Student Paper: HaskellDB Improved

Björn Bringert
Dept. of Computing Science
Chalmers University of Technology
d00bring@dtek.chalmers.se

Anders Höckersten
Dept. of Computing Science
Chalmers University of Technology
chucky@dtek.chalmers.se

Abstract

We present an improved version of the HaskellDB database library. The original version relied on TRex, a Haskell extension supported only by the Hugs interpreter. We have replaced the use of TRex by a record implementation which uses more commonly implemented Haskell extensions.

Additionally, HaskellDB now supports two different cross-platform database backends. Other changes include database creation functionality, bounded string support, performance enhancements, fixes to the optimisation logic, transaction support and more fine grained expression types.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Languages—*Query languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Data types and structures*; D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming

General Terms

Design,Languages

Keywords

Databases, Haskell, Extensible records, SQL

1 Introduction

HaskellDB [10, 8] is a combinator library for expressing queries and other operations on relational databases in a *type safe* and *declarative* way. The original HaskellDB [10] by Daan Leijen and Erik Meijer introduced the idea of describing database operations using a phantom typed relational algebra-like embedded domain specific language. However, because of its use of TRex [2], the original version of HaskellDB was limited to working with outdated versions of Hugs. The only available database backend used the Microsoft ADO database interface, which is only available under Microsoft Windows. Thus HaskellDB was satisfactory as a proof of concept, but not as a generally useful database library for Haskell.

We have implemented a number of changes to HaskellDB in order to address these problems and make it a more practical database library. This paper has three main parts. First we describe how to write programs using our version of HaskellDB, then we detail our changes and additions, and finally we discuss some of the advantages of and remaining problems with HaskellDB.

2 Motivation

There are two other well-known database interfaces for Haskell:

- *HSQL* is an SQL based interface to several different database systems, currently ODBC, MySQL, PostgreSQL and SQLite. The only user-visible difference between the backends is the need to use different connection functions, all other functions are common to all backends.
- *wxHaskell* [9] is a set of Haskell bindings to the wxWidgets GUI toolkit, which includes an ODBC interface.

SQL based database interfaces like those above and most database interfaces for other programming languages have a number of drawbacks:

- Syntactically or semantically invalid queries are not detected until they are executed. This means that it takes longer to debug queries and that invalid queries may go undetected.
- Queries are created by string manipulation, which can lead to security problems such as SQL injection vulnerabilities [1].
- The developer is forced to work in two separate languages: SQL and a general purpose programming language.

The original HaskellDB seems to be a good way to avoid these problems, and our goal has been to modify it to work with current versions of the mainstream Haskell implementations and with a broader spectrum of database interfaces.

3 Using HaskellDB

We begin by describing briefly how to use HaskellDB. HaskellDB supports a number of different backends. In the following examples we will use the HSQL MySQL driver, but any other driver could be

Submitted to Haskell Workshop 2004.

used by using a different connection function. Queries are written in the same way independently of the backend.

3.1 Getting started

3.1.1 Creating database tables

To use HaskellDB, one needs database tables to work on. It is possible to create these using HaskellDB (see 3.3), or by using some database specific method.

3.1.2 Generating a database description

The database layout description declares the types of the database tables. If the database layout is changed in any way, the layout description needs to be regenerated. However, such changes are normally only made during development, not as part of ordinary program usage.

HaskellDB includes a program called DBDirect to create these database descriptions, see section 4.5.

3.1.3 Connecting to the database

Depending on which underlying database is used, one needs to import an appropriate driver module. To perform queries on the database we define a "connecting function". By using this function, the program becomes independent of the underlying database. When switching to another database, this is the only function that needs to be modified.

```
import Database.HaskellDB.HSQL.MySQL
opts = MySQLOptions {
         server="myServer", db="myDataBase",
         uid="userID", pwd="password"}
withDB :: (Database -> IO a) -> IO a
withDB = mysqlConnect opts
```

3.2 Querying the database

Queries are written in the `Query` monad. To get the result of a query it must be run against the database.

```
query :: GetRec er vr =>
         Database -> Query (Rel er) -> IO [Row vr]
```

The `GetRec` constraint means that the `vr` type is the same as the `er` type, but with normal Haskell value types (such as `Int`) as record values, instead of HaskellDB query expressions of that type (such as `Expr Int`). `Expr` will be introduced later in this section.

The following example prints the contents of the table `farmers`. Note the use of `withDB`.

```
import Farming
desc = withDB $
  \db -> do rows <- query db (table farmers)
            mapM_ (putStrLn.show) rows
```

A typical query often consists of these three parts:

1. *Fetching tables* — select the tables needed for the query
2. *Restrictions on tables* — limit what rows to return
3. *Projections on tables* — limit what fields to return

The next query operates on a database containing names and phone numbers. It prints all the Johns and their respective phone numbers.

Note that a query will only return *unique* rows, in the sense that the result will never contain two identical rows.

```
aQuery =
  do tbl <- table phoneBook
     restrict (tbl!firstName .==. constant "John")
     project (lastName << tbl!lastName #
              phoneNo  << tbl!phoneNo)

listJohns = mapM_ (putStrLn . showRow)
    where
    showRow r = r!lastName ++ " " ++ r!phoneNo
runQuery = withDB $ \db ->
              query db aQuery >>= listJohns
```

The result of `runQuery` might look something like this:

```
> runQuery
Hughes 031-7721001
Doe 031-184442
```

3.2.1 Some query operators

The previous example uses the equality operator `(.==.)`. Most common binary operators have a corresponding operator for use in a query. The types of some representative operators are:

```
(.==.) :: Eq a => Expr a -> Expr a -> Expr Bool
(.<.)  :: Ord a => Expr a -> Expr a -> Expr Bool
(.+.)  :: Num a => Expr a -> Expr a -> Expr a
(.&&.) :: Expr Bool -> Expr Bool -> Expr Bool
```

The operators are translated to the corresponding operators used by the database backend, e.g. SQL operators.

`Expr` provides type safety by adding a phantom type to the underlying type, `PrimExpr`. `Expr` is defined in the following way:

```
data Expr a = Expr PrimExpr
```

The next example illustrates a query which joins two tables and uses an aggregate expression. The table *friends* describes friendship relations between people. For example, `Mary` knows `Chucky`. The second table, *petowners*, describes how many pets a person has. For example, `Chucky` has `4` pets. The query below lists the people in the `person` column of the first table, and the total number of pets that their friends have. We first join the two tables *friends* and *petowners* on the `friend` and `owner` fields. Conceptually, this gives us the intermediate relation *t1*. The final projection gives the relation *result*.

```
q = do f <- table friends
       p <- table petowners
       restrict (f!friend .==. p!owner)
       project  (person << f!person #
                 pets   << _sum (p!pets))
```

3.3 Changing the database layout

It can be useful to be able to create database tables with HaskellDB, for example when distributing a program that uses a database. Here we create the layout for the tables used in the previous example:

Table 1. The *friends* table

person	friend
Björn	Mardy
Mary	Chucky
Björn	Chucky

Table 2. The *petowners* table

owner	pets
Mardy	3
Chucky	4

```
import Database.HaskellDB.Database (createTable)
import MyConnection (withDB)
import Database.HaskellDB.FieldType

createTables =
  withDB $ \db ->
  do createTable db "friends"
       [("person",(StringT,False)),
        ("friend",(StringT,False))]
     createTable db "petowners"
       [("owner",(StringT,False)),
        ("pets",(IntT,False))]
```

3.3.1 Creating a fake field

In certain situations it may be necessary to put information into a field that does not exist in the database. For example, a field of type `Int` is needed to be able to retrieve the results of a `count` operation. Since the record type representation requires that all fields be declared, a "fake" field must be declared if there is no field of the right type. Declaring a new field is somewhat involved, see section 4.2.1 for more information.

4 Implementation

4.1 Overview

HaskellDB consists of two parts, one for specifying operations and one for generating database descriptions. The latter part also has a way of creating a database from a description. This section will mainly cover the part that operates on a database. We only discuss our additions to HaskellDB. The reader is encouraged to read Leijen's PhD Thesis [8] for more information on the basic theory and implementation of HaskellDB.

Queries are written in HaskellDB using a monadic embedded language. The surface layer is a set of phantom typed wrappers around an untyped relational algebra-like structure, PrimQuery. This structure is optimised, converted into the database query language (currently only SQL), and sent to the database via a database binding (currently wxHaskell or HSQL).

Table 3. The *t1* relation

person	friend	owner	pets
Björn	Mardy	Mardy	3
Mary	Chucky	Chucky	4
Björn	Chucky	Chucky	4

Table 4. The *result* relation

person	pets
Björn	7
Mary	4

4.2 Records

The original HaskellDB used the record types offered by TRex [2] to represent the types of relations and database rows. TRex is a Haskell extension which is only available in the Hugs interpreter. In order to make HaskellDB more portable, we have devised a system for extensible records which only requires extensions implemented in both Hugs and GHC. The required extensions include multi-parameter type classes [12], overlapping instances and functional dependencies [6].

A record is either the empty record, or a field and a record:

```
data RecNil = RecNil
data RecCons f a b = RecCons a b
```

The type parameters of `RecCons` are the field label type, the field value type and the type of the rest of the record.

4.2.1 Field labels

Singleton types are used as field labels. The field labels are included in the type of the record, but they are not present in the actual record, as the identity of a field label can be determined from its type. In order to convert records to and from strings, the label must be an instance of the `FieldTag` class:

```
class FieldTag f where
    fieldName :: f -> String
```

For example, in order to introduce a label `Name` with string representation "name", we write:

```
data Name = Name
instance FieldTag Name where
    fieldName _ = "name"
```

In order to make type inference of record selection easier and to get a more TRex-like syntax, we wrap the actual field labels in a label type which includes the type of the associated field value.

```
name :: Attr Name (Maybe Int)
name = mkAttr Name
```

The `Attr` type is HaskellDB-specific; the general record functions can use any type constructor of kind `* -> * -> *`.

4.2.2 Record construction

Record construction is similar to list construction since a record is either the empty record, or a field and a record. We would like to be able to construct records using a concise and intuitive syntax, as is possible with TRex records and ordinary Haskell lists. However, since we do not rely on any syntactic extensions, we are constrained to using normal Haskell operators for record construction. We use `(.=.)` to construct a single-field record from a label and a value, and `(#)` to add a single-field record to a record:

```
(name .=. "Foo" # age .=. 163)
```

If record construction was implemented in the most straightforward way, the user would have to end each record with `RecNil`. In order to avoid this, we introduce the type constructor `Record`:

```
type Record r = RecNil -> r
```

In the `Record` type, the empty record is represented by the identity function. Using this type, the `(.=.)` and `(#)` operators are implemented as follows:

```
infix   6 .=.
infixr  5 #

( .=. ) :: l f a -> a
        -> Record (RecCons f a RecNil)
_ .=. x = RecCons x

( # ) :: Record (RecCons f a RecNil)
      -> (b -> c) -> (b -> RecCons f a c)
f # r = let RecCons x _ = f RecNil
            in RecCons x . r
```

Note that `(.=.)` is not the same as the field construction operator used in HaskellDB queries, `(<<)`. They have the same implementation, but different types, since `(<<)` is used to create records of HaskellDB query expressions with the same labels that are used in the query results.

```
( << ) :: Attr f a -> e a
       -> Record (RecCons f (e a) RecNil)
_ << x = RecCons x
```

4.2.3 Field selection

Since different records have different types, all generic record operations must be overloaded. Most operations are implemented by recursing through the record. Field selection is shown below as an example:

```
class SelectField f r a where
   selectField :: f -> r -> a

instance SelectField f (RecCons f a r) a
   where selectField _ (RecCons x _) = x

instance SelectField f r a =>
         SelectField f (RecCons g b r) a
   where selectField f (RecCons _ r)
         = selectField f r

instance SelectField f r a =>
         SelectField f (Record r) a
   where selectField f r
         = selectField f (r RecNil)
```

Given a field label and a record, `selectField` returns the value of the field with that label. The `SelectField` type class has an instance for the types `f r a` if and only if the record type `r` has a field with a label of type `f` and a value of type `a`.

- The first instance covers the case where the field that we want is at the head of the record. In this case the value in the first field is simply returned.
- The second instance applies when the field exists in the tail of the record, in which case `selectField` is called on the tail.
- The third instance is there to allow using `selectField` directly on elements of `Record r`.

The first two instances overlap, but the first one requires that the label type that we are looking for is the same as the label type in the record field, making it more specific than the second one. Note that this means that if the same label occurs twice in the same record, the first field with that label will be used.

The heavy lifting involved in field selection is done by `selectField`. However, since there are no functional dependencies between the type parameters in the `SelectField` class, the return type of a `selectField` application cannot be inferred from the types of the arguments. The field selection operator `(!)` takes care of this problem, and allows us to use the TRex lookalike labels introduced above.

```
class Select f r a | f r -> a where
    (!) :: r -> f -> a

instance SelectField f r a =>
         Select (l f a) (Record r) a where
    r ! (_::l f a)
      = selectField (undefined::f) r
```

`(!)` is declared in a type class, so that it may be overloaded by users. An example of this is discussed briefly in section 4.7.4.

4.2.4 Evaluation

Our system is less powerful than TRex, but it contains most of the features that HaskellDB requires. The main differences compared to TRex are that we do not have the *lacks* predicate and the *restriction* operation, that the order of the fields is significant and that labels must be declared. Our syntax, especially the type syntax, is more awkward since we do not rely on syntactic extensions. The major advantage is that our system only requires more widely implemented language extensions.

Without the *lacks* predicate, we cannot ensure that labels occur at most once in every record. It seems difficult to implement a *lacks* predicate without type inequality constraints or negated type class instance constraints, neither of which Haskell supports.

Note: After implementing this system, a similar but more ambitious effort to create extensible records using only common Haskell extensions has come to our attention [7]. The main advantage of that system over ours seems to be that it contains a *lacks* predicate.

4.3 Bounded strings

When using the SQL types VARCHAR(*n*) and CHAR(*n*), database systems tend to silently truncate inputs longer than *n*. To be able to give users feedback and *type safety* against this, we allow treating VARCHAR and CHAR fields as *bounded strings*.

The maximum length of a bounded string is encoded in its type. Having this information in the type can prevent unexpected loss of data.

4.3.1 Usage

For example, a bounded string with maximum size 255 is created using either `trunc` or `toBounded` as follows:

```
newStr :: BStr255
newStr = trunc "teststring"

newMaybeStr :: Maybe BStr255
newMaybeStr = toBounded "teststring"
```

`toBounded` returns `Nothing` if the argument is larger than the specified maximum size, which enables runtime feedback. If `trunc` is used to create bounded strings no such feedback is possible. Existing bounded strings can be manipulated using `shrink` or `grow`:

```
shrink :: (Size n, Size m) => BoundedList a n
       -> Maybe (BoundedList a m)

grow :: LessEq n m =>
        BoundedList a n -> BoundedList a m
```

Using the example above, `shrink newStr` will evaluate to `Nothing` since the length of “teststring” is larger than 8.

4.3.2 Implementation details

The implementation of bounded strings are based on a more general data type called `BoundedList` as follows:

```
type BoundedString n = BoundedList Char n
type BStr1   = BoundedString N1
type BStr2   = BoundedString N2
             .
             .
type BStr255 = BoundedString N255
```

Here `BoundedList Char n` is a list of `Char` with maximum size set by the phantom type parameter `n`. To implement bounded lists, each possible size is declared using two type classes:

```
class Size n where
    size :: n -> Int

class (Size a, Size b) => Less a b
```

Sizes are declared as singleton types:

```
data N255 = N255
```

A more natural encoding would be:

```
data Zero = Zero
data Succ n = Succ n
```

However, for large bounds such as 65535 (the maximum size of some SQL data types) such types become large enough to be very impractical.

The `Size` instance is used by `toBounded` when determining whether the list to be converted will fit inside the type or not.

```
instance Size N255 where size _ = 255
```

The `Less` relation between the different sizes can be built up in two different ways:

1. *Building up the relation inductively*

   In HaskellDB we have used this implementation since it requires much less code and results in faster compilation than the second approach.

   ```
   instance Less N254 N255
   instance Less a N254 => Less a N255
   ```

   The first instance declares that the largest smaller size is less than this size. The second instance ensures that any size smaller than the next smaller size is also less than this size. This definition requires that the Haskell implementation supports “undecidable instances”. Note that the use of this extension is safe in this case, as `Less` is a well-founded order on the set of sizes.

2. *Hardcoding the relation*

   This requires $\theta(n^2)$ instances and takes very long time to type check. We mention this approach here because it is the simplest and does not require the “undecidable instances” extension.

   ```
       instance Less N1 N255
       instance Less N2 N255
                .
                .
       instance Less N254 N255
   ```

The class `LessEq` used in the declaration of `grow` is defined as follows:

```
class (Size a, Size b) => LessEq a b
instance (Size a) => LessEq a a
instance (Size a, Size b, Less a b) => LessEq a b
```

This ensures that `grow` is not used to shrink bounded lists.

4.4 Backend independence

In the original version of HaskellDB, the type of database handles was parametrised over the database driver’s connection handle type and the type used for result rows in the driver. This made it difficult to write a generic connection function that chooses the database driver to use based on, for example, a connection string.

The database handle parameter was removed by using an idiom similar to interface implementation in an object oriented language. This was inspired by a similar system used in the implementation of HSQL. The general HaskellDB driver interface does not know anything about the private data needed by the implementation to talk to the database. Below, we will use the implementation of the `dbInsert` function in the HaskellDB HSQL driver as an example. The HSQL driver has a function `hsqlInsert` which implements `dbInsert`. When the `Database` object is created, `hsqlInsert` is partially applied to the connection handle (which is of the HSQL-specific type `Connection`), giving a function which can be used as `dbInsert`.

Generic database driver interface:

```
data Database = Database
    { ...,
      dbInsert :: TableName -> Assoc -> IO (),
      ... }
```

Database driver implementation:

```
mkDatabase :: Connection -> Database
mkDatabase connection = Database
   { ...,
```

```
    dbInsert = hsqlInsert connection,
    ... }

hsqlInsert :: Connection -> TableName
           -> Assoc -> IO ()
```

After establishing a connection to the database, the HSQL driver passes the connection handle to `mkDatabase`, which returns a `Database` object where all the functions have been partially applied to the connection handle.

The second obstacle to making a single `Database` type for all drivers was the result row type parameter. Since we have implemented records, and query results have record types, it seemed natural to make result rows records. This means that all database drivers now use the same representation for query results — a list of records.

4.5 DBDirect

DBDirect is a program which automatically generates database descriptions from an existing database. Originally, DBDirect was only able to generate descriptions for databases using the Microsoft ADO interface, but the current version works with any database for which there is a HaskellDB driver.

However, users soon requested a way to generate databases directly from Haskell code. We decided to implement this feature and at the same time make the system more general. We call this system DBSpec.

DBSpec is rather simple in its construction, yet powerful. It is implemented as a record of the type:

```
data DBInfo = DBInfo {dbname :: String,
                      opts :: DBOptions,
                      tbls :: [TInfo]}
```

`DBOptions` describes the particular options we wish to use (currently only whether we want bounded strings or not). `TInfo` describes a table in the database, and is implemented similarly to `DBInfo`.

This system provides several advantages compared to the old one:

- Can generate databases directly from Haskell — no knowledge of SQL is needed.
- Easier overview of databases.
- Easier to extend than the original version.

4.6 Optimisation fixes

A few problems were found in the optimisation code in the original HaskellDB.

4.6.1 Aggregates in restriction and ordering

Pushing an ordering or a restriction through a projection can cause the ordering to use an aggregate expression, which is not allowed in SQL. This was fixed by not pushing orderings and restrictions through projections if it will make them contain any aggregate expressions.

4.6.2 Merging projections through multiplication

In the original HaskellDB, "*project (project op project)*" would be optimised by removing the outer projection and merging it with the two inner ones. This does not work if the operator is multiplication, since the outer projection will use attributes defined in only one of the inner projections. Thus this optimisation was disabled when the operator is multiplication.

4.6.3 Projections on empty queries

Originally, a projection on an empty query would be optimised to the empty query. This cannot be done in general, as the projection may contain constant entries that do not come from some inner relation. Thus this optimisation was removed.

4.7 Minor changes

4.7.1 Fine grained expression types

In the original version of HaskellDB there was a single type for all expressions. However, in SQL some classes of expressions can only occur in certain parts of a statement. To allow the type system to check this we add two more types of expressions, which are described in the following sections, and introduce a type class for expressions:

```
class ExprC e where
    primExpr :: e a -> PrimExpr

instance ExprC Expr where
    primExpr (Expr e) = e
```

4.7.2 Aggregates only in projections

One example of expressions which cannot be used everywhere are aggregate expressions, such as those using `\_sum` or `count`. In SQL, aggregate expressions can only be used in projection clauses, and it would be desirable for the HaskellDB type system to enforce this restriction. This problem was solved by introducing a separate type for aggregate expressions, `ExprAggr`, and a type class, `ProjectExpr`, for expressions that can be used in projections.

4.7.3 Default values and auto increment columns

SQL supports the notion of default values for columns. The *auto increment* (or *serial*) features found in for example MySQL and PostgreSQL interpret the default value as being the next value in the sequence used for the column values. In order to support these features, we have added a `\_default` construct with type `ExprDefault a`. In order to restrict the use of default values to insertions, a new type class for expressions that can be used in insertions, `InsertExpr`, was added.

4.7.4 Overloaded field selection

Originally, the operator `(!)` was used in HaskellDB for field selection inside queries, and `(!.)` for field selection in query results. From experience working with HaskellDB and helping users, we found that this distinction is difficult to keep in mind. To provide a more unified interface, the two operators were replaced by an overloaded `(!)` operator.

4.7.5 Case

A case construct was added to support conditional evaluation in queries:

```
_case :: [(Expr Bool, Expr a)] -> Expr a -> Expr a
```

The boolean expressions are evaluated until one of them matches, and the value of the corresponding expression is returned. If no condition is true, the value of the second argument is returned.

4.7.6 Converting from nullable types

The original HaskellDB has no facilities for converting from nullable to non-nullable types. We added a `fromNull` function to do this:

```
fromNull :: Expr a -> Expr (Maybe a) -> Expr a
```

4.7.7 Transactions

Support for transactions has been added. The transaction function starts a transaction, performs some database operations and commits the changes. If an exception is raised during the execution of the database operations, the transaction is rolled back.

```
transaction :: Database -> IO a -> IO a
```

5 Evaluation

5.1 Advantages

This section describes some of the advantages of using HaskellDB compared to using most SQL based systems.

5.1.1 Query correctness

Syntactic correctness and type correctness for all database operations are checked by the Haskell compiler at compile time, instead of by the database system when the operation is performed.

Automatic quoting of supplied constants prevents SQL injection vulnerabilities [1].

5.1.2 Ease of programming

With HaskellDB the program/debug cycle is much faster since most errors are caught at compile time instead of at runtime.

There is no need for the programmer to use or even know SQL; knowledge of Haskell and relational database basics should suffice.

5.1.3 Expressive power

Since queries are written in Haskell, the full power of the Haskell programming language is available. This means that common patterns can be abstracted out. For example, one can write a query that does all the necessary joins and projections to get some often used relation, and then use that query in constructing more specific queries by adding restrictions, projections and orderings.

5.1.4 Platform independence

A HaskellDB program can use any supported backend by simply changing the connection function used.

5.2 Disadvantages

This section outlines some of the drawbacks of HaskellDB in its current state and proposes some possible solutions.

5.2.1 Type errors

The type error messages produced by invalid HaskellDB programs can be quite verbose and difficult to understand without intimate knowledge of the record type implementation. For example, if you try to select a field from a record which does not have that field, the compiler will complain that there is no instance of `SelectField` for that record type. Due to the implementation of record types with algebraic data types, the record type itself will be quite large. Ideally the compiler should report that there is no field with that label in the record, and give a concise representation of the record type.

There a number of ways in which this problem could be mitigated:

- In *Helium* [4] a system for scripting the type inference process [3] is implemented. This could potentially be used to generate better error messages for type errors related to the record types.
- *Having records as a language extension*, like TRex, would be likely to lead to more understandable error messages.
- *Using infix type constructors* (as supported by GHC) could make the record type representations more palatable.

5.2.2 Non-standard SQL

The SQL92 standard [5] is silent on a number of issues. One example is whether `LIKE` should perform case sensitive or case insensitive matching. Another example is that SQL92 does not provide a way to limit the number of results returned by a query. As a result, different database systems have implemented these features differently. This means that the behaviour of HaskellDB programs are not completely independent of which backend is used.

This problem could be partly solved by letting the database drivers have a say in how HaskellDB queries are compiled to SQL. However, this would be difficult with drivers such as ODBC, which support multiple backend systems.

5.2.3 Declaring field labels

As described in section 3.3.1, if the fields declared in the database layout module are not enough to create all desired queries, new labels must be explicitly declared.

The effort involved in declaring field labels can be reduced by using a simple Template Haskell [13] function. The HaskellDB distribution includes an example of how that can be done.

5.2.4 Repeated field labels

The current record implementation allows a field label to be used multiple times in the same record. This leads to undesirable results, as it may cause the wrong value to be retrieved or invalid SQL to be generated. This could be solved by implementing a *lacks* predicate for the record types [7].

5.2.5 Concurrent lazy queries

Some database drivers do not allow a client to have multiple query results open concurrently. This means that if the client performs a lazy query and does not use all the results before performing a new query, an exception may be raised.

5.2.6 Recompilation

HaskellDB programs must be recompiled whenever the database layout is changed. Since the type system depends on the database layout, this is difficult to avoid. However, the application normally needs to be modified anyway whenever the layout of some part of the database changes. An interactive Haskell interpreter may be used to reduce the time needed for recompilation.

5.2.7 Dynamic driver loading

Since there is no standard way to load Haskell modules dynamically, HaskellDB programs must import, and thus be linked against, all the database drivers that they may use. This makes it difficult to write and distribute compiled database driver independent programs.

This problem could be solved by using the *hs-plugins* system for dynamic module loading [11].

6 Conclusions

Our users and we ourselves have implemented a number of applications using HaskellDB. Existing applications include a web forum, blogging software, a mailing list search application, an RSS feed generator and a web based collaboration tool. The static checking, power and portability of HaskellDB make it a productive environment for developing database applications in Haskell.

We have shown that (somewhat limited) extensible records can be implemented without extending Haskell beyond the extensions already available in Hugs and GHC. We have also implemented bounded lists in Haskell with extensions, something which is typically used as an example of the usefulness of dependent types.

The version of HaskellDB discussed in this paper, along with documentation and further examples, is available from
`http://haskelldb.sourceforge.net/`

Acknowledgements

We would like to thank John Hughes for supervising our project and for giving feedback on this paper, Daan Leijen and Erik Meijer for creating the original HaskellDB, Krasimir Angelov for his work on HSQL, Shae Erisson, Jeremy Shaw, Tom Moertel, Esa Ilari Vuokko, Thomas Bevan, Martijn Wijffelaars and Kees van Kooten for suggestions, patches and bug reports, and the anonymous referees for their many helpful comments and suggestions.

7 Additional Authors

Conny Andersson, `forester@dtek.chalmers.se`
Martin Andersson, `mardy@dtek.chalmers.se`
Mary Bergman, `d99mary@dtek.chalmers.se`
Victor Blomqvist, `viblo@dtek.chalmers.se`
Torbjörn Martin, `torma@dtek.chalmers.se`

Department of Computing Science
Chalmers University of Technology

8 References

[1] CERT Vulnerability Note VU#282403, Sept. 2002.

[2] B. R. Gaster and M. P. Jones. A Polymorphic Type System for Extensible Records and Variants. Technical Report NOTTCS-TR-96-3, Department of Computer Science, University of Nottingham, Nov. 1996.

[3] B. Heeren, J. Hage, and S. D. Swierstra. Scripting the type inference process. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 3–13. ACM Press, 2003.

[4] B. Heeren, D. Leijen, and A. van IJzendoorn. Helium, for learning Haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 62–71. ACM Press, 2003.

[5] International Organization for Standardization. *ISO/IEC 9075:1992: Title: Information technology — Database languages — SQL*. International Organization for Standardization, Geneva, Switzerland, 1992. Available in English only.

[6] M. P. Jones. Type Classes with Functional Dependencies. In *Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 230–244. Springer-Verlag, 2000.

[7] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *ACM SIGPLAN Haskell Workshop (HW'04)*. ACM Press, Sept. 2004.

[8] D. Leijen. *The λ Abroad - A Functional Approach to Software Components*. PhD thesis, Utrecht University, Nov. 2003.

[9] D. Leijen. wxhaskell – a portable and concise GUI library for Haskell. In *ACM SIGPLAN Haskell Workshop (HW'04)*. ACM Press, Sept. 2004.

[10] D. Leijen and E. Meijer. Domain-Specific Embedded Compilers. In USENIX, editor, *Proceedings of the 2nd Conference on Domain-Specific Languages (DSL '99), October 3–5, 1999, Austin, Texas, USA*, pages 109–122, Berkeley, CA, USA, 1999. USENIX.

[11] A. Pang, D. Stewart, S. Seefried, and M. M. T. Chakravarty. Plugging Haskell In. In *ACM SIGPLAN Haskell Workshop (HW'04)*. ACM Press, Sept. 2004.

[12] S. Peyton Jones, M. Jones, and E. Meijer. Type classes: Exploring the design space. In *Proceedings of the Second Haskell Workshop*, June 1997.

[13] T. Sheard and S. Peyton Jones. Template metaprogramming for Haskell. In M. M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, Oct. 2002.