The Integration of Functional Languages and Relational Databases

A.J. Field and J.A.R. Hutton
Department of Computing (Report No. 3/96)
Imperial College Of Science, Technology and Medicine
180 Queen's Gate
London SW7 2BZ, U.K.

May 22, 1996

Abstract

The rapid increase in the use and size of relational databases is demanding increasingly fast and efficient database management systems. There is currently considerable research effort being directed towards the use of parallel processing to provide such performance improvements.

Here we extend the Haskell language and its compiler to support SQL database queries as the first step towards performing query processing in a parallel function environment. SQL queries are generated from the translation of Haskell list comprehensions at compile-time and are used to query a relational database at run-time thereby allowing a Haskell program to access and process data stored in a relational database. We shown that query processing can be partitioned between the SQL and Haskell domains and conclude that if query processing is migrated into the Haskell domain with both domains supported on the same hardware platform then access performance is reduced. However, it is proposed that if separate platforms, and in particular a parallel Haskell platform are used to support the two processing domains, then increased performance should be possible. Finally we explore some other areas of interest such as lazy database access and dynamic query construction that might be the subject further work.

1 Introduction

In this paper we describe an extension to the Haskell programming language [5] which enables a relational database to be specified and queried in a manner similar to that of SQL [11, 12]. Database records are typed using an extension to the basic tupling mechanism which allows tuple elements to be labelled and relations are modelled by lists of database records. A database is queried via list comprehensions.

Although these extensions provide the expressive power to model conventional relational databases the performance is poor due to the inappropriate representation structure used (the list) and, in particular, the database indexing mechanisms that are induced (i.e. linear list traversal).

We have therefore developed an implementation of the database facilties on top of an existing commercial RDBMS (Relational Database Management System).

In the rest of this paper we summarise the lanaguges extensions that have been introduced to model relational databases and associated queries and detail the implementation route, which

enables a relational database to be constructed and queried from within a functional language. We summarise a pilot implementation study and outline the measured performance benefits that are accrued by the use of an off-the-shelf RDBMS.

This work was undertaken as part of the COMPAQT research project (Combined Program and Query Transformation for Relational Database Programming).

2 Relational Databases, Haskell and SQL

2.1 Relational Databases

A relational database stores information in one or more tables the structure (or *schema*) of which is defined by the creator. Each table in the database has a name and consists of a number of columns (attributes) and rows (records). e.g. Table *employees*: Tables are created and

Initials	Surname	Age	Salary	Project number
AB	Allen	20	1000	1
BC	Blacklock	30	2000	2

Figure 1: Example records in the *employees* table

manipulated using Structured Query Language that passes requests to the relational database management system which maintains the database tables.

Table creation: When a table is created it is given a name and the column names and types are defined. The above example table can be created using the following SQL statement:

```
CREATE TABLE employees(initials char(3)     not null,
                       surname  varchar(16) not null,
                       age      integer     not null,
                       salary   money       not null,
                       projnum  int         not null);
```

The `not null` term indicates that a column will not accept null values which indicate missing values (e.g. where an employee's age is unknown). When this command is executed the RDBMS creates the necessary files to support the table and on completion, although the table schema has been created, the table itself does not contain any data.

Record insertion: The `INSERT` statement is used to insert a single record into a database table. For example, the first row in the table shown in Figure 1 could be inserted into the *employees* table using:

```
INSERT INTO employees VALUES ('AB', 'Allen', 20, 1000, 1)
```

Row retrieval: Rows of data are retrieved from a table in the database using the SQL "select" statement. For example, to retrieve the row inserted in the above paragraph:

```
SELECT * FROM employees WHERE surname = 'Allen'
```
or
```
SELECT * FROM employees WHERE age = 20
```

where the "*" means "all columns". Note that in these examples that if there were other employees in the table with the surname `Allen` or aged 20 then these would be retrieved as well. To retrieve AB Allen's age and project number we would use:

```
SELECT age, projnum FROM employees
WHERE surname = 'Allen'
```

The work described here is based on the Ingres RDBMS which supports SQL for creating, accessing and updating a relational database. The Ingres RDBMS environment consists of two main components:

1. A single back-end data manager/server which manages and maintains the integrity of the database and processes requests from the front-end.

2. Single or multiple front-ends which manage the user interfaces passing requests to the back-end and results to the user in the desired format.

There are three primary methods for accessing data stored:

1. General purpose front-end applications allowing the user direct interactive access to the database. These are: Query by forms (QBF), report by forms (RBF), interactive SQL (ISQL) and the visual graphics editor (VIGRAPH).

2. An application developed in the fourth generation language Ingres 4GL. This high-level language is used to control the interaction of the user with the application to manipulate data in the database and the user's display.

3. An application developed in a traditional programming language (e.g. C, COBOL, FORTRAN etc.) which includes embedded Ingres commands allowing the application to access and control Ingres databases and applications.

It is the last method that interests us here as the Haskell compiler translates the source code into equivalent C code which is then compiled by a standard C compiler. By embedding Ingres SQL commands at the C code level we enable Haskell to inter-work with an Ingres database.

2.2 Relations in a Functional Language

We begin with the Haskell programming language and introduce first an extension to allow *record* types, which are simply tagged versions of conventional product (tuple) types—these are not a feature of Haskell. The general format for a record type will be as follows:-

```
T == ( f1 :: T1, ..., fn :: Tn )
```

where, in keeping with the functional style, `f1, ... fn` are *projectors* and `T, T1, ..., Tn` are types (these may be parameterised by one or more type variables). Projector `fk` is a function of type `T -> Tk`, $1 \leq k \leq n$. Thus, if `r = (v1,...,vn) :: T` then `fk r = vk`, $1 \leq k \leq n$. We will use records such as these to model the elements of our relational database.

A relation can now be considered as a set of records in the conventional sense. In fact we use the existing list notation and, most significantly, the notation for list comprehensions which will form the basic mechanism for querying a database. The implementation distinguishes query comprehensions from standard list comprehensions.

Consider a database table such as the one shown in Figure 1. Ingres sees this as a table of 2 rows and 5 columns but it could equally be represented in Haskell as a list of 2 tuples each having 5 fields. By comparing the representations it is immediately apparent that the following general equivalences hold: The general structure of an SQL "select" statement is:

SQL representation	Haskell representation
table	list
row	tuple
column	field

Figure 2: Relationship between SQL and Haskell data representations

```
SELECT <columns> FROM <tables> WHERE <conditions>
```

The structure of a Haskell list comprehension is:

```
[ <expression> | <qualifier>, ... ,<qualifier> ]
```

where the qualifiers can be generators or filters.

Data from the database table would normally be retrieved using SQL, however, by recognising the similarity of the operation performed by the select statement and the list comprehension it can be seen that a database query can be formulated in list comprehension notation. For example, to retrieve all the rows from the table we would use the following select statement:

```
SELECT * FROM employees
```

This operation can equally be described by the Haskell list comprehension:

```
[ e | e <- employees ]
```

The columns/fields to be retrieved can be explicitly specified, e.g. "retrieve the initials and surnames of all employees":

```
SELECT initials, surname  FROM employees
```

This is equivalent to:

```
[ (initials e, surname e) | e <- employees ]
```

Single or multiple conditions/filters can be used. e.g. "retrieve all the information on employees whose salary is greater than 1000":

```
SELECT * FROM employees WHERE salary > 1000
```

is equivalent to:

```
[ e | e <- employees, salary e > 1000 ]
```

A combination of column/field specification and conditions/filters can be used. e.g. "retrieve the surname and project number of employees whose salary is greater than 1000 and age is less than 60":

```
SELECT surname, projnum FROM employees
WHERE salary > 1000 AND age < 60
```

is equivalent to:

```
[ (surname e, projnum e) | e <- employees,
salary e > 1000, age e < 60 ]
```

Finally, inter-table joins also have a list comprehension equivalent; consider an additional database table containing information on the projects employees are working on: Given this

Project number	Project name	Budget
1	AS400	£10010.00
2	BASIC	£10020.00

Figure 3: Example records in the *projects* table

table consider the query: "retrieve the surname and the budget of the project each employee is working on". In SQL this would be formulated as:

```
SELECT surname, budget FROM employees, projects WHERE
employees.projnum = projects.num
```

which is equivalent to:

```
[ (surname e, budget p) | e <- employees, p <- projects,
projnum e == num p ]
```

The above examples show how the most common forms of the SQL select statement used to interrogate a database can be represented as Haskell list comprehensions.

It is the equivalent access mechanisms of Haskell and an Ingres SQL database that enables the implementation of an interface between the two domains via translation of list comprehensions to SQL query statements.

2.3 The Haskell/Ingres Interface

The design of the Haskell/Ingres interface is divided into two major components:

1. Compile-time list comprehension to SQL query translation.
2. Run-time SQL query processing.

The structure of the interface is shown in Figure. 4 and the design of the major components are introduced below. The implementation of the above components is described in Section 2.6 and 2.8.

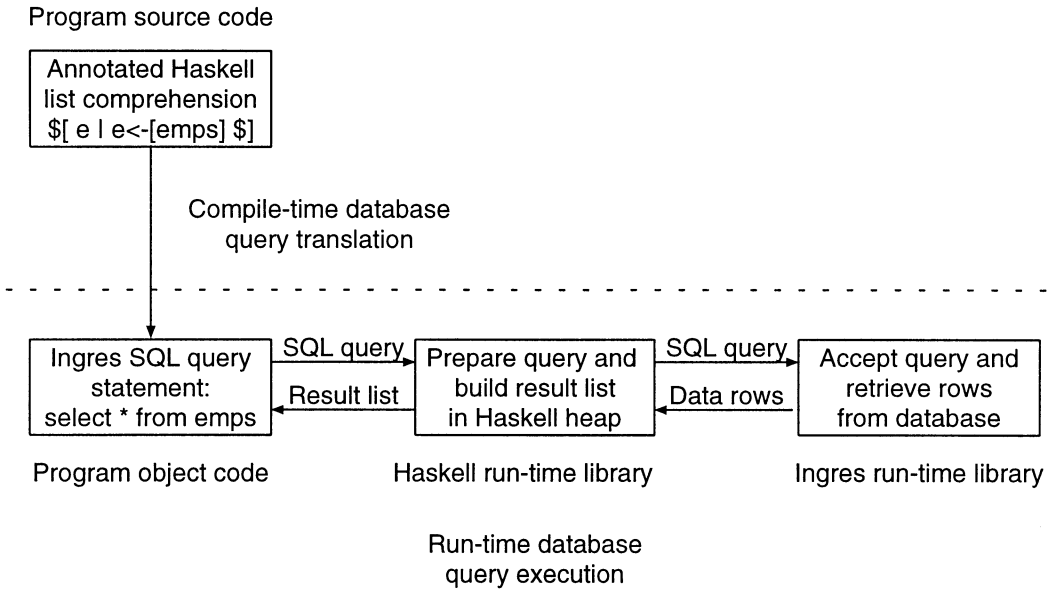

Figure 4: High-level Haskell/Ingres interface design

2.4 Compile-time List Comprehension to SQL Query Translation

The Haskell list comprehension syntax is extended using `$[` and `$]` to identify those list comprehensions that represent database queries. This involves an extension to the Haskell front end. These are translated into corresponding Ingres SQL database query statements during the compilation of the Haskell program source code.

During execution, the Haskell program calls the Haskell run-time library (RTL) passing the SQL query statement as an argument. The Haskell RTL in turn calls the Ingres RTL which processes the database query and returns the resulting database rows. The Haskell RTL converts these rows into a corresponding Haskell list structure which is finally returned to the calling Haskell program.

2.5 Translation of List Comprehensions

The file structure of the FAST compiler is shown in Figure 5. This shows the locations of the files referenced in the following sections.

Translation of DQLCs into corresponding SQL "select" statements takes place following program parsing and type checking. On identifying a DQLC the following translation is performed:

From section 2.2 the general structure of a Haskell list comprehension is:

```
$[ <expression> | <qualifier>, ... ,<qualifier> $]
```

where the qualifiers can be either generators or filters. *Tquery* processes the DQLC building up the corresponding SQL query string as follows:

1. Checks that the LC has at least one qualifier.
2. Creates an environment mapping variable names to table names for each generator in the qualifier list. Counts the number of generators and filters in the query.

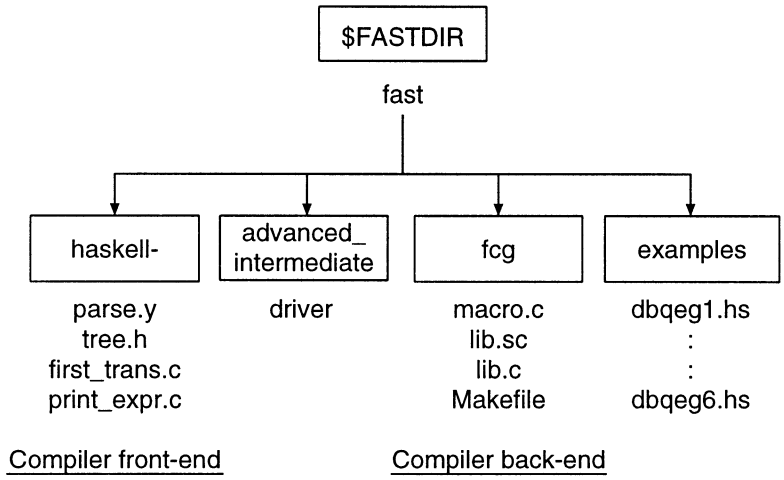

Figure 5: FAST compiler directory structure

3. Translates `<expression>`
   Where the expression is a single variable this is translated to:

   ```
   SELECT *
   ```

   Where the expression is a single selector applied to a variable and only one table is involved in the query this is translated to:

   ```
   SELECT column_name
   ```

   Where the expression is a bracketed list of selectors applied to a variable it is translated to:

   ```
   SELECT col_name1.table_name1, col_name2.table_name2 ...
   ```

4. Translates generator qualifiers:
   Variables are translated to their equivalent table names via the environment mapping to:

   ```
   FROM table_name1, table_name2 ...
   ```

5. Translates filter qualifiers:
   If there are no filters in the query then no `WHERE` clause is appended to the query string. A single filter is translated to:

   ```
   WHERE filter_expression
   ```

   Multiple filters are translated to:

   ```
   WHERE filter_expression1 AND filter_expression2 ...
   ```

6. Converts the complete query string into a list of characters to make it transparent to the back-end of the compiler. This process is shown in Figure 6

The translation of the filters involves printing the filter expressions. This isstraightforward although some translation is required to generate the correct infix notation. An example translation is shown below:

```
$[ (surname e, name p) | e<-employees, p<-projects,
                         projnum e == num p,
                         age e /= 40,
                         budget p >=1000 $]
```

which becomes:

```
SELECT employees.surname, projects.name
FROM projects, employees
WHERE employees.projnum = projects.num
AND employees.age != 40
AND projects.budget >= 1000
```

2.6 Run-time Database Query Processing

A database query function in the Haskell run-time library forms the bridge between a Haskell program and the Ingres database server. This function accepts query statements from the calling program and passes them to the Ingres run-time library which processes the query and returns the resulting rows of data. The function then packages up the data into a suitable list structure in the heap before returning to the main program.

Embedded Ingres allows SQL statements to be incorporated within a general purpose host language such as C. The provision of this facility requires that the host program source code is passed through an Ingres specific pre-processor before being passed through the normal host language compiler. The pre-processor translates the embedded SQL code into the appropriate Ingres run-time library calls whilst leaving the host language code untouched. In the case of C, the pre-processor expects the source code to be contained in a file with a *.sc* suffix and this is pre-processed to a file with a *.c* suffix suitable for submission to the C compiler.

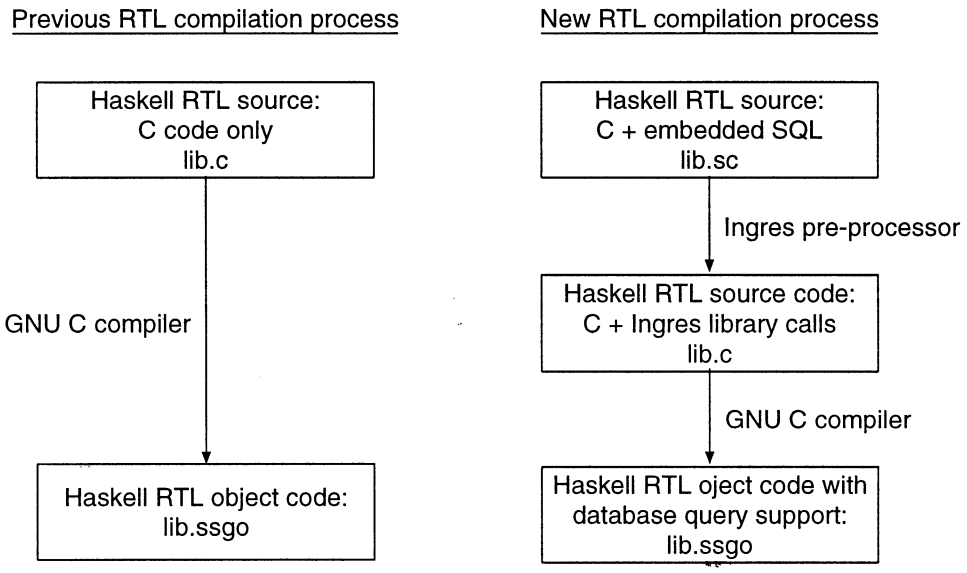

Figure 6: Modification of the Haskell run-time library compilation process

A new function is required in the Haskell run-time library (HRTL) to prepare dynamically and execute SQL queries on an Ingres database. *sql_query* is called with two arguments:

1. A pointer to the list of characters that represents the SQL query statement.
2. The current value of the Haskell heap pointer.

The function operates in the following manner:

1. The Haskell list of characters that represents the database query is converted into an equivalent query statement string.
2. Using Ingres run-time library calls a connection to the Ingres database server is established.
3. The query statement is initially submitted to Ingres which checks that it is a valid query and returns the number and type definitions of the columns that will be returned by the query. This enables the HRTL to allocate appropriate temporary storage for a row of data when each is later retrieved from the database.
4. The HRTL accesses the results of the query using an Ingres "cursor" to retrieve individual rows of data which are built into the query result list in the heap.
5. When all the rows have been retrieved the connection to the database server is closed.
6. A pointer to the result list in the heap and the updated value of the heap pointer are returned to the calling program.

3 Interface Operation and Performance Analysis

We illustrate the use of the interface by means of a simple example which is subsequently developed to show how the processing of a query can be migrated between the SQL and Haskell domains. Initially all the processing of the query is performed in the SQL domain. Subsequently, through a series of changes to the database query list comprehension, the processing is systematically moved into the Haskell domain. Only Haskell program fragments are shown.

4 Example Query Processing

The queries all retrieve data from a table *employees* which comprises 5000 rows by 5 columns containing a mixture of hand-produced and synthetically generated data.

4.1 Simple Query with All Processing Performed in the SQL Domain

Consider a query on the *employees* table to extract the ages of the employees who have a salary of £12500. This can be formulated in the following database query list comprehension:

```
showlist $[ age e | e <- employees, salary e == 12500 $];
```

where *showlist* is simply a function to display the results of the query. At compile time this is translated to:

```
SELECT age FROM employees WHERE salary = 12500
```

It can be seen that all the processing of the query to yield the required list of ages is undertaken in the SQL domain.

```
nproc = 1, available = 524288, max_task_size = 524288
66  56  46  36  26  67  57  47  37  27
Time : 0.91 seconds
Sim time: 0
proc0: heap usage:  310 words  (1240 bytes)
```

This shows the ages of the 10 employees who have salaries of £12500 and that the program required 310 words of heap space (where 1 word is 4 bytes). In this case the result list built up in the heap by the *sql_query* in the run-time library is simply a list of 10 integers (the ages of the employees). The times displayed in these example outputs are misleading as they are based on elapsed time rather than processor time; an initial performance investigation is covered in section 5.

4.2 Migration of Column Selection into the Haskell Domain

Using the same query as in the previous section the column selection processing can be migrated into the Haskell domain as follows:

```
showages $[ e | e <- employees, salary e == 12500 $];
```

When the program is compiled and executed, the following output is produced:

```
nproc = 1, available = 524288, max_task_size = 524288
66  56  46  36  26  67  57  47  37  27
Time : 0.87 seconds
Sim time: 0
proc0: heap usage:  530 words  (2120 bytes)
```

Note that the query result is identical (as it should be) but the heap space used is increased to 530 words. This is because *sql_query* builds a list of 10 tuples including all the data from the 5 columns of the *employees* table.

4.3 Migration of Row Selection into the Haskell Domain

Having migrated the column selection into the Haskell domain it is possible to do the same thing with the row selection. Consider firstly using a separate function to filter out the required rows from the table:

```
filtersal $[ e | e <- employees $]
```

The output when executed is:

```
nproc = 1, available = 524288, max_task_size = 524288
66  56  46  36  26  67  57  47  37  27
Time : 18.19 seconds
Sim time: 0
proc0: heap usage:  130348 words  (521392 bytes)
```

Again, the query result is the same but the heap space used has increased considerably to 130348 words as *sql_query* is now building the entire *employees* table in the heap (that is 5 columns by 5000 rows).

As an alternative, consider embedding the database query list comprehension within an ordinary list comprehension to perform the row filtering in place of using a separate function:

```
showages [ n | n <- $[ e | e <- employees $],
                    salary n == 12500 ];
```

Execution output is shown below:

```
nproc = 1, available = 524288, max_task_size = 524288
66  56  46  36  26  67  57  47  37  27
Time : 14.50 seconds
Sim time: 0
proc0: heap usage:  120342 words  (481368 bytes)
```

As can be seen the result is the same and the heap space used is similar at 120342 words to that using a separate function.

4.4 Character String Predicate Handling

Haskell "character strings" (lists of characters) cannot be compared using the numerical relational operators (== etc.). A string comparison function is defined for direct use in the Haskell domain and for translation when used in a database query list comprehension. The use of this *strcomp* function in the two domains is shown in the following examples that extract the surnames of employees who have the surname "AAAAAA":

```
showinitials $[ surname e | e <- employees,
       strcomp (surname e) (''AAAAAA'') $]
```

Note that the *strcomp* function has to be fully defined despite not being used during execution as it is translated at compile-time to:

```
SELECT surname FROM employees WHERE initials = 'ZZ'
```

In this case all the selection processing is carried out in the SQL domain and the output from the program is:

```
nproc = 1, available = 524288, max_task_size = 524288
AA  AA  AA  AA  AA  AA  AA
Time : 1.12 seconds
Sim time: 0
proc0: heap usage:  469 words  (1876 bytes)
```

As before the selection processing can be moved into the Haskell domain:

```
showinitials [ e | e < $[ emp | emp <- employees $],
                      strcomp (surname e) (''ZZ'') $]
```

the program output is:

```
nproc = 1, available = 524288, max_task_size = 524288
AA  AA  AA  AA  AA  AA  AA
Time : 17.22 seconds
Sim time: 0
proc0: heap usage:  120341 words  (481364 bytes)
```

Note: In this case the Haskell definition of *strcomp* is used directly in the row selection processing. The query result is the same but again the heap space used is much greater as the whole *employees* table is built in the heap.

5 Interface Performance Analysis

We now present the results of some perliminary performance testing. We have shown that different query formulations can use radically different amounts of heap space to store rows from the database tables. Since this list in the heap has to be constructed by *sql_query* and accessed by the calling program it is not surprising that the performance of the interface is closely linked to the size of this list. The execution times and heap usage for the programs containing the example queries are summarised in Figure 10. (The execution times are the sum of the user and system times measured by the Unix program `/usr/bin/time` and averaged over 10 executions.)

Program name	Query processing	Execution time (secs)	Heap usage (words)
dbqeg1.hs	All processing in the SQL domain	0.24	310
dbqeg2.hs	Column selection in the Haskell domain	0.25	530
dbqeg3.hs	Col + row selection in Haskell using function	6.08	130348
dbqeg4.hs	Col + row selection in Haskell using list comprehension	6.17	120342
dbqeg5.hs	String comparison in row selection in SQL	0.28	469
dbqeg6.hs	String comparison in row selection in Haskell	6.17	120341

Figure 7: Execution time and heap usage of the example query programs

The first two examples construct the smallest lists in the heap and hence they are the fastest. When the row selection is moved into the Haskell domain, the whole of the *employees* table is constructed in the heap and the execution time is approximately 24 times longer. Similarly, string comparison queries processed in the SQL domain are faster than those processed in the Haskell domain. In practice the interface can be trivially modified to build a minimal list in the heap.

There are two main factors causing the reduced query processing performance as the processing is migrated into the Haskell domain:

1. All the result rows from the database are retrieved and built into the result list before the Haskell domain starts to process them and hence query processing in the two domains proceeds in an entirely *sequential* manner.

2. Both the SQL and Haskell processing domains are supported on the same shared hardware platform.

This suggests that separation of the processing platforms and parallelisation of the operation of the two domains would improve performance and this is discussed in Section 6, Further Work.

The current implementation is prototype in nature and the database name is currently hard-coded in *lib.sc*. The database can easily be altered to incorporate additional or longer tables

through interactive SQL (ISQL) or embedded SQL programs so that for experimental purposes this is not a serious limitation.

6 Future Work

As the type information is actually supplied from the database, it is necessary to ensure that Haskell variables used in database queries have types corresponding to the column types in the database table otherwise a mismatch will occur at some stage. e.g. If a Haskell variable *empid* is defined as a character and the database table column *empid* is defined as an integer then the query:

```
$[ empid e | e<-[ employees ] $]
```

which is translated to

```
SELECT empid FROM employees
```

will fail because *sql_query* will return a list of integers whereas the Haskell program will be expecting a list of characters. This error may not be noticed until the Haskell program attempts to process the list. It would be useful to provide some form of compile-time database query type-checking to help prevent problems caused by such mismatches.

An initial problem with the interface was that it could only return a very limited number of rows of data (2 - 8 depending on the row length). This was due to a limited `default_page` size of 32 words in *lib.sc*. This meant that any return lists longer than 32 words ended up overwriting memory outside the legally allocated heap. For the short-term this has been overcome by increasing it to 524288 words (which enables it to store the entire *employees* table) and incorporating a heap pointer check in *sql_query* just prior to its completion. This is not a satisfactory in the long-term as queries resulting in a large number of rows will cause the same problem. A better solution would allow *sql_query* to dynamically request additional pages of heap space when required.

Ingres allows tables to be defined with columns which accept "null" entries which have no defined value (e.g. an employee might have a null salary which may indicate that it is unknown but it does not necessarily suggest that it is zero). Haskell does not support such a concept and so the columns in the test database tables should all be defined as "not null". *sql_query* in *lib.sc* will trap columns that allow null entries and print an error message and exit. Null values could be supported in Haskell by the definition of a set of axioms for handling them under different circumstances and the extension of *sql_query* to handle them. However, consider the list:

```
[ 3, null, 9]
```

The sum of the elements of this list might easily be defined as 12, but the average of the elements could be calculated as either 4 or 6 depending on whether a null is counted in the number of elements in a list. Hence support for null values would require careful consideration as different operations might expect nulls to be handled in different ways.

Although the prototype interface exhibits a low coupling between a Haskell program and the run-time library, this ultimately goes against the lazy evaluation philosophy of the Haskell language. This is because the *sql_query* function retrieves all the result rows from the database and returns the complete list to the calling program which is essentially *eager* in operation. The interface could be modified to allow *lazy* access to a database where each result row would be individually retrieved, packaged up and returned to the calling program for processing as

necessary. This would be advantageous in certain types of query e.g. "retrieve the name of the first employee who has a salary greater than £2000" from a large database:

1. The current interface would generate a list of all the employees who have salaries above this figure and then the Haskell program would be responsible for selecting the first one. This method introduces the possibility of considerable unnecessary processing in the Haskell and Ingres RTLs.

2. A lazy interface would retrieve the first row that fulfilled the criteria and then pass this back to the calling program immediately, circumventing the current interface's possibility of unnecessary processing overhead.

Modification to provide lazy operation would not be trivial and would involve greatly increased coupling between the Haskell program and RTL. The *sql_query* function would have to be split into two parts, namely query preparation and row retrieval. The overall design of such an interface is shown in Figure 11.

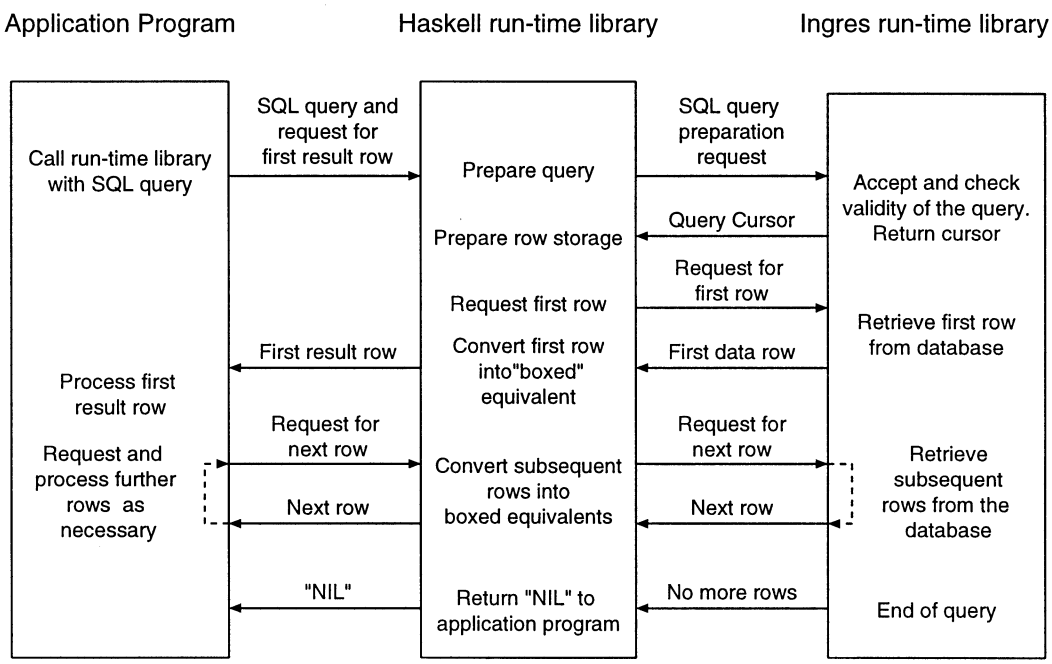

Figure 8: High-level design of a *lazy* Haskell/Ingres interface

This type of design poses some interesting questions regarding query termination. There appear to be 3 scenarios:

1. A query terminates normally after all the appropriate rows have been returned. In this case the *sql_query* could simply return 'nil' to indicate termination to the application program.

2. The application program could request termination once it had received as many rows as it required. Here, *sql_query* would have to close the cursor that points into the table of result rows prematurely.

3. Undefined termination: in a true lazy processing environment, *sql_query* will not know if any more rows will be requested from the query and indeed, the application program may not know either. This could result in a deadlock situation where effectively the query never terminates properly leaving the query cursor open indefinitely.

The major advantage of a lazy access mechanism would be realised where multiple servers or a parallel application platform are in use where the actual database query *access* and subsequent query *processing* are separated between different platforms. In these cases, the database server simply retrieves full rows from the database and passes them straight on to one or a number of other processors which perform the filtering element of the query process. This would allow a degree of parallel access and simultaneous processing of data in the query.

The prototype interface only allows queries to be statically defined at compile-time as the variables have to be defined *within* the list comprehension itself. Dynamic query construction would allow queries to be built at run-time depending on the results of other processing and in particular the results of other queries. For example, take the query:

```
$[ e | e <- employees, age e < avg_age $]
```

where `avg_age` is the average age of all employees. `avg_age` could be obtained using an SQL sub-query and the SQL "average" aggregate function but this would go against the philosophy of trying to move query processing into the Haskell domain and also `avg_age` might be required on a number of occasions. For these reasons it is more attractive to calculate and pass `avg_age` in the Haskell domain.

Dynamic query construction would also enable a Haskell progam to dynamically control the balance of processing in the Haskell and SQL domains by varying the complexity of the queries made to the database server. In effect this would allow database access to be dynamically optimised to the database server load conditions.

Such a facility would require two main components:

1. A database query performance monitor.

2. A dynamic query constructor.

The implementation of these would be major tasks and their execution overhead might not be justified by the improvements in query performance.

Given the similar characteristics of Haskell lists and database tables as explained in section 2.2, tables could provide a non-volatile storage mechanism for Haskell lists. The implementation of such a facility could be separated into the following phases:

A similar method to that used to extract data from a table could be used to support the reverse operation. Consider the list comprehension:

```
#[ e | e <- employees #]
```

where `#[` and `#]` denote a database insertion rather than retrieval. This could be translated at compile-time to:

```
INSERT INTO employees
VALUES (:initials, :surname, :age, :salary, :projnum)
```

In this case the Haskell run-time library can simply take each tuple in the list and issue an `INSERT` statement to the database server to insert the data from each tuple in the Haskell *employees* list into the *employees* table. Extending the idea to include a filter in the list comprehension:

```
#[ e | e <- employees, age e > 40 #]
```

would be translated to:

```
INSERT INTO employees
VALUES (:initials, :surname, :age, :salary, :projnum)
WHERE age > 40
```

However, at run-time the library function would have to strip off, parse and process the `WHERE` clause to extract the appropriate tuples from the Haskell list before issuing `INSERT` instructions to the database server. Furthermore, if selectors are included in the list comprehension such as:

```
#[ (surname e, age e) | e <- employees, age e > 40 #]
```

this would be translated to:

```
INSERT INTO employees(surname, age)
VALUES (:surname, :age)
WHERE age > 40
```

and the RTL function would only insert two columns of data in the table necessitating that the remaining columns contain null values which as mentioned in section 6 have no equivalent in Haskell.

The Haskell language could be extended to include table creation expressions but supplying column type information could be difficult. Type information is built into data values themselves as the functional code generator [14] encodes type information in the bit pattern of a data word. This is unfortunately very non-standard and a compile-time translator would not be able to supply type information to an RTL table creation function. This facility would be less useful than tuple insertion described above as table creation is easily performed using ISQL or embedded SQL in a C program.

At present Haskell stores character strings as a list of individual characters as seen in Figure 6 for example. This storage mechanism results in the following problems:

1. Strings are expensive in terms of memory: e.g. a 4 character string requires 4 words of storage for the characters, 4 words for the list constructors and 1 word for the nil list terminator. At 4 bytes per word this gives a total of 36 bytes. In contrast, C stores strings using 1 byte per character plus 1 for the null terminator, giving a total of 5 bytes which is over 7 times more efficient in this case.

2. Strings are inefficient to process: Haskell lists consist of "boxed" constructors and "boxed" elements. To access the characters in a list requires each `CONS` to be unboxed (one subtraction operation) and then the associated character has to be unboxed (one subtraction and 1 shift operation). Again in contrast, C simply has a pointer to the first character in the string and allows much faster access to the characters in the string.

Modification of the string storage mechanism could lead to appreciable improvements in heap usage and processing efficiency where character strings are in use.

7 Summary and Conclusions

8 Summary

We have described an extension to the Haskell programming language which allows simple relations to be defined and queried using list comprehensions as a query specification lanaguge. We have also described an experimental prototype implementation in which the Haskell run-time system is integrated with the Ingres RDBMS. This has enabled the efficient implementation of relation database by exploiting the existing, and highly efficient, B-tree representation supported by Ingres as well as the SQL query optimiser which is built in.

Preliminary performance studies have demonstrated the considerable benefit which can be accrued by the use of an off-the-shelf RDBMS.

Although the interface implemented is not capable of improving database access performance this has been attributed to the use of a single processor to perform database access processing in both the SQL and Haskell processing domains. In terms of pure database access performance, the most promising avenue for further work would be the separation of the platforms used to support these domains and the implementation of a lazy database access mechanism which would allow concurrent query processing in both domains. In turn this would reduce the level of processing required of the database server and hand the job of data filtering to the potentially more efficient functional domain.

Acknowledgements

This work was supported in part by EPSRC Grant No. GR/J14448.

References

[1] T. Field, P. Kelly and H. Khoshnevisan.
COMPAQT: *Combining Program and Query Transformation for the Efficient Exploitation of Parallel Database Hardware*
SERC grant proposal Reference GR/J 14448. Oct 1992.

[2] H. Pirahesh,C. Mohan, J. Cheng, T.S. Liu and P. Selinger.
Parallelism in Relational Database Systems: Architectural Issues and Design Approaches
Proc. 2nd International Symposium on Databases in Parallel and Distributed Systems, July 1990 pp. 4-29.

[3] J. Liu.
The FAST Project Front-end Language.
Technical report, Dept. of Computing, Imperial College, 1992.

[4] H.P. Barendregt.
The Lambda Calculus - its syntax and semantics.
North Holland, 1984.

[5] P. Hudak and J. H. Fasel.
A Gentle Introduction to Haskell.
SIGPLAN Notices, 27(5):1–53, May 1992.

[6] P. Hudak, S. L. Peyton Jones and P. L. Wadler, eds.
Report on the programming language Haskell - a non-strict purely functional language, version 1.2.
SIGPLAN notices, 27(5):1–162, May 1992.

[7] R. Bird and P. Wadler.
Introduction to Functional Programming.
Prentice Hall. Hemel Hempstead, England. 1988.

[8] A. Field and P. Harrison.
Functional Programming.
Addison-Wesley. Wokingham, England. 1988.

[9] I. Holyer.
Functional Programming with Miranda.
UCL Press. London, England. 1993.

[10] P. Hartel, H. Glaser, J. Wild.
FAST Compiler User's Guide.
Technical report, Dept. of Electronics and Computing Science, University of Southampton, 1992.

[11] C. Date.
An Introduction to Database Systems. Volume 1, 5th edition.
Addison-Wesley, Reading Massachusetts, USA. 1991.

[12] Relational Technology.
Ingres/Embedded SQL User's Guide and Reference Manual.
Manchester Computing Centre, England. 1990.

[13] Relational Technology.
Ingres/Embedded SQL Companion Guide for C.
Manchester Computing Centre, England. 1990.

[14] K. Langendoen, P. Hartel.
FCG: A Code Generator for Lazy Functional Languages.
Koen Langendoen, Pieter Hartel, Proceedings of the Conference on Compiler Construction, Paderborn, Germany, pages 278–296, U. Kastens, P. Pfahler (editors), Lecture Notes in Computer Science volume 641, October 1992, Springer-Verlag