

The Functional Approach to Integrating Heterogeneous Data

Prof. Peter Gray
Dr. Graham Kemp*

*Dept. Computing Science
Univ of Aberdeen, SCOTLAND*

**Chalmers University, Gothenburg, SWEDEN*

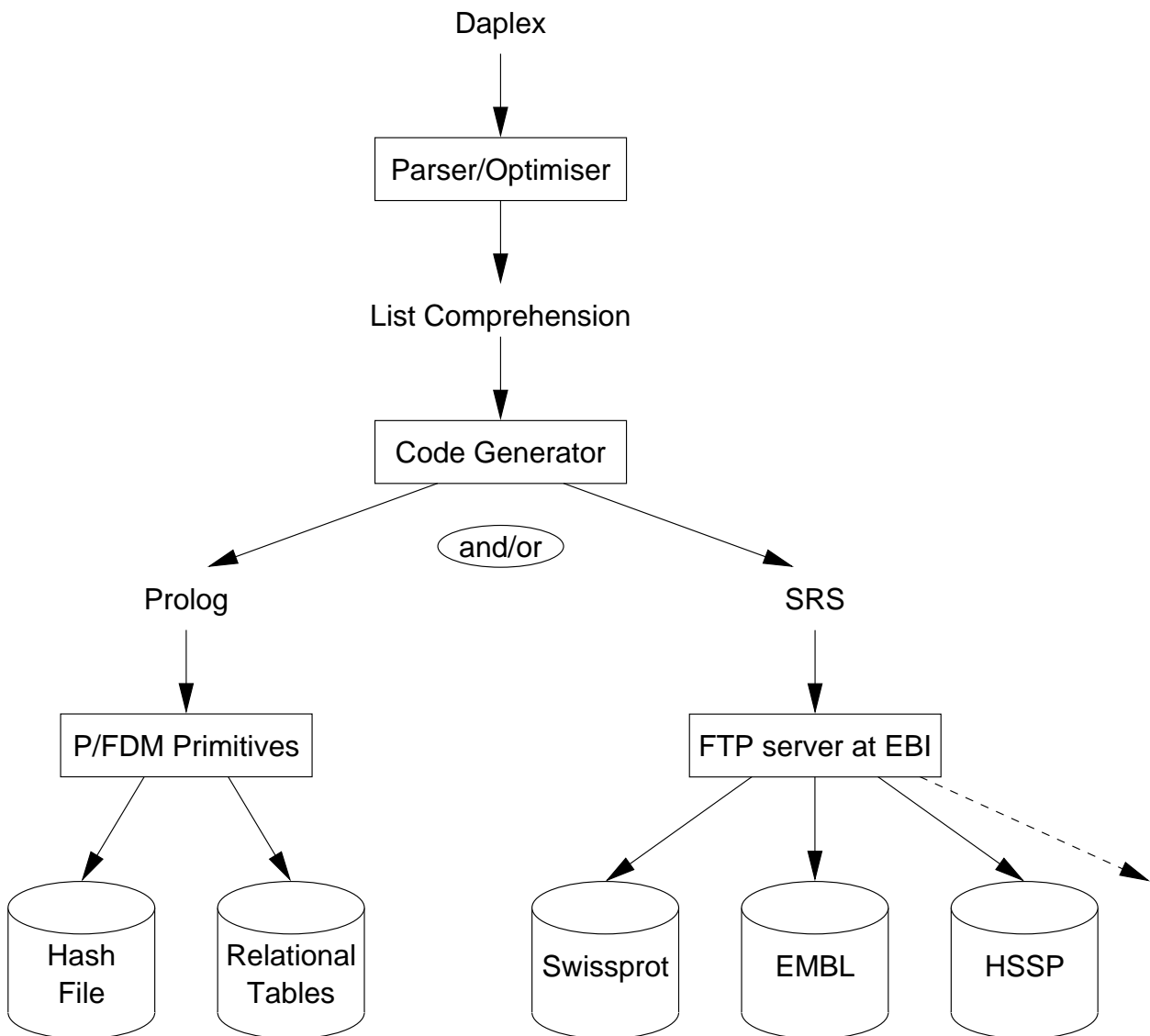
Book (September 2003) - *[http : //www.springer.de](http://www.springer.de)*

Themes:

- List Comprehensions:
advantage over Rel.Algebra;
work with Objects or Relations;
combine Calculation with Data Access.
- Functional Data Model FDM combines both
E-R and O-O Interoperability
- Mediator Architecture: P/FDM, AMOS II
- Heterogeneous growing federated DB: (Obj-
Relational)
- Based on experience with Scientific DB
(protein structure)

Themes: cont'd

- Extracting and Transforming Existential Sub-Queries for Remote Exec'n
- uses Knowledge about Domain and remote Optimiser
- Referential Transparency suits Rewrite Rules (not Prod'n rules)



A Daplex query may be translated into a Prolog query to access data held locally or SRS code to access data at EBI. However, some Daplex queries will need to access both local and remote data and so will be translated into a combination of Prolog and SRS code.

List Comprehensions

Zermelo–Fraenkel set expressions

SQL query to find the set of surnames of persons whose forename is “Jim”:

```
SELECT  surname
FROM    person
WHERE   forename = "Jim"
```

Using a list comprehension:

```
[surname(p) | p <- person; f <- forename(p);
                    f = "Jim"]
```

OR

```
[surname(p) | p <- person; forename(p) = "Jim"]
```

We can transliterate it as:

The set of values of the surname of p such that p is in the set person and f is in the set of forenames of p and f is equal to “Jim”.

List Comprehension syntax:

A list comprehension consists of a *pattern*, on the left of the | symbol, and a series of *generators* and *restrictions*, on its right.

- The pattern is an arbitrary Miranda expression.
- A generator has the form “<var> <- <list>”.
- A restriction is any Boolean-valued expression.
- Generators and restrictions are separated by ‘;’ symbols.

List Comprehension examples in Miranda with types (D.Turner, predates Haskell):

```
repeat :: num -> * -> [*]
```

```
repeat n x = [ x | y <- [1..n]]
```

```
factors :: num -> [num]
```

```
factors n = [x | x <- [1..n]; (n mod x) = 0]
```

```
cross_prod :: [*] -> [**] -> [(*, **)]
```

```
cross_prod xs ys = [ (x, y) | x <- xs; y <- ys]
```

```
filter :: (* -> bool) -> [*] -> [*]
```

```
filter pred xs = [x | x <- xs; (pred x)]
```

```
e.g. filter (>5) [2,7,1,9] == [7,9]
```

Generality of Comprehensions and Set Theory

The comprehension comes from ideas of mathematical set theory. It originated as a way of defining sets of values, starting from other well-defined sets and using some carefully chosen constructors and filters, so as to avoid the famous paradoxes of early set theory.

The values in the sets could be:

- Tuples of basic values, which suits the Relational model,
- Object identifiers, which fits with ODMG object data models,
- Tagged variant records which fit well with Semi-Structured data.
- Sets, Lists or Bags defined by other Comprehensions.

Comprehensions and Higher Order Funs

The list containing the squares of all even numbers between 1 and 20:

```
[ square x | x <- [1..20]; even x]
```

Equivalent using higher order funs:

```
(map square (filter even [1..20]))
```

```
map f xs = [ (f x) | x <- xs ]
```

Bulk Operations vs Comprehensions

We can think of the last example done in two operations:

- Start with set [1 to 20]
- Filter/Copy the even items to a new set.
- Compute a result set of squares.

Each time we store a new intermediate result set, which could get very large. The Operators are typical **Bulk Operations** like those used in **Relational Algebra**: Join, Filter, Set Union, Project, Difference...

They work well for High Hit Rates, where you need to scan nearly all tuples.

Stream Implementation

```
[1..20] ---> Filter(even) ---> Compute(square)
                               ---> [1,4,..
```

Data flows through **Pipes** and operations are performed a **Bufferfull** at a time. Thus you avoid materialising the whole intermediate result set unless you need to **Sort** it.

A sequence of `map` and `filter` can be done efficiently this way.

This is not a good method for **Selective Queries** because you lose the advantage of **Precomputed Indexes**. Its just not worth recomputing them for each intermediate set!

With a Comprehension, you can keep hang on to Object Identifiers and use indexes to look them up.

Nested Generators and Nested Loops

Generators may just be based on Finite Sets
(or Subranges of integers)

Filters can also do Calculations.

Pythagorean Triangle example

```
[(x,y,z) | x <- [1..50]; y <- [1..50];  
          z <- isqrt(x*x + y*y);  
          z*z = x*x + y*y; z <50]
```

OQL Object Query Language example

```
[name(x) | x <- students; y <- takes(x);  
          z <- taught_by(y);  
          rank(z) = "full prof"]
```

Representing Associations by Functions

The function `takes(x)`, for each value of `x` - the **Object Identifier** of a student delivers the set of related course objects.

It does what an **index** to one column in a two column relationship Relation does in a Rel. DB!

```
TAKES(Student ID,      Course ID)
```

```
=====
```

```
S117          C4
S119          C4
S119          C5
S124          C5
S124          C6
```

```
takes(S117) = [C4]
```

```
takes(S119) = [C4,C5]
```

```
takes_inv(C5) = [S119,S124]
```

Converse or Inverse Functions **XXX_inv**:

Think of a **Relation Table** $f(x : col1; y : col2)$

For value of x -Return set of vals of $y = f(x)$
i.e. Select value of x -List(Project) vals of y ;

Converse $x = f_inv(y)$ **means:**

Select value of y -List(Project) vals of x ;

Nested Loop Code for Comprehensions

- generators ($z \leftarrow \text{exp}$) = **for**
- filters (predicate $p(x,y)$) = **if**

```
result := [];  
for x in students()  
  for y in takes(x)  
    for z in taught_by(y)  
      if rank(z) = "full prof"  
        then result := result ++ [name(x)];
```

Optimise by reordering loops!

```
[name(x) | z <- lecturers;  
      rank(z) = "full prof"; %%FILTER  
      y <- taught_by_inv(z);  
      x <- takes_inv(y)]
```

Typically we follow a **path** through an **E-R** Diagram, Each **arc** is an **association** or relationship function.

Depending on the direction we traverse the arc, we may need to use a **Converse** function.

Optimisation using Index for Selection

If, instead of an Object DB, we have a **relational table Takes(S:student; C:course)** we would implement the generator `y <- takes(x)` thus:

```
for x in students()
  for t in Takes() if name(x) = S(t)
    then for z in taught_by(C(t)) ...
```

The corresponding list comprehension is:

```
[name(x) | x <- students;
      t <- takes; name(x) = S(t);
      z <-taught_by (C(t)); ...]
```

Efficiency: the implementor spots they can use the index on table Takes to **locate directly** those rows in column S containing the name x, and thus satisfy the filter `name(x) = S(t)`, **without iterating and testing** all rows in the Takes table.

Note: the writer of the query need not know how the data is stored. The **Function Abstraction** `y <- takes(x)` is sufficient!

Comprehension Theory

- Early (1975) use by Darlington and Burstall in a predecessor of Functional Programming Language **ML**.
- Use by Turner (1985) in **Miranda**. Later in **Python**.
- Paton & Gray (1990) Use in Optimiser for **P/FDM**.
- Buneman (1990) introduces **Set** and **Bag** Comprehensions.
- Buneman, Davidson & Wong. Use with **Kleisli** language for bioinformatics data.
- Wadler (1990) generalises comprehensions to algebraic **Monads**
- Grust & Fegaras (1995) use algebraic **Monoids** to optimise GROUP BY expressions.

Comprehensions in AMOS-II

AMOS-II **Active Mediator Object System**

Tore Risch et. al. Uppsala

Uses **Functional Data Model** + Functional Query Language **AmosQL** (like Object-Relational SQL).

Works as **Distributed Mediator System**, with mix of Main Memory and Persistent DB. Implemented in Lisp.

Entities and Functions are typed:

```
create type Person;  
create type Student under Person;  
create function name(Person) -> Charstring;
```

Query stored as function:

```
create function sailch(Person p) -> Charstring as  
  select name(c)  
  from Person c  
  where parent(c)=p and hobby(c) = 'sailing';
```

This is turned into a *comprehension*, treated as a *typed object calculus expression*.

Merging Comprehensions in AMOS-II

```
sailch(p) == [name(c) | c <- person; parent(c)=p;  
             hobby(c) = 'sailing']
```

Suppose information about `hobby(c)` was held in connection with `sports-person`, a subclass of `person` with a `surname` attribute, and that we could express it thus:

```
hobby(c) == [recreation(x) | x <- sportsperson;  
            surname(x) = name(c)]
```

In the comprehension form, such functions may easily be *substituted* because of referential transparency, *reordered and optimised*. These comprehensions merge into the following which can be further simplified:

```
sailch(p) == [name(c) | c <- person; parent(c)=p;  
             x <- sportsperson;  
             surname(x) = name(c);  
             recreation(x) = 'sailing']
```

Note that Risch's internal form of comprehension does *not* explicitly distinguish the use of generators, and just uses equality as in a filter.

This has the conceptual advantage that *generators are not always a fixed role*.

Rewrite rules (later) can also transform generators $\langle \Rightarrow \rangle$ filters, taking advantage of *substitutability of comprehensions*, in a systematic formal fashion.

Jython Functional Scripting Language

Jython is an implementation of the high-level, dynamic, object-oriented language Python seamlessly integrated with the Java platform.

- **Interactive experimentation** - Jython provides an interactive interpreter that can be used to interact with Java packages or with running Java applications.
- **Rapid application development** - Python programs are typically 2-10X shorter than the equivalent Java program.
- **Seamless Interaction** between Python and Java allows developers to freely mix the two languages, both in development and as shipped.

Jython List Examples

On-line Tutorial:

see <http://www.python.org/doc/tut/tut.html>

```
>>> a = ['spam', 'eggs', 100, 1234]
```

```
>>> a[3]
```

```
1234
```

```
>>> a[-2]
```

```
100
```

```
>>> a[:2] + ['bacon', 2*2]
```

```
['spam', 'eggs', 'bacon', 4]
```

```
>>> q = [2, 3]
```

```
>>> p = [1, q, 4]
```

```
>>> len(p)
```

```
3
```

```
>>> p[1]
```

```
[2, 3]
```

```
>>> p[1].append('xtra')
```

```
>>> p
```

```
[1, [2, 3, 'xtra'], 4]
```

Jython Iteration Examples

ITERATION, FIBONNACCI

=====

```
>>> def fib(n):    # Comment Function def
...     a, b = 0, 1
...     while b < n:
...         print b,
...         a,b = b, a+b
...
>>> fib(10)
1 1 2 3 5 8 >>>
```

ITERATION OVER LISTS

=====

```
>>> a = ['cat', 'window', 'defenestrate']
>>> for x in a:
...     print x, len(x)
...
cat 3
window 6
defenestrate 12
```

Map, Filter, Fold

Miranda: (map cube [1..11])

```
>>> map(lambda x: x*x*x, range(1, 11))
```

```
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

#Miranda: f x = ((x mod 2) = 0) & ...

```
>>> def f(x): return x % 2 == 0 and x % 3 != 0
```

```
...
```

```
>>> f(3)
```

```
0
```

```
>>> filter(f, range(2, 11))
```

```
[2, 4, 8, 10]
```

```
>>>
```

```
>>> def add(x,y): return x+y
```

```
...
```

#Miranda: (fold add 0 [1..11])

```
>>> reduce(add, range(1, 11))
```

```
55
```

List Comprehensions

Generator: *for* x *in* xlist

Restrict: *if* boolexp

```
>>> vec = [2, 4, 6]
```

```
>>> [3*x for x in vec]
```

```
[6, 12, 18]
```

```
>>> [3*x for x in vec if x > 3]
```

```
[12, 18]
```

```
>>> [(x, x**2) for x in vec]
```

```
[(2, 4), (4, 16), (6, 36)]
```

```
>>> vec2 = [4, 3, -9]
```

```
>>> vec1 = [2, 4, 6]
```

```
>>> [x*y for x in vec1 for y in vec2]
```

```
[8, 6, -18, 16, 12, -36, 24, 18, -54]
```

```
>>> [vec1[i]*vec2[i] for i in range(len(vec1))]
```

```
[8, 12, -54]
```

Classes+Recursion

RECURSIVE DEFN, IF:

=====

```
>>> def fac(n):
...     if (n==0): return 1
...     else: return  n * fac(n-1)
...
>>> fac(3)
6
```

OBJECT CLASS DEFS

=====

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
>>> x.r - x.i
7.5
```

Jython Modules/Scripts

See Tutorial section 6

Put your function defs and class defs in a file XXX.py (maybe also XXX.jy?)

To Import a module:

```
>>> from XXX import * # import all
>>> from XXX import cube, square
```

To see module Contents:

```
>>> dir(XXX) # e.g. dir(sys)
```

To see current names defined:

```
>>> dir()
```

To Execute a Script of commands:

```
>>> execfile('.pythonrc.py')
```

To Extend Module Search Path sys.path
initialised from env PYTHONPATH

```
>>> import sys
```

```
>>> sys.path.append('/ufs/guido/lib/python')
```