the language. *Aggregate* objects, on the other hand, are made up of components. This structure is made visible in the language by the provision of functions to pick out components, iterate over components and construct aggregate objects from components.

PIL provides *type constructors* for four kinds of aggregates: *bag* to model unordered collections with duplicates, *sequence* to model ordered collections and *tuple* and *labeled tuple* to model records. An additional type constructor, *function*, is provided to model functions.

Figure 4 shows the types provided in PIL and how they are related by a subtype relationship. Types constructed using the type constructors are not shown since there are an infinite number of such types. Types constructed

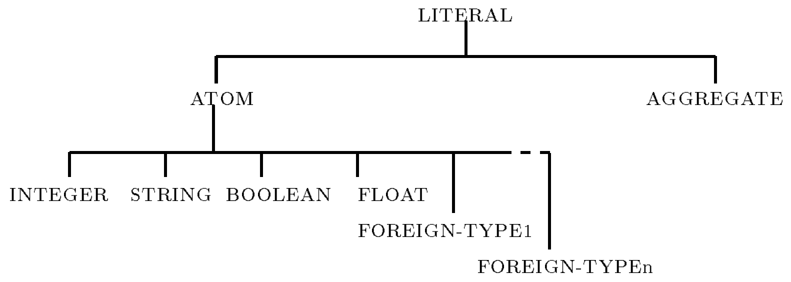

Figure 4: The PIL type graph

using the aggregate type constructors are subtypes of *Aggregate* while types constructed using the function type constructor are subtypes of *Atom*.

Kent[Ken] provides a general approach for characterizing a large variety of aggregate objects by giving dimensions along which they may be characterized. We have chosen specific kinds of aggregate objects which we describe along some of the suggested dimensions. Table 1 summarizes some of the characteristics and [Has92] provides precise definitions of the functions described informally below.

Each aggregate type has an associated *constructor* function. The symbol $\{\}$ is used to denote an empty bag, $\langle\langle\rangle\rangle$ for an empty tuple and [] for an empty sequence. Given components $e_1, \ldots, e_n$ $[e_1, \ldots, e_n]$ constructs a sequence, $\langle\langle e_1, \ldots, e_n\rangle\rangle$ constructs a tuple, $\langle\langle l_1 : e_1, \ldots, l_n : e_n\rangle\rangle$ constructs a labeled tuple with labels $l_1, \ldots, l_n$, and $\{e_1, \ldots, e_n\}$ constructs a bag[3]. Null components are permitted for tuples and sequences but not for bags.

Functions are provided to access components of an aggregate object. Tuples and sequences are ordered collections. The function *tuple_pick(t, i)*, where i is a positive integer returns the ith component of tuple t. Similarly *seq_pick(s, i)* (which is alternatively written as $s[i]$) returns the ith component of sequence s. Bags are unordered collections. Therefore, for a bag, b, *bag_pick(b)* returns an arbitrarily chosen component or $Null$ if the bag is empty.

The *equality* of aggregate objects is based on the equality of their components. Two sequences are equal if corresponding components are equal. Tuple equality is similar to that for sequences. Two bags are equal if the multiplicity of each member element is the same in each bag.

Bags and sequences are *homogeneous* in the sense that all components are of the same type. Tuples and labeled tuples are *heterogeneous* and a different type may be specified for each component.

3.1.1 Discussion

Inclusion of sequences: PIL provides sequences to model ordered collections of data. Having sequences permits the direct modeling of ordered domains such as time. For example the complexity of writing a query to compute the k-day moving average of stock prices changes considerably depending on whether

[3] Constructors are no different from ordinary functions. For example $\{e_1, \ldots, e_n\}$ is simply a different notation for $bag\_construct(e_1, \ldots, e_n)$.

	Access to Components	*Iter. over Compnts.*	*Null Compnts.*	*Form of type expression*
Bag	$bag\_pick(b)$	Yes	No	$\{T\}$
Sequence	$s[i]$	Yes	Yes	$[T]$
Tuple	$tuple\_pick(t,i)$	No	Yes	$\langle\langle T_1,\ldots,T_n\rangle\rangle$
Labeled Tuple	$tuple\_pick(t,i)$ $t.l_i$	No	Yes	$\langle\langle l_1 : T_1,\ldots,l_n T_n\rangle\rangle$

Table 1: Characteristics of Aggregate Types

the daily stock quotes are represented as a sequence or as a bag. Sequences naturally provide the notion of the "next" element. If sequences are simulated using bags, the "next" element has to be computed as the minimum of all elements which are greater than the given element. We believe direct support for sequences is essential for their efficient implementation.

Another advantage of providing sequences is to remove the restriction of disallowing SORT to occur in view definitions in languages such as SQL.

Nested Model: Forcing a flat representation for objects that are naturally nested has some disadvantages. Flattening leads to the introduction of of artificial identifiers. It also leads to a loss of certain uniqueness constraints that are implemented by the very structure of a nested representation. Another advantage that is lost is the granularity of manipulation.

The disadvantages of the nested model stems primarily from sacrificing the simplicity of the flat model and the poor state of the art in effective query processing and optimization techniques (how do you estimate selectivities or implement pipelining for example).

Exclusion of Pointer: The *pointer* type constructor has been deliberately excluded from PIL. The presence of pointers increases the complexity of optimization. For example, in the absence of pointers it is easy to see that any two variable are disjoint and therefore may be operated upon in parallel. In the presence of pointers, any inference procedure for disjointness is likely to be highly pessimistic in order to ensure correctness. The intuitive reason is that the procedure would only have type information available to it. Therefore if the type of any component of one variable is "$pointer(t)$" where type t is the type of some component of the other variable, then we would not be able to infer the two variables to be disjoint.

Exclusion of Inheritance and Overloading: We believe that inheritance and overloading are end-user language features that increase programmer productivity but can easily be pre-processed in the translation to PIL. An overloaded function can either be resolved at translation time or an if-statement can be generated if the resolution is to be postponed to run-time.

Another reason for the exclusion of such features is that PIL is an intermediate language intended to support *multiple* end-user languages. Since end-user languages differ in their model of such features, we felt it would not make much sense to build yet another model of these features into PIL.

3.2 Expressions and Evaluation

We summarize below the syntax and semantics of the *primitive* expressions in PIL. More useful derived expressions such as abstractions for iteration may be constructed from the primitive expressions.

1. A constant c is an expression which evaluates to itself.
2. A variable v is an expression. A variable evaluates to its binding.
3. A conditional expression has the form **if** p **then** e_1 **else** e_2. First p is evaluated and after that if p was $True$, e_1 is evaluated, otherwise e_2 is evaluated. The else part is optional.
4. A lambda expression **lambda**$(v_1, \ldots, v_n)e$ consists of a list of formal parameters (variables) and a body. It evaluates to a function object.
5. A function call $f(e_1, \ldots, e_n)$ is an expression. It is evaluated by first evaluating the parameters in an arbitrarily chosen order. Then the body of the function is evaluated.
6. An assignment $v := e$ is an expression. The expression e is evaluated and v is bound to the resulting value. The result value of an assignment expression is undefined.

We also introduce the following *derived* constructs:

1. **begin** $expr_1; \ldots; expr_n$ **end** explicitly sequences the evaluation of the expressions and returns the value of $expr_n$ as the result[4].
2. **function** $f(v_1, \ldots, v_n)$ **as** e defines f as a function[5].

Example 3 The following expression evaluates to *False* in the environment of the SimpleGIS database shown in Figure 3.

begin
  function empty(b) **as** is_null(bag_pick(b));
  empty(Roads)
end

■

3.3 Typed Expressions

The primary intention of a type system in PIL is to increase the information available to the PIL compiler. More specific type information permits the generation of better code in the translation of PIL to PEL[CS92, SC92].

The PIL compiler uses type inferencing to derive information on the types of expressions. Since PIL is an intermediate language, we decided that the compiler should desist from type checking. It is assumed that the type-correctness of programs is guaranteed by translator from the end-user language.

[4]For example **begin** $expr_1; expr_2$ **end** is equivalent to $(lambda(x)expr_2)expr_1$. $(lambda(x)expr_2)$ evaluates to a function which takes one parameter. $expr_1$ is the parameter which is evaluated *before* the body of the function, consisting of $expr_2$ is evaluated.

[5]This construct is equivalent to $f :=$ **lambda** $(v_1, \ldots, v_n)$ e.

In PIL programs, a type must be declared for each variable. A type declaration is simply a *constraint* on the set of values considered legal for a variable. In addition to type constraints, it is also possible to specify a variable to be a *const*ant. In other words the variable is assigned to exactly once.

Example 4 The expression of Example 3 must now be written as:

> **begin**
>   **function** empty(b:{Literal}) $\rightarrow$ Boolean
>   **as** is_null(bag_pick(b));
>   empty(Roads)
> **end**

■

We require that types be declared for the parameters and the result of a lambda expression. Consider T or T_i to be type expressions. The syntax for a lambda expression becomes as follows:
T **lambda**$(v_1 : T_1, \ldots, v_n : T_n)e$

The derived construct for function definition is also extended to be of the following form[6]
function $f(v_1 : T_1, \ldots, v_n : T_n) \rightarrow T$ **as** e

4 Abstractions for Iteration

We provide several abstractions for iteration over bags and sequences. The intention of providing these built-in constructs is to capture the operations *common* in database applications in such a way as to facilitate their optimization and parallelization. The choice of abstractions takes into consideration the capabilities that can be reasonably expected from a compiler.

A *while* loop construct is provided as a general means of iteration. We also provide *simple iteration*, *reduction* and *grouping* as special cases. These special cases represent forms of iteration common in database applications and for which we can expect to develop optimization and parallelization techniques.

Simple iteration is a generalization of select-project-join queries. It provides opportunities for cloned parallel and the design facilitates the reuse of relational optimization techniques. Reduction provides the ability to compute aggregate properties such as sum or standard deviation. The semantics of reduction is defined so as to facilitate cloned parallel execution. The *groupBy* function divides a bag into equivalence classes and thus provides a natural granularity for the cloning of subsequent operations.

Clearly our philosophy is not to have the minimal set of abstractions but to provide optimizable constructs in a declarative language framework. For example, simple iteration is a special case of reduction and reduction is a special case of the while loop.

[6] This construct is equivalent to *Const* $f : (T_1, \ldots, T_n) \rightarrow T :=$ **lambda** $(v_1 : T_1, \ldots, v_n : T_n)\ e$

4.1 Simple Iteration Over Bags

Simple iteration is modeled as the higher order function

$$filter(f, p, b_1, \ldots, b_n)$$

which is a generalization of select-project-join queries. It takes n bags $b_1, \ldots, b_n$ and two n-ary functions f and g as parameters. The arbitrary predicate, p, may be used for selection and the arbitrary function, f, for projection. It is equivalent to

$$\Pi_f(\sigma_p((b_1 \times \ldots \times b_n)))$$

In other words, the $filter$ function takes each element x in the cartesian product of the bags, tests whether $p(x)$ is true, and if so computes $f(x)$. The result is the bag of all such $f(x)$s.

Simple iteration offers the opportunity for parallel execution through cloning. Cloning is essentially based on the fact that it is possible to view a bag as the union (merge) of m partitions (split) based on some hash function.

$$b = Merge(Split(b, 1), \ldots, Split(b, m))$$

Operations such as Π_f and σ_p commute with Split and Merge. Join methods also offer opportunities for commuting since we can use the following rewrite rule.

$$\begin{aligned} Join(b_1, b_2) = Merge(&Join(Split(b_1, 1), Split(b_2, 1)), \ldots, \\ &Join(Split(b_1, m), Split(b_2, m)))) \end{aligned}$$

These rewrite rules open up a large space of parallel executions from which an optimizer can be expected to choose the cheapest plan. Several parallel join algorithms based on such rewrites are discussed in [SD89].

The FAD language provides a similar filter function but combines the selection (p) and projection (f) into a single function (g). The abstraction $filter_{FAD}(g, b_1, \ldots, b_n)$ creates two complications. Firstly the semantic complication is that g must be allowed to return $Null$ and set insertion must be defined so that inserting a $Null$ into a set returns the original set. Second, and more importantly, optimization becomes harder since selections, which are important to predict the cardinality of results, are "mixed" with projections. For example an expensive but highly selective selection should be done early while expensive projections should be delayed.

We will use the following syntax for the *filter* function in order to make PIL programs more readable ($expr(v_1, \ldots, v_n)$ represents an expression containing $v_1, \ldots, v_n$ as free variables). A type T_i is declared for each variable v_i.
for $v_1 : T_1$ **in** $b_1, \ldots, v_n : T_n$ **in** b_n
where $p(v_1, \ldots, v_n)$
apply $f(v_1, \ldots, v_n)$

Example 5 (Generalized Selection): The following query retrieves the shape of all roads in the Cupertino area. The Cupertino region is approximated by a rectangle and specified by giving the end-points of a diagonal of the rectangle. The $intersect(Cell, \{\langle\langle Float, Float\rangle\rangle\} \rightarrow Boolean$ function determines whether a cell intersects a rectangular window.

```
begin
    var Cupertino: {⟨⟨Float, Float⟩⟩};
    Cupertino:= {⟨⟨37.1500, 122.0000⟩⟩, ⟨⟨372230,1220730⟩⟩};
    for r: RoadSeg in Roads
    where intersect(r.segment, Cupertino)
    apply shape(r.segment)
end
```

■

Example 6 (Generalized Projection): The following query smooths the shape of Page Mill Road. Short curved segments are approximated by straight lines.

```
begin
    for r: RoadSeg in Roads
    where r.name = "Page Mill"
    apply
        begin
            var s:{⟨⟨Float, Float⟩⟩};
            s := shape(r.segment);
            if (length(s) < 10.0) then [s[1], s[card[s]]
            else s
        end
end
```

■

4.2 Reduction Over Bags

Reduction is a form of iteration which permits communication between iterations. It is intended to model queries which compute aggregate properties of bags such as averages and standard deviations of elements.

Reduction is modeled a higher order function of the following form

$$reduce(f, c, b)$$

where f is an binary function, c is a constant, b is a bag.

$$reduce(f, c, \{\}) = c$$
$$reduce(f, c, insert(e, b)) = f(e, reduce(f, c, b))$$

reduce iterates over all elements e of the bag b. The function $f(e, v)$ is evaluated in each iteration where v is the value returned by $f(e', v')$ in the previous iteration. The first iteration uses c as the value returned by the previous iteration.

Example 7 $reduce(+, 10, \{2, 4\})$ evaluates to 16. ■

Reduction also offers the opportunity for cloned execution provided that compiler knows the identity element, *id*, for f. The bag may be partitioned and clones of f may be used to reduce each partition as follows:

$$reduce(f, c, b) = reduce(f, c, \{r_1, \ldots, r_m\}))$$

where $r_i = reduce(f, id, Split(b, i))$

We will use the following syntax for reduction to make PIL programs more readable.

with $v_2 : T_2 := c$;
for $v_1 : T_1$ **in** b
reduceby $v_2 := f(v_1, v_2)$

Example 8 (Average): The following query returns the average length of road segments. The sum and count of the road segments is computed into the variable sc and then the average is computed by division.

```
begin
    var sc:⟨⟨ Float,Integer ⟩⟩;
    sc :=
        with sum:Float := 0; count:Integer := 0
        for r: RoadSeg in Roads
        reduceby
            begin
                sum := plus(sum, length(shape(r.segment)));
                count := plus(count, 1)
            end;
    divide(sc[1], sc[2])
end
```

■

Reduction iterates over bag elements in an arbitrary order. This leaves open a degree of freedom for exploitation by the compiler. Note that it is possible for the result of an expression to depend on the order of evaluation.

Example 9 The expression
with v_2:Integer:= c;
for v_1*:Integer* **in** $\{2, 4\}$
reduceby $v_2 := minus(v_1, v_2)$

evaluates to either $4 - (2 - 1) = 3$ or to $2 - (4 - 1) = -1$ depending on the order of iteration. ■

Such "ambiguity" is avoided if f is a *commutatively reducible* function.

Definition: A function f is termed *commutatively reducible* if for an arbitrary bag b and constant c,

$$reduce(f, c, b)$$

produces the same result for any order of iteration.

Lemma: A function f is *commutatively reducible* if and only if for arbitrary c, e_1 and e_2

$$f(e_1, f(e_2, c)) = f(e_2, f(e_1, c))$$

The following lemma follows easily. Note that this lemma is a sufficient condition but is not necessary. A counter example is $v - e$ which is commutatively reducible though minus is neither associative nor commutative.

Lemma: If f is an associative and commutative function then f is a commutatively reducible function.

4.3 Grouping over Bags

The higher order function

$$groupBy(f, b)$$

partitions the bag, b, into equivalence classes based on the result of applying the function, f, to each of its elements. The result is a bag of bags.

$groupBy(f, b) = \{p_1, \ldots, p_k\}$ where $p_1, \ldots, p_k$ is a partition of b (that is, $p_1 \cup \ldots \cup p_k = b$ and $i \neq j \Rightarrow p_i \cap p_j = \emptyset$); the elements in each partition, p_i, return the same value when f is applied to them ($c_1, c_2 \in p_i \Rightarrow f(c_1) = f(c_2)$); and elements from different partitions return different values ($c_1 \in p_i \wedge c_2 \in p_j \wedge i \neq j \Rightarrow f(c_1) \neq (c_2)$).

Grouping provides a natural granularity for cloning of subsequent operations.

Example 10 The following query computes a bag in which each tuple contains the name of a road and its length.

begin
    function roadLength(R:{roadSeg}) → Float **as**
        with len:Float := 0
        for r: roadSeg **in** R
        reduceby len := plus(len, length(shape(r.segment)));

    function roadName(r:⟨⟨ name:String, segment:Cell⟩⟩) → String **as** r.name;

    for p: roadSeg **in** groupBy(Roads, roadName)
    apply ⟨⟨ bag_pick(p).name, roadLength(p)⟩⟩
end

■

groupBy is also useful for creating nested objects. For example we could replace the second last line in the above program by

    apply ⟨⟨ bag_pick(p).name,
            for r:roadSeg **in** p
            apply r.2
        ⟩⟩

4.4 Simple Iteration, Reduction, and Grouping Over Sequences

In this section, we extend the definitions of filter, reduce and groupby for sequences. We also show that directly supporting sequences not also makes some queries easier to write but also makes it easier for a compiler to obtain efficient execution plans.

Opportunities for cloned parallelism while iterating over sequences are similar to the opportunities available in iteration over bags.

Each element of a sequence has an associated index value. For example in the sequence $[5, 7]$, the element 7 has the associated index value 2. The abstractions for iteration and reduction over sequences are similar to those for

$stockOpen : [\langle\langle name : String, date : Integer, price : Integer\rangle\rangle]$
$stockClose : [\langle\langle name : String, date : Integer, price : Integer\rangle\rangle]$

Figure 5: Logical Schema for the StockMarket Database

bags with two differences. Firstly, in addition to the value of an element, the index value of the element may also be used in the iteration. Secondly, the order of iteration is well defined.

The function

$$filter(f, p, s_1, \ldots, s_n)$$

is used for pure iteration over sequences. It takes n bags $s_1, \ldots, s_n$ and two $2n$-ary functions f and g as parameters. The functions are $2n$-ary since both the element and its index are passed as arguments to the functions. The $filter$ function iterates over the elements of the sequence in the well defined order. If i_j is the index for an element of s_j, the index values $i_1, \ldots, i_n$ are traversed with i_n changing the fastest and i_1 the slowest [7]. In each iteration, it is tested whether $p(s_1[i_1], i_1, s_2[i_2], i_2, \ldots, s_n[i_n], i_n)$ is true and if so then $f(s_1[i_1], i_1, s_2[i_2], i_2, \ldots, s_n[i_n], i_n)$ is computed. The result is the sequence of all such results of computing $f()$.

The syntax for simple iteration over sequences is the same as for bags.

The examples below use the logical schema for the Stock Market database (Figure 5). We assume that bags are ordered by name as the primary sort key and date as the secondary key.

Example 11 (Merge) The following query computes, for each stock, the change in price for each day.

for o:⟨⟨ name:String, date:Integer, price:Integer ⟩⟩ **in** stockOpen,
c: ⟨⟨ name:String, date:Integer, price:Integer ⟩⟩ **in** stockClose
where o.index = c.index
apply $\langle\langle o.name, o.date, (c.price - o.price)\rangle\rangle$

This query may also be posed fairly easily even if *stockOpen* and *stockClose* were modeled as bags,by using the predicate $o.date = e.date \ and \ o.name = e.name$. The next example illustrates how sequences are useful as compared to bags. ■

Example 12 (Shifted Merge) The following query computes the change in stock price between the closing price on one day and the opening price on the *next* day. Sequences make the notion of *next* quite simple since the next element after an element with index i is the element with index $i + 1$.

for o:⟨⟨ name:String, date:Integer, price:Integer ⟩⟩ **in** stockOpen,
c: ⟨⟨ name:String, date:Integer, price:Integer ⟩⟩ **in** stockClose
where o.index = c.index+1
apply $\langle\langle o.name, o.date, (c.price - o.price)\rangle\rangle$

[7] More precisely the index values $i_1, \ldots, i_n$ are traversed in the order defined by $i_n + card(e_{n-1}) * (i_{n-1} + \ldots + (card(e_2) * (i_2 + card(e_1) * i_1)) \ldots)$.

Defining the next date without using index is quite complex since the Stock Market is closed on weekends and some other holidays. Therefore the next date has to be computed as the minimum of all dates which are greater than the given date. ■

Reduction over a sequence is similar to reduction over a bag with the difference that the iteration is ordered. It is a higher order function of the following form

$$reduce(f, c, s)$$

where f is a binary function, c is a constant, s is a sequence.

$reduce(f, c, []) = c$
$reduce(f, c, insert(e, 1, s)) = f(e, reduce(f, c, s))$

The syntax for reduction over sequences is the same as for bags.

Example 13 (Moving Average) The following query computes the k-day moving average of the closing price of HP stock.

```
begin
var hpstock:[⟨⟨name : String, date : Integer, price : Integer⟩⟩];
hpstock :=
    for c:⟨⟨ name:String, date:Integer, price:Integer ⟩⟩ in stockClose
    where c.name = hp
    apply c;
for c2:⟨⟨ name:String, date:Integer, price:Integer ⟩⟩ in hpstock
where c2.index >= k
apply ⟨⟨c2.date,
    divide( with sum : Integer := 0
            for c3: ⟨⟨ price:Integer ⟩⟩ in
                for c1:⟨⟨ name:String, date:Integer, price:Integer ⟩⟩ in hpstock
                where c1.index <= i and c1.index > i − k
                apply c1.price
            reduceby sum := sum + c3,
            k)⟩⟩
end
```

■

Grouping over sequences is similar to grouping over bags. The difference is that the result is a bag of sequences with the elements of a sequence being ordered in the same relative order as in the original (input) sequence.

4.5 While Loop

The higher order function, $while(c, p, f)$, provides a general iteration construct. The generality lies in the fact that the domain of iteration is arbitrary. We will use the following syntax for while loops in PIL programs.

with $v : T := c$
while $p(v)$
$v := f(v)$

The *while* function is defined as:

$$while(c,p,f) = c \quad \text{if p(c) is false}$$
$$while(c,p,f) = while(f(c),p,f) \quad \text{otherwise}$$

Example 14 The following program finds all road segments within a radius of distance 10 miles from a starting point such that it is possible to drive to the road segment without getting more than 10 miles from the starting point at any time during the drive.

```
begin
    var startPoint:[⟨⟨ latitude:Float, longitude:Float⟩⟩];
    var nearByRoads: {Cell};
    var connectedRoad: {Cell};
    startPoint:= [ 37.5, 122.0 ];

    nearByRoads :=
        for  r: RoadSeg in Roads
        where distance(r.segment,startPoint ) < 10
        apply r.segment;

    connectedRoads :=
        with new :=
            for  r:Cell in nearByRoads
                where pointInCell(startPoint, r.segment )
                apply r.segment
        while not(empty(minus(new, connectedRoads)))
            begin
                connectedRoads:= union(connectedRoads, new);
                new :=
                    for  r1: Cell in new, r2:Cell in nearByRoads
                    where intersect(r1.segment, r2.segment)
                    apply r.segment
            end
end
```

■

5 Conclusion

We have used a RISC design philosophy in choosing optimizable and parallelizable language features. We have shown how these features facilitate optimization and parallel execution through cloning.

We have combined a functional model of computation with a data model that directly supports sequences in addition to bags and tuples. Our choice of a functional model permits side-effect free (sub)expressions to have a declarative reading. We have also chosen a language semantics in which, even in the presence of updates, the order of evaluation is only partially defined in order to increase the space of possible executions.

An implementation of PIL is almost complete and is expected to be ready before the workshop. We are also implementing a translator from OSQL to PIL to validate our belief that "pre-processing" of object-oriented features is a good idea.

We are investigating mechanisms for the integration of foreign types, variables and functions in PIL. This feature is important towards achieving goal of integrating pre-existing data managers in the Papyrus. The semantics of updates to aggregate objects is also a subject of current research.

The investigation of optimization techniques for parallel execution is another important area of current work. Some preliminary results which provide a theoretical framework for the problem are reported in [GHK92].

Acknowledgements: We thank Marie-Anne Neimat for her advice, encouragement and support from the early stages of this research. We are particularly grateful to Bill Kent for several enlightening discussions. Thanks are also due to Stefano Ceri, Surajit Chaudhuri, Michael Heytens, Curtis Kolovson, Spyridon Potamianos, and Donovan Schneider. Comments from Surajit Chaudhuri, Michael Heytens, and Marie-Anne Neimat helped in substantially improving an earlier draft of the paper.

References

[A+91] H. Abelson et al. Revised^4 report on the algorithmic language scheme, November 1991.

[ABD+89] M.P. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The Object-Oriented Database Manifesto. In *Proceedings of DOOD*, December 1989.

[ACO85] A. Albano, L. Cardelli, and R. Orsini. Galileo: A Strongly Typed, Interactive Conceptual Language. *Transactions on Database Systems*, 10(2):230–260, 1985.

[AGO90] A. Albano, G. Ghelli, and R. Orsini. Objects and Classes for a Database Programming Language. Technical report, Progetto Finalizzato Sistemi Informatici E Calcolo Parallelo, October 1990. Report n. 5/24.

[AS85] H. Abelson and G.J. Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, 1985.

[BBKV87] F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez. FAD, A Simple and Powerful Database Language. In *Proceedings of the Thirteenth International Conference on Very Large Data Bases*, 1987.

[BW88] R. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice Hall, 1988.

[CCRLZ89] S. Ceri, S. Crespi-Reghizzi, G. Lamperti, and R. Zicari. ALGRES: An Advanced Database System for Complex Applications. Technical report, Dipartmento di Elettronica, Politecnico di Milano, 1989. Report n. 89-010.

[CHK+91] T. Connors, W. Hasan, C. Kolovson, M.-A. Neimat, D. Schneider, and K. Wilkinson. The papyrus integrated data server. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, December 1991.

[CS92] T. Connors and D. Schneider. The Papyrus Query Processing Engine. In *Proceedings of the Second International Workshop on Research Issues on Data Engineering: Transactions and Query Processing*, February 1992.

[FAC+89] D. H. Fishman, J. Annevelink, E. Chow, T. Connors, J. W. Davis, W. Hasan, C. G. Hoch, W. Kent, S.Leichner, P. Lyngbaek, B. Mahbod, M. A. Neimat, T. Risch, M.C. Shan, and W. K. Wilkinson. Overview of the Iris DBMS. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Languages, and Applications*. Addison-Wesley Publishing Company, 1989.

[GHK92] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query Optimization for Parallel Execution. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, June 1992. To Appear.

[Has92] W. Hasan. Papyrus Interface Language. Technical report, HP Laboratories, 1992. HPL-DTD-92-4.

[Ken] W. Kent. Personal Communication and Several Unpublished Memos.

[MBCD89] R. Morrison, A.L. Brown, R.C.H. Connor, and A. Dearle. Napier88 Reference Manual. Technical report, Department of Computational Science, University of St. Andrews, 1989.

[PJ87] S.L. Peyton-Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[SC92] D. Schneider and T. Connors. Managing query execution for an advanced database programming language, 1992. Submitted for publication.

[SD89] D. A. Schneider and D. J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *ACM SIGMOD*, Portland, Oregon, June 1989.

[WLH90] K. Wilkinson, P. Lyngbaek, and W. Hasan. The Iris Architecture and Implementation. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):63–75, March 1990.