Haskell Boards the Ferry

Database-Supported Program Execution for Haskell

George Giorgidze, Torsten Grust, Tom Schreiber, and Jeroen Weijers

Wilhelm-Schickard-Institut für Informatik,
Eberhard Karls Universität Tübingen
{george.giorgidze,torsten.grust,tom.schreiber,jeroen.weijers}@uni-tuebingen.de

Abstract. Relational database management systems can be used as a *coprocessor* for general-purpose programming languages, especially for those program fragments that carry out *data-intensive* and *data-parallel* computations. In this paper we present a Haskell library for database-supported program execution. Data-intensive and data-parallel computations are expressed using familiar combinators from the standard list prelude and are entirely executed on the database coprocessor. Programming with the expressive list comprehension notation is also supported. The library, in addition to queries of basic types, supports computations over arbitrarily nested tuples and lists. The implementation avoids unnecessary data transfer and context switching between the database coprocessor and the programming language runtime by ensuring that the number of generated relational queries is only determined by the program fragment's type and not by the database size.

1 Introduction

Relational database management systems (RDBMSs) provide well-understood and carefully engineered query processing capabilities. However, RDBMSs are often operated as plain stores that do little more than reproduce stored data items for further processing outside the database host, in the general-purpose programming language heap. One reason for this is that the query processing capabilities of RDBMSs require mastering of advanced features of specialised query languages, such as SQL, in addition to the general-purpose language the application is programmed in. Moreover, query languages are often inadequately integrated into host programming languages.

An application that is programmed in the aforementioned manner may perform substantial data transfers even when the final result of the computation is very small. Instead, it may be much more efficient (and sometimes the only option when dealing with data that can not be fitted in the heap) to transfer a part of the program to the database and then let the database perform the computation. Database kernels are optimised for intra-query parallel execution and can thus very efficiently carry out *data-intensive* and *data-parallel* computations.

A number of approaches have been proposed providing for better integration of database query languages into programming languages. Well-known examples include: Kleisli [18], LINQ [21] and Links [8]. Although very successful (e.g.,

LINQ is distributed with the Microsoft .NET framework and is widely adopted), current language-integrated approaches have a number of limitations. For example, Links only permits the database-supported execution of program fragments that compute a flat result (i.e., a list of tuples of basic types), while Kleisli and LINQ support data nesting but may compile the fragment into queries whose number is proportional to the size of the queried data (i.e., they do not feature the so-called *avalanche-safety* property). In addition, LINQ's standard query operators do not maintain list order and thus may fail to preserve the host programming language's semantics.

Recently, in order to solve the aforementioned problems with the current language-integrated approaches, and more generally, to investigate to what extent one can push the idea of RDBMSs that directly and seamlessly participate in program evaluation, the Ferry language has been proposed [12]. Ferry is a functional programming language that is designed to be entirely executed on RDBMSs. So far, the most notable feature of Ferry has been its compilation technique that supports database execution of programs of nested types, maintains list order and provides avalanche-safety guarantees [13].

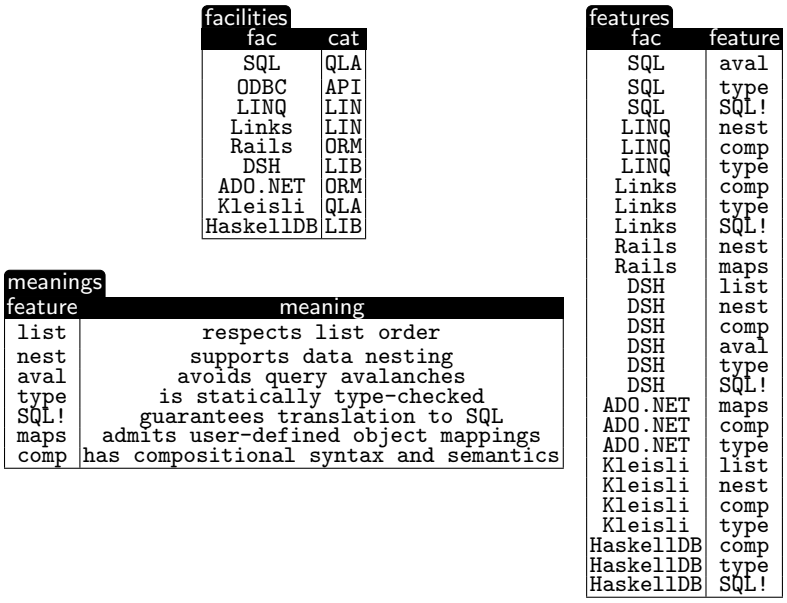
Although the Haskell programming language [23] has inspired a number of language-integrated query facilities (most notably LINQ which is based on monad comprehensions) so far no such system has been proposed or implemented for Haskell. With *Database-Supported Haskell* (DSH) we provide a library that executes parts of a Haskell program on an RDBMS. The library is available online [2]. The design and implementation of the library is influenced by Ferry and it can be considered as a Haskell-embedded implementation of Ferry.

The library is based on Haskell's list comprehensions and the underlying list-processing combinators and provides a convenient query integration into the host language. The library, just like the Ferry language, in addition to queries of basic types, supports computations over arbitrarily nested tuples and lists. The implementation minimises unnecessary data transfer and context switching between the database *coprocessor* and the programming language runtime. Specifically, in DSH, the number of queries is only dependent on the number of list type constructors in the result type of the computation and does not depend on the size of the queried data.

Our contribution with this paper is the first proposal and implementation of a library for database-supported program execution in Haskell.

2 DSH by Example

Consider the database table facilities in Figure 1 which lists a sample of contemporary facilities (query languages, APIs, *etc.*) that are used to query database-resident data. We have attempted to categorise these facilities (see column cat of table facilities): query language (`QLA`), library (`LIB`), application programming interface (`API`), host language integration (`LIN`), and object-relational mapping (`ORM`). Furthermore, each of these facilities has particular features (table features). A verbose description of these features is given by the table meanings.



facilities

fac	cat
SQL	QLA
ODBC	API
LINQ	LIN
Links	LIN
Rails	ORM
DSH	LIB
ADO.NET	ORM
Kleisli	QLA
HaskellDB	LIB

features

fac	feature
SQL	aval
SQL	type
SQL	SQL!
LINQ	nest
LINQ	comp
LINQ	type
Links	comp
Links	type
Links	SQL!
Rails	nest
Rails	maps
DSH	list
DSH	nest
DSH	comp
DSH	aval
DSH	type
DSH	SQL!
ADO.NET	maps
ADO.NET	comp
ADO.NET	type
Kleisli	list
Kleisli	nest
Kleisli	comp
Kleisli	type
HaskellDB	comp
HaskellDB	type
HaskellDB	SQL!

meanings

feature	meaning
list	respects list order
nest	supports data nesting
aval	avoids query avalanches
type	is statically type-checked
SQL!	guarantees translation to SQL
maps	admits user-defined object mappings
comp	has compositional syntax and semantics

Fig. 1. Database-resident input tables for example program.

Given this base data, an interesting question would be: What features are characteristic for the various query facility categories (column cat) introduced above? Interpreting a table as a list of tuples, this question can be answered with the following list-based Haskell program:

$$
\begin{aligned}
&\mathit{descrFacility} :: \mathit{String} \to [\mathit{String}] \\
&\mathit{descrFacility}\ f = [\mathit{mean} \mid (\mathit{feat}, \mathit{mean}) \leftarrow \mathit{meanings}, \\
&\qquad\qquad\qquad\qquad\quad (\mathit{fac}, \mathit{feat}') \quad \leftarrow \mathit{features}, \\
&\qquad\qquad\qquad\qquad\quad \mathit{feat} \equiv \mathit{feat}' \wedge \mathit{fac} \equiv f\,] \\
&\mathit{query} :: [(\mathit{String}, [\mathit{String}])] \\
&\mathit{query} = [\ (\mathit{the}\ \mathit{cat}, \mathit{nub}\ (\mathit{concatMap}\ \mathit{descrFacility}\ \mathit{fac})) \\
&\qquad\quad \mid (\mathit{fac}, \mathit{cat}) \leftarrow \mathit{facilities}, \mathbf{then}\ \mathit{group\ by}\ \mathit{cat}]
\end{aligned}
$$

The program consists of the actual query and the helper function *descrFacility*. Function *descrFacility*, given a query facility f, returns a list of descriptions of its features. We deliberately use a combination of list comprehension notation and list-processing combinators in this example program to demonstrate the breadth of DSH. Evaluating this program results in a nested list like:

```
[("API", []),
 ("LIB", ["respects list order", ...]),
 ("LIN", ["supports data nesting", ...]),
 ("ORM", ["supports data nesting", ...]),
 ("QLA", ["avoids query avalanches", ...])]
```

As the example program processes database-resident data, it would be most efficient to perform the computation close to the data and let the database query engine itself execute the program. With DSH, this is exactly what we propose and provide. The program is translated into two SQL queries (see Appendix). These queries fully represent the program and can completely be executed on the database.

In order to execute the program on the database we have to apply a few modest changes to the example program. These changes turn the program into a DSH program. We will discuss these changes in the remainder of this section. The adapted program will be our running example in the remainder of this paper.

First, consider the function *descrFacility*. There are three modest changes to this function related to its type signature, the comprehension notation and a new combinator named *table*. The new function looks as follows:

$$
\begin{aligned}
&\mathit{descrFacility} :: Q\ \mathit{String} \to Q\ [\mathit{String}]\\
&\mathit{descrFacility}\ f = [\, qc \mid \mathit{mean} \mid (\mathit{feat}, \mathit{mean}) \leftarrow \mathit{table}\ \texttt{"meanings"},\\
&\qquad\qquad\qquad\qquad\qquad\quad (\mathit{fac}, \mathit{feat}') \ \ \leftarrow \mathit{table}\ \texttt{"features"},\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{feat} \equiv \mathit{feat}' \wedge \mathit{fac} \equiv f\ \mid]
\end{aligned}
$$

The slight change of the list comprehension syntax is due to *quasi-quotes* [19], namely $[\, qc \mid$ and $\mid]$. Otherwise, the syntax and semantics of quasiquoted comprehensions match those of regular Haskell list comprehensions with the only exception that, instead of having type $[a]$, a quasiquoted comprehension has type $Q\ [a]$ (to be read as "*a query that returns a value of type* $[a]$"). This explains the change in the type signature of function *descrFacility*. The last change that has to be made is to direct the program to use database-resident data instead of heap data. This is achieved by using the *table* combinator that introduces the name of the queried table.

Finally let us consider the main function, *query*. With DSH, we support most of the Haskell list prelude functions, modified to work with queries that return lists. Once the changes are applied, the code looks as follows:

$$
\begin{aligned}
&\mathit{query} :: \mathit{IO}\ [(\mathit{String}, [\mathit{String}])]\\
&\mathit{query} = \mathit{fromQ}\ \mathit{connection}\\
&\quad [\, qc \mid (\mathit{the}\ \mathit{cat}, \mathit{nub}\ (\mathit{concatMap}\ \mathit{descrFacility}\ \mathit{fac}))\\
&\qquad \mid (\mathit{cat}, \mathit{fac}) \leftarrow \mathit{table}\ \texttt{"facilities"}, \mathbf{then}\ \mathit{group\ by}\ \mathit{cat}\ \mid]
\end{aligned}
$$

The list comprehension in this function is adapted as in the *descrFacility* function. Function *fromQ*, when provided with a connection parameter, executes its query argument on the database and returns the result as a regular Haskell value. This value is wrapped inside the *IO* monad as database updates may alter the queried data between two invocations of *fromQ* (i.e., this is not a referentially transparent computation).

The following section describes how DSH programs are compiled and executed by effectively using RDBMSs as coprocessors supporting the Haskell runtime.

3 Internals

The execution model of DSH is presented in Figure 2. By using the quasi-quoter that implements the well-known desugaring approach [16], list comprehensions are translated into list-processing combinators at Haskell compile-time (①, Figure 2). With a translation technique coined *loop-lifting* [13], these list-processing combinators are compiled into an intermediate representation called *table algebra*, a simple variant of relational algebra (②, Figure 2). Through Pathfinder [10, 11], a table algebra optimiser and code generation facility, the intermediate representation is optimised and compiled into relational queries (③, Figure 2). Pathfinder supports a number of relational back-end languages (e.g., SQL:1999 and the MonetDB Interpreter Language (MIL) [5]). The resulting relational queries can then be executed on off-the-shelf relational database systems (④, Figure 2). The tabular query results are transferred back into the heap and then transformed into vanilla Haskell values (⑤ and ⑥, Figure 2). In the remainder of this section we describe the aforementioned steps in further detail.

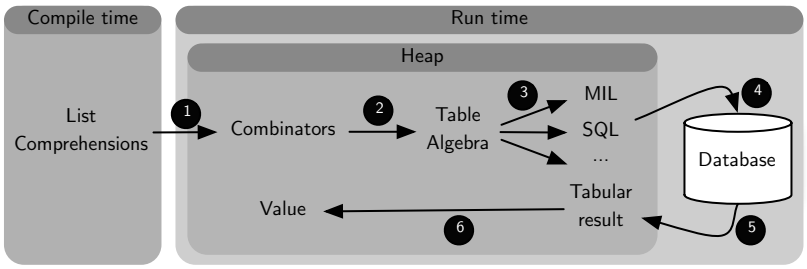


Fig. 2. Code Motion in DSH

3.1 Haskell Front-End

The following provides an incomplete lineup of the supported Haskell list combinators along with their types:

$$
\begin{array}{lll}
\mathit{map} & :: (\mathit{QA}\ a, \mathit{QA}\ b) & \Rightarrow (Q\ a \to Q\ b) \to Q\ [a] \to Q\ [b] \\
\mathit{filter} & :: (\mathit{QA}\ a) & \Rightarrow (Q\ a \to Q\ \mathit{Bool}) \to Q\ [a] \to Q\ [a] \\
\mathit{concat} & :: (\mathit{QA}\ a) & \Rightarrow Q\ [[a]] \to Q\ [a] \\
\mathit{groupWith} & :: (\mathit{Ord}\ b, \mathit{QA}\ a, \mathit{QA}\ b) & \Rightarrow (Q\ a \to Q\ b) \to Q\ [a] \to Q\ [[a]] \\
\mathit{sortWith} & :: (\mathit{Ord}\ b, \mathit{QA}\ a, \mathit{QA}\ b) & \Rightarrow (Q\ a \to Q\ b) \to Q\ [a] \to Q\ [a] \\
\mathit{the} & :: (\mathit{Eq}\ a, \mathit{QA}\ a) & \Rightarrow Q\ [a] \to Q\ a \\
\mathit{nub} & :: (\mathit{Eq}\ a, \mathit{QA}\ a) & \Rightarrow Q\ [a] \to Q\ [a] \\
\mathit{table} & :: (\mathit{TA}\ a) & \Rightarrow \mathit{String} \to Q\ [a]
\end{array}
$$

These combinators behave as their namesakes in the Haskell list prelude, but instead of operating on regular lists and values they work on *queryable lists and values*. This is reflected in their type signatures. It is easy to arrive at the type signature of a DSH combinator starting from the type signature of its namesake in the Haskell list prelude: (1) the Q type constructor needs to be applied to all types except function types, and (2) the QA type class constraint (read: *queryable*) needs to be applied to all type variables in the signature.

In order to restrict the combinators to only work with the data types that we are able to represent relationally, the QA type class is used:

$$\begin{aligned}&\textbf{class } QA\ a \textbf{ where}\\ &\quad toQ \quad :: a \to Q\ a\\ &\quad fromQ :: Connection \to Q\ a \to IO\ a\end{aligned}$$

DSH provides QA instances for the basic types (i.e., Boolean, character, integer, real, text, date and time types), as well as arbitrarily nested lists and tuples of these basic types. In addition, by leveraging metaprogramming capabilities of Template Haskell [25], we provide for automatic derivation of QA instances for *any* user-defined product type (including Haskell records) and automatic generation of Haskell records from database schemas. The relational representation of the supported Haskell types is given in Section 3.2.

The current implementation of DSH does not support general folds (e.g., *foldr* and *foldl*). All other functions from the list prelude are supported, including the special folds. The compilation of general folds and user-defined recursive definitions would yield relational queries that build on recursion constructs of the underlying query language (e.g., recursive queries using common table expressions in SQL:1999). This is something that we are currently investigating.

DSH also lacks sum types. However, this is easier to address than the missing support for general folds and recursion. In fact, in related work (which remains to be published), we have already devised a relational representation for sum types and compilation rules for functions on sum types. The aforementioned related work addresses the shortcomings of Links that are outlined in Section 1 by leveraging and extending the compilation technology developed for the Ferry language.

The QA type class provides functions to convert Haskell values into queries (i.e, the toQ function) and vice versa ($fromQ$). The latter function triggers the compilation of queries, communicates with the database, submits the generated relational queries to the database, fetches the results and converts them into Haskell values. These values can be used in the program for further in-heap processing as well as for the generation of new database-executable program fragments. Thus, DSH supports iterative staging of embedded programs [9, 20] by allowing for runtime code generation, compilation, and execution of database-executable programs.

The combinator *table* deserves special attention. As can be seen in the lineup of combinators, its type features a type class constraint $TA\ a$. The constraint restricts the type variable a, which represents the table rows, to the basic types

and flat tuples of the basic types. Use of the *table* combinator does not result in I/O, as it does not initiate communication with the database: it just references the database-resident table by its unique name. In the case that the table has multiple columns, these columns are gathered in a flat tuple whose components are ordered alphabetically by column name. With the current DSH implementation, it is the user's responsibility to make sure that the referenced table does exist in the database and that type a indeed matches the table's row type—otherwise, an error is thrown at runtime.

The DSH combinators, in the tradition of deep embeddings, construct an internal data representation of the embedded program fragment they represent. The subsequent optimisation and compilation into relational queries is based on this representation. The following presents the data type that is used to represent the DSH combinators internally:

```
data Exp = BoolE    Bool          Type
         | IntegerE Integer       Type   -- ... and other basic types
         | TupleE   [Exp]         Type
         | ListE    [Exp]         Type
         | VarE     String        Type
         | TableE   String        Type
         | LamE     String Exp    Type
         | AppE     Exp    Exp    Type
```

The type annotations are used to map the DSH-supported data types to their relational encodings (see Section 3.2). The *Exp* data type features type annotations at the value-level, represented by elements of the algebraic datatype *Type*. As a direct result of this, the internal representation is not guaranteed to represent a type-correct expression. However, this data type is not exposed to the user of the library and extra care has been taken to make sure that the combinators map to a consistent underlying representation.

The DSH combinators are typed using a technique called *phantom typing* [17, 24]. In particular, the Q data type is defined as **data** $Q\ a = Q\ Exp$ featuring the type variable a that does not occur in the type definition. Instead, as shown above, this type variable is used to give the required Haskell types to the DSH combinators. Thus, we obviate the need for type-checking the internal representation by delegating this task to the host language type-checker.

The abstract data type $Q\ a$ and its representation is not exposed to the user. As a consequence, direct pattern matching on values of type $Q\ a$ is not possible. However, we do provide a limited form of pattern matching on queries of product types using *view patterns*, a syntactic extension of the Glasgow Haskell Compiler (GHC) [3]. To illustrate, the following expression that contains a view pattern $\lambda(view \to pat) \to expr$ is desugared to $\lambda x \to$ **case** $view\ x$ **of** $pat \to expr$. A view pattern can match on values of an abstract type if it is given a *view* function that maps between the abstract type and a matchable data type. In order to support multiple types with the same *view* function, we introduce a type class *View* as follows:

$$\textbf{class } \mathit{View}\ a\ b \mid a \to b \textbf{ where}$$
$$\quad \mathit{view} :: a \to b$$

Instead of projecting all data types onto the same matchable type, we use a type variable b that is uniquely determined by the type a. The signature of the *View* instance for pairs, for example, reads:

$$\textbf{instance } (\mathit{QA}\ a, \mathit{QA}\ b) \Rightarrow \mathit{View}\ (Q\ (a, b))\ (Q\ a, Q\ b)$$

DSH provides for automatic derivation of *View* instances for *any* user-defined product type.

Regular Haskell list comprehensions only support generators, guards and local bindings [23, Section 3.11]. List comprehensions extended with SQL-inspired *order by* and *group by* constructs were added as a language extension to improve the expressiveness of the list comprehensions notation [16]. Further, parallel list comprehensions are supported by GHC as an extension. DSH supports standard list comprehensions as well as both extensions through quasi-quoting machinery [19, 25]. This results only in a slight syntactic overhead. A DSH list comprehension is thus written as:

$$[\,\mathit{qc} \mid \mathit{expr} \mid \mathit{quals} \mid]$$

where *qc* (read: *query comprehension*) is a quasi-quoter that desugars list comprehensions into DSH list-processing combinators.

3.2 Turning Haskell into SQL

After building the type-annotated abstract syntax tree (AST) of a DSH program, we use a syntax-directed and compositional translation technique called loop-lifting to compile this AST into table algebra plan(s) [13]. This intermediate representation has been designed to reflect the query capabilities of modern off-the-shelf relational database engines.

Loop-lifting implements DSH computations over arbitrarily nested data using a flat data-parallel evaluation strategy (see Section 4.2) executable by any relational database system. Specific database engines are targeted by code generators that derive tailored back-end queries from the generic table algebra plans. A SQL:1999 code generator allows us to target any standards-compliant RDBMS [11], a MIL back-end enables DSH to target the MonetDB database system [5] .

All data types and operations supported by DSH are turned into relational representations that *faithfully* preserve the DSH semantics on a relational back-end. We will discuss this relational encoding in the following.

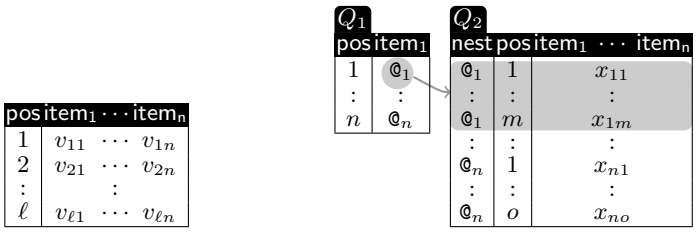
Atomic values and (nested) tuples. DSH values of atomic types are directly mapped into values of a corresponding table column type. An n-tuple $(v_1, ..., v_n)$, $n \geqslant 1$, of such values maps into a table row of width n. A singleton tuple (v) and value v are treated alike. A nested tuple $((v_1, ..., v_n), ..., (v_{n+k}, ..., v_m))$ is represented like its flat variant $(v_1, ..., v_n, ..., v_{n+k}, ..., v_m)$.

Ordered lists. Relational back-ends normally cannot provide ordering guarantees for rows of a table. We therefore let the compiler create a *runtime-accessible encoding of row order*. A list value $[x_1, x_2, ..., x_l]$—where x_i denotes the n-tuple $(v_{i1}, ..., v_{in})$—is mapped into a table of width $1 + n$ as shown in Figure 3(a). Again, a singleton list $[x]$ and its element x are represented alike. A dedicated column pos is used to encode the order of a list's elements.

Nested lists. Relational tables are flat data structures and special consideration is given to the representation of nested lists. If a DSH program produces the list $[[x_{11}, x_{12}, ..., x_{1m}], ..., [x_{n1}, x_{n2}, ..., x_{no}]]$ (with $m, n, o \geqslant 0$) which exhibits a nesting depth of two, the compiler will translate the program into a *bundle* of two separate relational queries, say Q_1 and Q_2. Figure 3(b) shows the resulting tabular encodings produced by the relational query bundle:

Q_1, a query that computes the relational encoding of the outer list $[\mathtt{@}_1, \ldots, \mathtt{@}_n]$ in which all inner lists (including empty lists) are represented by *surrogate keys* $\mathtt{@}_i$, and

Q_2, a query that produces the encodings of *all* inner lists—assembled into a single table. If the ith inner list is empty, its surrogate $\mathtt{@}_i$ will not appear in the nest column of this second table.



(a) Encoding a flat ordered list.

(b) Encoding a nested list (Q_1: outer list, Q_2: all inner lists).

Fig. 3. Relational encoding of order and nesting on the database backend.

This use of the surrogates $\mathtt{@}_i$ resembles van den Bussche's simulation of the nested algebra via the flat relational algebra [27]. In effect, our compiler uses a non-parametric representation for list elements [15] in which the element *types* determine their efficient relational representation: either in-line (for tuples of atomic items) or surrogate-based (for lists)—we come back to this in Section 4.2. In [13], we describe a compile-time analysis phase—coined *(un)boxing*—that infers the correct non-parametric representation for all subexpressions of a DSH program.

The most important consequence of this design is that it is exclusively the *number of list constructors* $[\cdot]$ *in the program's result type* that determines the

number of queries contained in the emitted relational query bundle. We refer to this crucial property as *avalanche safety*.

For the example program from Section 2 with type $[(String, [String])]$, the bundle size thus is two. This is radically different from related approaches like LINQ [21] or HaskellDB [17] which may yield sequences of SQL statements whose size is dependent on the *size of the queried database instance* (see Section 4.1). The two SQL queries that are generated by the SQL:1999 code generator for the example program are given in the Appendix.

Operations. Relational query processors are specialists in *bulk-oriented evaluation*: in this mode of evaluation, the system applies an operation to *all* rows in a given table. In absence of inter-row dependencies, the system may process the individual rows in any order or even in parallel. To actually operate the database back-end in this data-parallel fashion, our compiler draws the necessary amount of independent work from DSH's list combinators (see Section 3.1). Consider this application of *map*,

$$map\ (\lambda x \to e)\ [v_1, ..., v_n] = [\, e\,[v_1/x]\,, ..., e\,[v_n/x]\,]$$

which performs n *independent* evaluations of expression e under different bindings of x ($e\,[v/x]$ denotes the consistent replacement of free occurrences of x in e by v). Loop lifting exploits this semantics and compiles *map* into an algebraic plan that evaluates $e\,[v_i/x]$ $(i = 1, ..., n)$ in a data-parallel fashion: all n bindings for x are supplied in a single table and the database system is free to consider these bindings and the corresponding evaluations of e in any order it sees fit (or in parallel). Loop-lifting thus fully realises the independence of the iterated evaluations and enables the relational query engine to take advantage of its bulk-oriented processing paradigm. A detailed account of the loop-lifting of operations is given in [13].

4 Related Work

Embedding database query capabilities into a functional programming language is not a new idea. The most notable examples include Kleisli [18], Links [8], LINQ [21, 26] and HaskellDB [17].

Kleisli [18] is a data integration system mainly used to query bioinformatics data. The system features the Collection Programming Language (CPL) for specifying queries. This language supports a rich data model with arbitrarily nested sets, bags, lists, tuples, records and variants. CPL queries can be formulated using comprehension notation for the supported collection types. However, the Kleisli system does not provide for avalanche safety, the feature that DSH inherits from the Ferry language. The Kleisli system supports querying of disparate data that resides in different databases—this is something we have not yet considered in the design and implementation of DSH.

Links [8] is a web programming language that provides for tier-less web development. That is, a Links program is compiled into client-side, server-side

and database-executable fragments. The Links system only supports database execution of program fragments that deal with flat data.

LINQ seamlessly embeds a declarative query language facility into Microsoft's .NET language framework [21, 26]. Similar to DSH, a LINQ query against database-resident relational tables is compiled into a sequence of SQL statements, but without DSH's avalanche safety guarantee. Also, LINQ does not provide any relational encoding of order. As a consequence, in LINQ, particular order-sensitive operations are either flagged as being unsupported or are mapped into database queries that return list elements in some arbitrary order [13].

HaskellDB [17] is a combinator library for Haskell that enables the construction of SQL queries in a type-safe and declarative fashion. As HaskellDB is a well-known system in the Haskell community that is related to DSH's approach of letting developers use the Haskell programming language itself to formulate (type-safe) queries against database-resident data, we will compare DSH to HaskellDB in more detail in the following.

4.1 HaskellDB

As with DSH, a HaskellDB [17] query is formulated completely within Haskell without having to resort to SQL syntax. Let us highlight two significant differences between HaskellDB and DSH.

Query Avalanches DSH provides a guarantee that the *number of SQL queries issued* to implement a given program is exclusively *determined by the program's static type*: each occurrence of a list type constructor $[t]$ accounts for exactly one SQL query. Our running example from Section 2 with its result type of shape $[[\cdot]]$ thus led to the bundle of two SQL queries shown in Appendix. This marks a significant deviation from HaskellDB: in this system, the length of the SQL statement sequence may depend on the *database instance size*, resulting in an avalanche of queries.

To make this concrete, we reformulated the example program in HaskellDB (Figure 4). Table 1 shows the number of SQL queries emitted by HaskellDB and DSH in dependence of the number of distinct categories in column cat of table facilities. Table 1 also gives the overall runtimes of both programs for the different category counts. To measure these times we ran both programs on the same 2.8 GHz Intel Core 2 Duo computer (running Mac OS X 10.6.6) with 8 GB of RAM. PostgreSQL 9.0.2-1 was used as the database back-end. We executed each program ten times. Table 1 lists the average runtimes for the two programs along with upper and lower bounds with 95% confidence interval, as calculated by the criterion library [1].

For HaskellDB, the query avalanche effect is clearly visible: the number of queries generated depends on the population of column cat and the relational database back-end is easily overwhelmed by the resulting query workload. With DSH, the number of queries issued remains constant significantly reducing the overall runtime.

List Order Preservation List element order is inherent to the Haskell data model. DSH relationally encodes list order (column pos) and carefully trans-

```
getCats :: Query (Rel (RecCons Cat (Expr String) RecNil))
getCats = do
            facs ← table facilities
            cats ← project (cat << facs ! cat)
            unique
            return cats

getCatFeatures :: String → Query (Rel (RecCons Meaning (Expr String) RecNil))
getCatFeatures cat = do
                       facs    ← table facilities
                       feats   ← table features
                       means ← table meanings
                       restrict $ feats ! feature . == . means ! feature .&&.
                                  facs ! cat      . == . constant cat .&&.
                                  facs ! fac      . == . feats ! fac
                       m ← project (meaning << means ! meaning)
                       unique
                       return m

query :: IO [(Record (RecCons Cat String RecNil)
             ,[Record (RecCons Meaning String RecNil)])]
query = do
          cs ← doQuery getCats
          sequence $ map (λc → do
                               m ← doQuery $ getCatFeatures $ c ! cat
                               return (c, m)) cs
```

Fig. 4. HaskellDB version of running example.

lates operations such that this order encoding is preserved (see Section 3.2). In contrast, HaskellDB does not provide any relational encoding of order. As a consequence, HaskellDB does not support order-sensitive operations.

4.2 Data Parallel Haskell

RDBMSs are carefully tuned and highly efficient table processors. A look under the hood reveals that, indeed, database query engines provide a sophisticated *flat data-parallel* execution environment: most of the engine's primitives—typically a variant of the relational algebra—apply a single operation to all rows of an input table (consider relational selection, projection, or join, for example).

In this sense, DSH is a close relative of Data Parallel Haskell (DPH) [6, 7, 15]: both accept very similar comprehension-centric Haskell fragments, both yield code that is amenable for execution on flat data-parallel backends (relational query engines or contemporary multi-core CPUs). We shed light on a few striking similarities here.

Parallel arrays *vs.* tables DPH programs operate over so-called parallel arrays of type $[:a:]$, the primary abstraction of a vector of values that is subject

Table 1. Number of SQL queries emitted and observed overall program execution times in dependence of the population of column `cat` for the HaskellDB and DSH implementations of the running example (DNF: did not finish within hours).

	HaskellDB		**DSH**	
# categories	# queries	⏱ (sec)	# queries	⏱ (sec)
1 000	1 001	$11.712^{+0.2\%}_{-0.2\%}$	2	$0.604^{+1.1\%}_{-0.3\%}$
10 000	10 001	$291.369^{+3.2\%}_{-2.4\%}$	2	$6.419^{+1.5\%}_{-2.0\%}$
100 000	100 001	DNF	2	$74.709^{+0.5\%}_{-0.3\%}$

$$
\begin{aligned}
&\textbf{type } \mathit{Vector} = [:\mathit{Float}:]\\
&\textbf{type } \mathit{SparseVector} = [:(\mathit{Int}, \mathit{Float}):]\\
&\mathit{sumP} :: \mathit{Num}\ a \Rightarrow [:a:] \to a\\
&(!:) \quad :: [:a:] \to \mathit{Int} \to a\\
&\mathit{dotp} :: \mathit{SparseVector} \to \mathit{Vector} \to \mathit{Float}\\
&\mathit{dotp}\ \mathit{sv}\ v = \mathit{sumP}[:x * (v\ !:\ i) \mid (i, x) \leftarrow \mathit{sv}:]
\end{aligned}
$$

Fig. 5. DPH example: sparse vector multiplication (taken from [7]).

to bulk computation. Positional indexing into these arrays (via !:) is ubiquitous. Parallel arrays are strict: evaluating one array element evaluates the entire array.

In comparison, the DSH-generated database primitives operate over (unordered) tables in which a dedicated column `pos` explicitly encodes element indexes (see Section 3.2). Primitive operations are always applied to all rows of their input tables.

Non-parametric data representation In DPH, arrays of tuples $[:(a, b):]$ are represented as tuples of arrays $([:a:], [:b:])$ of identical length. The representation of a nested array $[:[:a:]:]$ has two components: (1) an array of $(\mathit{offset}, \mathit{length})$ descriptors, and (2) a flat data array $[:a:]$ holding the actual elements.

In DSH, the fields of a tuple live in adjacent columns of the same table. A nested list is represented in terms of two tables: (1) a table of surrogate keys, each of which identifies a nested list, and (2) a table of data elements, each accompanied by a foreign surrogate key to encode list membership.

This foreign-key-based representation of nesting can readily benefit from relational indexes that map any data element x to the surrogate key of its containing list. A direct adoption of DPH's $(\mathit{offset}, \mathit{length})$ descriptor-based representation, instead, would ultimately lead to range queries of the form $x.\mathsf{pos}$ **BETWEEN** *offset* **AND** $\mathit{offset} + \mathit{length}$— a workable but less efficient alternative on off-the-shelf relational database back-ends.

Note that it is crucial, though, that DPH's as well as DSH's representation preserve the locality of the actual elements held in nested data structures.

Lifting operations In DPH, operations of type $a \to b$ are lifted to apply to entire arrays of values: $[:a:] \to [:b:]$. Consider a DPH variant of sparse vector multiplication (Figure 5). The comprehension defines an iterative computation to be applied to each element of sparse vector sv: project onto the components i and x, perform positional array access into v ($!:$), multiply. With *vectorisation*, Data Parallel Haskell trades comprehension notation for data-parallel combinators (*e.g.*, $fst\hat{}$, $*\hat{}$, $bpermuteP$), all operating over entire arrays (Figure 6, left).

DSH's translation strategy, loop-lifting, compiles a family of Haskell list combinators into algebraic primitives, all of which operate over entire tables (*i.e.*, the relational engine performs lifted application by definition) [13].

Intermediate code produced by DPH and DSH indeed exhibits structural similarities. To illustrate, we have identified the database primitives that implement the sparse vector multiplication program of Figure 5 in Figure 6 (right): $bpermuteP$, which performs bulk indexed array lookup, turns into a relational equi-join over column pos, for example.

A study of the exact relationship between DPH and DSH still lies ahead. We conjecture that DSH's loop-lifting compilation strategy does have an equivalent formulation in terms of vectorisation or Blelloch's flattening transformation [4].

$$
\begin{aligned}
&fst\hat{} && :: [:(a,b):] \to [:a:]\\
&snd\hat{} && :: [:(a,b):] \to [:b:]\\
&bpermuteP && :: [:a:] \to [:Int:] \to [:a:]\\
&\textbf{let } sv && = [:(1,0.1),(3,1.0),(4,0.0):]\\
&\quad v && = [:10,20,30,40,50:]\\
&\textbf{in}\\
&\quad sumP\ (snd\hat{}\ sv *\hat{}\ bpermuteP\ v\ (fst\hat{}\ sv))
\end{aligned}
$$

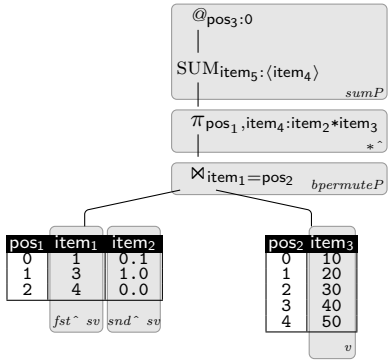


Fig. 6. Intermediate code generated for the sparse vector multiplication example of Fig. 5: DPH (left) *vs.* DSH (right).

4.3 Embedding Approaches

The DSH implementation is realised through a combination of a number of established language embedding techniques. In the tradition of deep embeddings [14] DSH's list processing combinators construct data representation of embedded queries allowing for domain-specific optimisation and code generation. Haskell's

template meta-programming and quasiquoting facilities are used to provide convenient syntactic sugar (i.e., the list comprehension notation with extensions) for the list-processing combinators. Type correctness of the embedded queries is ensured by phantom typing [17, 24]. View patterns [28] provide for pattern matching on otherwise abstract data representation of embedded queries.

5 Future Work and Conclusions

In this paper we presented Database Supported Haskell (DSH), a library that allows the use of a relational database management system (RDBMS) as a co-processor for the Haskell runtime. Thus, DSH is capable of processing large scale data, that would otherwise exceed heap capacity. Database-executable program fragments are written in a style that has been designed to imitate the syntax, types and semantics of typical Haskell functions over lists. The expressive list comprehension notation is supported with only slight syntactic overhead (due to our use of quasi-quoting).

Overall, as a host language, Haskell served our needs well in the process of DSH design and implementation. However, there is room for improvements. In particular, if the comprehension notation would have been supported not only for lists but for any monad [29], as was the case in Haskell 1.4 [22], we could define DSH combinators as monadic combinators and reuse monad comprehension notation instead of implementing our own quasi-quoter. This would save the implementation effort, eliminate the syntactic overhead and lead to better error messages referring to the original source code instead of the generated code.

We think that monadic **do** notation is a poor fit for DSH and, more generally, for list-based libraries and Domain Specific Languages (DSLs). Thus, we are bringing back monad comprehensions to Haskell, and are currently working on an implementation as an extension for the Glasgow Haskell Compiler [3]. We are also generalising the recently proposed extensions of the list comprehension notation [16] to monads so that those extensions could also be used in the monad comprehension notation. Although we think that list-based libraries and DSLs would benefit most, this extension could have other interesting applications.

DSH uses compilation technology developed for the Ferry language. Thus, we support nested lists, preserve list order and guarantee that the number of generated relational queries only depends on the static type of the query result and remains unaffected by the database instance size. We expect to continue work on DSH and Ferry in the following directions:

- support for functions as first-class citizens (so that the result of a sub-query can also be a function),
- support for sum types,
- support for general folds and recursion,
- the application of DSH to large-scale data analysis problems, and
- an exploration of DSH's relationship to Data Parallel Haskell.

Acknowledgements. Tom and Jeroen have been supported by the German Research Foundation (DFG), Grant GR 2036/3-1. George has received support from the ScienceCampus Tübingen: Informational Environments. We would also like to acknowledge Alexander Ulrich, Jan Rittinger and the anonymous reviewers for their helpful comments and feedback.

Bibliography

[1] Criterion. `http://hackage.haskell.org/package/criterion`.
[2] Database Supported Haskell. `http://hackage.haskell.org/package/DSH`.
[3] The Glasgow Haskell Compiler. `http://www.haskell.org/ghc/`.
[4] Guy E. Blelloch and Gary W. Sabot. Compiling Collection-Oriented Languages onto Massively Parallel Computers. *Journal of Parallel and Distributed Computing*, 8:119–134, February 1990.
[5] Peter A. Boncz and Martin L. Kersten. MIL primitives for querying a fragmented world. *The VLDB Journal*, 8:101–119, October 1999.
[6] Manuel M. T. Chakravarty and Gabriele Keller. More Types for Nested Data Parallel Programming. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 94–105, Montreal, Canada, 2000. ACM.
[7] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data Parallel Haskell: a status report. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming (DAMP)*, pages 10–18, Nice, France, 2007. ACM.
[8] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *Proceedings of the 5th International Symposium on Formal Methods for Components and Objects (FMCO)*, pages 266–296, Amsterdam, The Netherlands, 2006. Springer-Verlag.
[9] George Giorgidze and Henrik Nilsson. Mixed-level Embedding and JIT Compilation for an Iteratively Staged DSL. In *Proceedings of the 19th International Workshop on Functional and (Constraint) Logic Programming (WFLP)*, Madrid, Spain, Jan 2010. Springer-Verlag.
[10] Torsten Grust, Manuel Mayr, and Jan Rittinger. Let SQL Drive the XQuery Workhorse (XQuery Join Graph Isolation). In *Proceedings of the 13th International Conference on Extending Database Technology (EDBT)*, pages 147–158, Lausanne, Switzerland, 2010. ACM.
[11] Torsten Grust, Manuel Mayr, Jan Rittinger, Sherif Sakr, and Jens Teubner. A SQL: 1999 Code Generator for the Pathfinder XQuery Compiler. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1162–1164, Beijing, China, 2007. ACM.
[12] Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber. FERRY: Database-Supported Program Execution. In *Proceedings of the 35th SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1063–1066, Providence, RI, USA, 2009. ACM.
[13] Torsten Grust, Jan Rittinger, and Tom Schreiber. Avalanche-Safe LINQ Compilation. In *Proceedings of the 36th International Conference on Very Large Databases (VLDB)*, pages 162–172, Singapore, Sep 2010. VLDB Endowment.
[14] Paul Hudak. Modular Domain Specific Languages and Tools. In *Proceedings of Fifth International Conference on Software Reuse (ICSR)*, pages 134–142, June 1998.

[15] Simon Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. Harnessing the Multicores: Nested Data Parallelism in Haskell. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 2, pages 383–414, Bangalore, India, 2008. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[16] Simon Peyton Jones and Philip Wadler. Comprehensive Comprehensions. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 61–72, Freiburg, Germany, 2007. ACM.

[17] Daan Leijen and Erik Meijer. Domain Specific Embedded Compilers. In *Proceedings of the 2nd Conference on Domain-Specific Languages (DSL)*, pages 109–122, Austin, Texas, United States, 1999. ACM.

[18] Wong Limsoon. Kleisli, a functional query system. *Journal of Functional Programming*, 10:19–56, January 2000.

[19] Geoffrey Mainland. Why It's Nice to be Quoted: Quasiquoting for Haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 73–82, Freiburg, Germany, 2007. ACM.

[20] Geoffrey Mainland and Greg Morrisett. Nikola: Embedding Compiled GPU Functions in Haskell. In *Proceedings of the Third ACM Haskell Symposium on Haskell*, pages 67–78, Baltimore, Maryland, USA, 2010. ACM.

[21] Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: Reconciling Objects, Relations and XML in the .NET Framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 706–706, Chicago, IL, USA, 2006. ACM.

[22] John Peterson and Kevin Hammond. Haskell 1.4: A Non-strict, Purely Functional Language. Technical Report YALEU/DCS/RR-1106, Department of Computer Science, Yale University, 1997.

[23] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003.

[24] Morten Rhiger. A Foundation for Embedded Languages. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25:291–315, May 2003.

[25] Tim Sheard and Simon Peyton Jones. Template Meta-programming for Haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 1–16, Pittsburgh, PA, USA, October 2002. ACM.

[26] Don Syme. Leveraging .NET Meta-programming Components from F#: Integrated Queries and Interoperable Heterogeneous Execution. In *Proceedings of the 2006 Workshop on ML*, pages 43–54, Portland, Oregon, USA, 2006. ACM.

[27] Jan Van den Bussche. Simulation of the nested relational algebra by the flat relational algebra, with an application to the complexity of evaluating powerset algebra expressions. *Theoretical Computer Science*, 254:363–377, March 2001.

[28] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 307–313, Munich, West Germany, 1987. ACM.

[29] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming (LFP)*, pages 61–78, Nice, France, 1990. ACM.

Appendix

DSH's SQL:1999 code generator emits the following bundle of two SQL queries for the Haskell example program of Section 2:

```
WITH
  -- binding due to duplicate elimination
t0000 (item1_str) AS
  (SELECT DISTINCT a0000.categorie AS item1_str
    FROM facilities AS a0000)
SELECT
  DENSE_RANK () OVER
    (ORDER BY a0001.item1_str ASC) AS item4_nat,
  1 AS iter3_nat, a0001.item1_str
FROM t0000 AS a0001
ORDER BY a0001.item1_str ASC;

WITH
  -- binding due to rank operator
t0000 (item9_str, item10_str, item37_nat) AS
  (SELECT a0000.categorie AS item9_str, a0000.facility AS item10_str,
      DENSE_RANK () OVER
        (ORDER BY a0000.categorie ASC) AS item37_nat
    FROM facilities AS a0000),
  -- binding due to rank operator
t0001 (item9_str, item10_str, item37_nat, item3_str, item4_str,
  item1_str, item2_str, item38_nat) AS
  (SELECT a0001.item9_str, a0001.item10_str, a0001.item37_nat,
      a0002.feature AS item3_str, a0002.meaning AS item4_str,
      a0003.facility AS item1_str, a0003.feature AS item2_str,
      DENSE_RANK () OVER
      (ORDER BY a0001.item9_str ASC, a0001.item10_str ASC,
        a0002.feature ASC, a0002.meaning ASC,
        a0003.facility ASC, a0003.feature ASC) AS item38_nat
    FROM t0000 AS a0001,
      meanings AS a0002,
      features AS a0003
    WHERE a0002.feature = a0003.feature
      AND a0001.item10_str = a0003.facility),
  -- binding due to aggregate
t0002 (pos29_nat, iter30_nat, item31_str) AS
  (SELECT MIN (a0004.item38_nat) AS pos29_nat,
      a0004.item37_nat AS iter30_nat,
      a0004.item4_str AS item31_str
    FROM t0001 AS a0004
    GROUP BY a0004.item37_nat, a0004.item4_str)
SELECT a0005.item31_str, a0005.iter30_nat
  FROM t0002 AS a0005
  ORDER BY a0005.iter30_nat ASC, a0005.pos29_nat ASC;
```