MetaHDBC
Statically Checked SQL for Haskell (Draft)

Mads Lindstrøm
`mads_lindstroem@yahoo.dk`

September 18, 2008

Abstract

Relational databases are ubiquitous in academia as well as the enterprise. Therefore, any general purpose language should have good support for interfacing with such databases. Inhere I present a database binding for the general purpose language Haskell. This novel approach accesses databases at compile-time using metaprogramming. By leveraging existing libraries, Template Haskell and ODBC, we achieve a simple system that still provides the features a database vendor has to offer, including as much implicit static typing as a database vendor provides type inference.

1 Introduction

Databases are common within computing and are used with most enterprise systems like customer relationship management, banking and web server applications. Also embedded databases are used in standalone applications like the Firefox web browser and Apple mail.

It is popular to use ODBC for accessing databases. ODBC is cross platform with respect to operating system, programming language and database backend. Being cross platform makes ODBC attractive for programmers, as they get access to numerous databases running on multiple operating systems using only one interface. Unfortunately ODBC libraries postpone the evaluation of queries until run-time and they therefore lack static typing and syntax check.

Another approach to database access is *embedded SQL* as seen in SQLJ[5][8] for Java and for other languages[6] such as C, Ada and COBOL. As with ODBC the programmer writes ordinary SQL, but instead of evaluating the SQL at run-time, a preprocessor extracts and parses the SQL. Compared with ODBC this adds the possibility of strict typing and syntax check.

In this paper I present *MetaHDBC*[1] that is a database binding for Haskell[1] similar to embedded SQL, though with the difference that MetaHDBC use compile-time metaprogramming instead of preprocessing. I build MetaHDBC upon ODBC, as ODBC has support for a wide variety of *RDMS*s (Relational Database Management System) and as ODBC supports type inference.

In this paper I describe MetaHDBC, which has the following characteristics:

- A programmer can easily extend MetaHDBC with new functionality. E.g. make a function, that executes a SQL statements and logs the executed statement to a local file (section 7).
- Integrated with the host language by giving access to host-language bound variables in SQL statements (sections 4.2 and 5).
- Has compile-time syntax and type -check of SQL statements (sections 4, 4.1 and 5). MetaHDBC validates statements against actual database

[1]Source code and installation notes available at `http://www.haskell.org/haskellwiki/MetaHDBC`

metadata. For example, MetaHDBC will catch misspelled column names at compile-time (section 8).

- Has type inference of SQL statements (sections 4, 4.1 and 6).
- Can leverage the existing error reporting in RDMSs and compilers (section 8).

Furthermore, I explore the ability of popular databases to do type inference (section 6).

2 Database access using HDBC

As ODBC presents a low level interface to RDMSs, I only use it indirectly though the HDBC library. HDBC heightens the level of abstraction by both abstracting over multiple interfaces (one of them being ODBC) and by hiding implementation details in the underlying interfaces.

We will examine the use of HDBC via an example, which extracts data from a car table and present the result to the user. The car table consists of the car's make and top speed:

Column Name	Column Type
Make	VARCHAR(25)
Top speed	Int

The table contains two rows:

Make	Top speed
Honda	160
Lotus	220

Being hooked on speed, our only interest is cars with a top speed above 200 kilometres per hour and in figure 2 you can see how to extract these cars. We use the function *printCar*, seen in in figure 1, to show the extracted cars.

The program in figure 2 first connects to the RDMS using a *DSN* (Data Source Name). A DSN describes a database instance to the underlying ODBC library. Then, in line three, we execute a SQL statement and passes the topSpeed to the RDMS. Line three also fetches all the cars. Finally we print all the cars we retrieved.

Retrieving cars, as shown in the example, contains a lot of possibility for errors that is not caught until run-time. First of all, ODBC checks the SQL string for syntax and type errors at run-time, not compile-time. Second, when we execute the SQL we pass in a number of parameters. Again ODBC check the number and the types of these parameters at run-time, not compile-time. Third, when retrieving values we see the lack of type checking clearly, as the program pattern match against a list with exactly two elements, with no static guaranties that it actually has two elements. Finally, when ODBC converts the types to the Haskell types *String* and *Int* we risk run-time type failures again.

As described above, HDBC postpones most error handling until run-time. HDBC do *not* fail in this respect. Quite contrary to a failure, HDBC do one thing and do it well. Nevertheless, building a layer on top of HDBC could give us these static guaranties. In the following sections we will see how to add these safety measures using metaprogramming and HDBC.

3 Metaprogramming with Template Haskell

A metaprogram manipulates existing code or outputs new program code, or both. *TH* (Template Haskell)[9] is an extension to Haskell, which allows metaprogramming to occur at compile time. This section gives an informal introduction to TH.

TH can evaluate any Haskell code, even arbitrary IO, at compile-time. After evaluation, TH splices the resulting code back into the main program. An example:

```
do putStrLn $( let x = show (3 + 5)
               in [| x |]
             )
```

TH evaluates *show (3 + 5)* and splices the result (the string "8") into the main program resulting in:

```
do putStrLn "8"
```

```
printCar :: String -> Int -> IO ()
printCar make topSpeed =
    putStrLn ("A " ++ make ++ " with a top speed of " ++
              show topSpeed ++ " kilometre per hour")
```

Figure 1: Printing cars

```
do conn <- connectODBC dataSourceName
   let sqlString = "SELECT make, topSpeed FROM Car WHERE topSpeed > ?"
   cars <- quickQuery conn sqlString [toSql (200::Int)]
   let printCar' [make, topSpeed] =
       printCar (fromSql make) (fromSql topSpeed)
   mapM_ printCar' cars
```

Figure 2: Selecting fast cars with HDBC

the Haskell compiler then compiles this program *as if* it had been written directly.

The code *$(expression)* is called a splice. It means: Evaluate the expression (of type *ExpQ*) between the parenthesis and compile the result as if it had been written directly.

The code *[| ordinary Haskell code |]* is called Quasi notation. It turns the ordinary Haskell code into an expression of type *ExpQ*. We can manipulate this expression further or we can splice it back into the main program using the *$(...)* notation.

An essential feature, for us, is TH's ability to evaluate arbitrary IO. That is, we can communicate with the world outside the compiler and use the result as a basis for producing code. Specifically, MetaHDBC asks a RDMS to evaluate SQL statements and return whether the SQL is syntactically correct, whether the SQL type checks, the parameter types and output types. This is the basis for producing strictly typed SQL statements.

4 MetaHDBC by Example

In this section we will see how to access a database using MetaHDBC, mimicking the car query created in section 2.

The MetaHDBC program, you can see in figure 3, follows the same pattern as the HDBC program. That is, we establish a connection, execute the statement and finally print the result using the *printCar* function. But with the difference that MetaHDBC statically checks the SQL statement.

The careful reader will notice that the function *runStmt* is spliced into our program. Thus the query is partially processed at compile-time, enough so that the compiler knows the queries type.

4.1 Data manipulating SQL expressions

We are not satisfied with just extracting information from a database, we also want to store information into it. In figure 4 we can see a function which inserts a row into the Car table using the cars make and its top speed as parameters.

The function connect to the RDMS and execute the insert statement. In addition to the previous examples, we also commit the transaction, as we want to store the new row permanently.

For the sake of clarity, we have annotated the *insertStmt* function with types. If we had omitted the types the compiler would have inferred them for us.

```
  do conn <- connectODBC dataSourceName
     cars <- $(runStmt dataSourceName
               "SELECT make, topSpeed FROM Car WHERE topSpeed > ?") conn 200
    mapM_ (\(make, topSpeed) -> printCar make topSpeed) cars
```

Figure 3: Selecting fast cars with MetaHDBC

```
insertStmt :: String -> Int -> IO ()
insertStmt carMake carSpeed =
  do conn <- connectODBC dataSourceName
     $(runStmt dataSourceName
       "INSERT INTO Car (make, topSpeed) VALUES (?, ?)") conn carMake carSpeed
     commit conn
```

Figure 4: Insert car

4.2 Access to bound variables

We have seen how to use positional parameters, but they can get confusing as their number increase. As seen in figure 5, we can also bind the values directly into the SQL.

The key difference between this and the former example is in line two. In stead of using anonymous parameters, we use the already bound variables *carMake* and *topSpeed*.

5 RunStmt dissected

Previously we accessed a RDMS using MetaHDBC's function *runStmt*. In this section we examine what happens when GHC[2] encounters a call to *runStmt*.

Figure 6 shows *runStmt* as a flow diagram. For clarity of explanation, the diagram depicts a logical flow, rather than a flow 100% true to its implementation.

At "Infer Types" MetaHDBC infer the type of a SQL expressions. The expression may contain host-language bound variables (see section 4.2) and these variables are discovered at "Simple Parser" by MetaHDBC. Second, HDBC asks a RDMS to infer the type of the SQL expression (without host-language bound variables).

[2]Glasgow Haskell Compiler, see `http://www.haskell.org/ghc`

The next step "Create TH expression" is split into two steps. Namely the creation of the left hand side (LHS) and creation of the right hand side (RHS).

The LHS consists of the parameters inferred by the RDMS and a connection parameter. At run-time the connection parameter decides which database the SQL expression should be executed upon. Note that the database used for type inference at compile-time need *not* be the same database used for execution at run-time.

The RHS is a more complex endeavour. All input parameters and output values to HDBC has type *SQLValue*. HDBC, at run-time, converts the *SQLValue*-s to what the RDMS needs. This carries the risk of run-time type failures, but by MetaHDBC's earlier compile-time access to the database MetaHDBC knows the types of input and output -values. MetaHDBC use this knowledge to create two conversion function. One that converts input parameters into *SQLValue* (the step "Input conversion function") and one that converts output values from *SQLValue* into a type safe Haskell representation (the step "Output conversion function").

Before MetaHDBC creates the RHS it must infer if this is an updating (update, delete or insert) SQL

```
insertStmtBound :: String -> Int -> IO ()
insertStmtBound carMake carSpeed =
  do conn <- connectODBC dataSourceName
     $(runStmt dataSourceName
        "INSERT INTO Car (make, topSpeed) VALUES (?carMake, ?carSpeed)") conn
     commit conn
```

Figure 5: Insert car using bound variables

statement or a query (select). If an SQL expression contains any return values it must be a query, otherwise it is an updating statement, as selects must choose at least one value. Query versus update do not just result in different calls to HDBC, but also affects the return type of *runStmt*. When we do an update we want to return the number of rows affected, whereas a query should return the result of the query.

Finally, MetaHDBC construct the expression and pass it onto GHC (in the step "TH/GHC") for evaluation and compilation.

6 Quality and consistency of type inference

MetaHDBC rests upon RDMSs ability to infer types and I have therefore investigated these abilities for select, insert, delete and update -statements. All statements contained a where clause, except for the insert statement. My experiment[3] used two tables each containing two columns, namely a primary key column and a "value" column. The tables differed in that one tables "value" column could contain null values, the other tables "value" column could not. I executed each statement on both tables and observed each RDMS's ability to infer the type of the "value" column.

Note that, parameters in where clauses is never nullable, as this is expressed as: "... WHERE ... IS NULL" or as "... WHERE ... IS NOT NULL".

[3]Source code for experiment can be found at `http://code.haskell.org/MetaHDBC/test/ColumnTests/`

I did the experiment on five different RDMSs and you can see the results in table 7. "Yes" means that the RDMS inferred the type or nullability correctly for both nullable and not-nullable columns. "No" means it did is not. We only see one pair (type and nullable) for the where clauses (opposed to one for each of select, insert, delete and update -statements), as the results proved to be independent of the statement type for all RDMSs. I left the delete statement out of the table, as its only parameters is in the where clause.

The results are less than encouraging, as only DB2 (**FIXME: and maybe SQL server**) do a decent job of inferring types and nullability and even DB2 do only infer nullability for output values (select).

7 Extensibility

It is harder to extend a preprocessors than it is to extend an ordinary library. The former requires changing the preprocessor internals, whereas the latter can be achieved by composing the library functions into new functions and possibly adding new code. For example, the preprocessor based parser generators Yacc and Harpy are hard to extend, whereas the parser combinator library Parsec is easily extendable.

We face a similar issue with embedded SQL and ODBC. We can easily extend ODBC to log all database access, but cannot easily extend embedded SQL. Similarly, we can extend MetaHDBC, by binding some of the parameters in the call to *runStmt*:

```
runStmtWithBoundDSN :: String -> ExpQ
runStmtWithBoundDSN sql =
    runStmt "our_data_source_name" sql
```

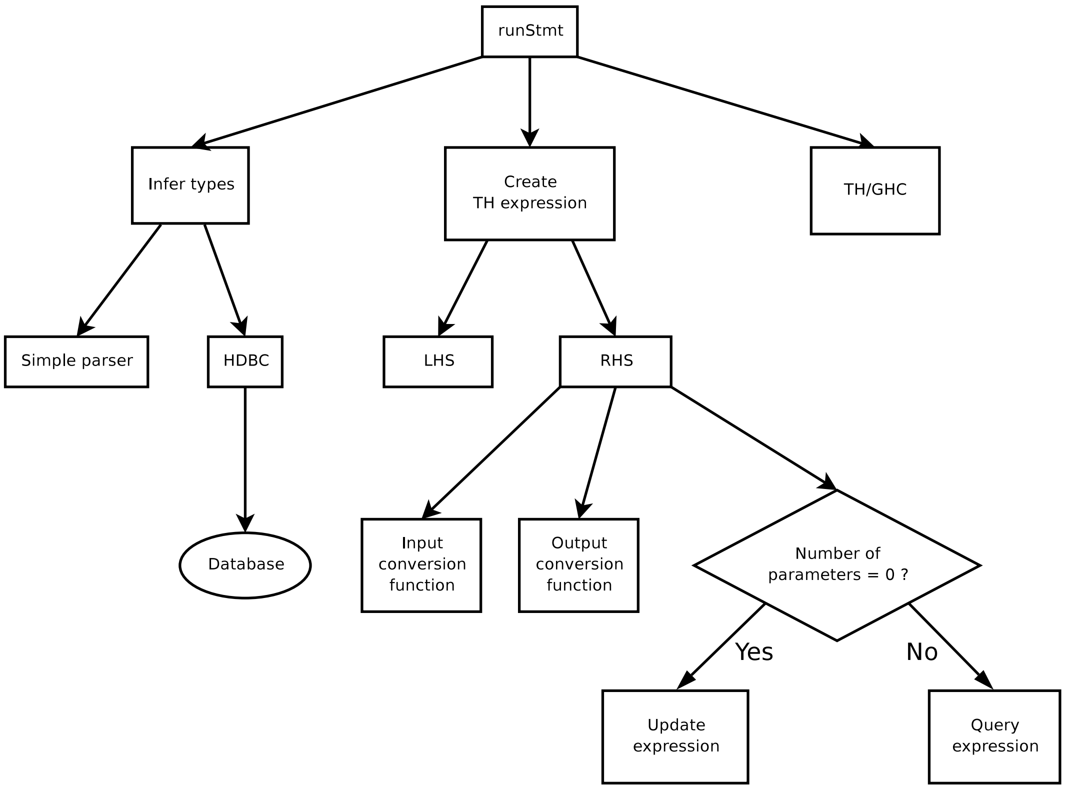

Figure 6: Program-flow for runStmt

	Select		Where		Insert		Update	
Database vendor	Type	Nullable	Type	Nullable	Type	Nullable	Type	Nullable
DB2	Yes	Yes	Yes	No	Yes	No	Yes	No
MySQL	Yes	Yes	No	No	No	No	No	No
PostgreSQL	Yes	No	No	No	No	No	No	No
SQLite	No	No	No	No	No	No	No	No
SQL Server	**FIXME: Not done yet - Help needed as I lack access to SQL Server**							
	FIXME: If anybody is interested in testing SQL server, please contact the author.							
	FIXME: It would also be interesting to test on Oracle.							

Figure 7: Correct type inference

runStmtWithBoundDSN binds the DSN. In this way we can control, from one place, which database is accessed. It just requires that we use *runStmtWithBoundDSN* in stead of *runStmt*.

When we want to let our extension depend upon the parameters to the SQL expressions (the question marks seen in earlier examples), we must use Template Haskell. In figure 8 we can see how to extend MetaHDBC with the ability to log all database access to standard output. The logging includes parameters given to the database.

In figure 8 the MetaHDBC function *makeExprParts* makes parts that will be needed later, by inferring types (as explained in section 5) and by creating (or extracting) names for (bound) variables. The MetaHDBC functions *runStmtLHS* and *runStmtRHS* creates the LHS and RHS, by using the parts returned in *makeExprParts*. Finally, we see the code to print the parameters, the executed SQL and the result of executing the SQL.

8 Error reporting

Helpful error messages are important for any program development tool as it directly impacts productivity. Bad error messages leaves the programmer doing trial and error to figure out the exact problem. In contrast, helpful error messages makes spotting and correcting programmer errors easy.

MetaHDBC leverages the existing ability of RDMSs and compilers to do error reporting, by using the RDMS to provide a textual explanation of the problem and by leveraging the compiler to provide location and context.

In figure 9 and 10 we show an erroneous program and the error message we get when trying to compile the program. *GHC's* (Glasgow Haskell Compiler) error messages can be quite large as they provide increasingly larger contexts. In the interest of saving space I have therefore only shown the beginning of the first context.

This error message is helpful to the programmer as:

- the location of the problem is reported (on the first line). Clearly a programmer needs to know the location of a problem to be able to fix it.
- a textual description of the problem is reported, which tells the programmer that "CARMKE" is invalid (missing an 'A').
- increasingly larger context are provided to the programmer. Often programmers find it difficult to understand errors in isolation. GHC helps with providing context.

MetaHDBC achieve error message reporting by using ODBC's and TH's ability to report errors. By calling the standard Haskell function fail TH will report the error to the user. As MetaHDBC prepares the SQL statements at compile-time we also get error messages at compile-time.

9 Related work

9.1 ODBC / HDBC

MetaHDBC builds upon HDBC and therefore shares some of its advantages: Access to variables bound in the host language from within SQL statements and extensibility (section 7). Albeit the last advantage is less prominent with MetaHDBC, as it sometimes requires using Template Haskell, which are harder to program than ordinary Haskell code.

As MetaHDBC communicates with the RDMS at compile time (section 5), it can do syntax and type -check at compile time (section 8). In contrast, HDBC postpones all checks until run-time. Like MetaHDBC, HDBC can also do type inference, but the type information is more valuable with MetaHDBC, as it is available earlier at compile time.

MetaHDBC also provides error reporting earlier, at compile-time. Furthermore, MetaHDBC's error messages are more useful as GHC assist us with the erroneous codes context (section 8).

9.2 Embedded SQL

Both MetaHDBC and embedded SQL communicates with the RDMS before run-time and therefore they

```
loggedRunStmt :: String -> String -> ExpQ
loggedRunStmt dsn extendedSql =
    do parts <- makeExprParts dsn extendedSql
       let parmsTup = tupE (map (varE . parmName) $ parameters parts)
       runStmtLHS parts
            [| do res <- $(runStmtRHS parts)
                  putStrLn ("Ran statment: " ++ extendedSql ++
                            " with parameters: " ++ show $(parmsTup))
                  putStrLn ("Showing result: " ++ show res)
                  return res
             |]
```

Figure 8: Logged database access

```
main = do
  conn <- connectODBC dataSourceName
  cars <- $(runStmt dataSourceName
            "SELECT carMake, topSpeed FROM Car WHERE topSpeed > ?") conn 127
  return ()
```

Figure 9: Non-existing column

```
Errors/MissingColumn.hs:10:20:
    Exception when trying to run compile-time code:
      user error (Error while getting type information from the database server.
Error message from database: SQLNumResultCols: -206: [IBM][CLI Driver][DB2/LINUX]
SQL0206N  "CARMAKE" is not valid in the context where it is used.
SQLSTATE=42703


The error orcurred while preparing:
SELECT carMake, topSpeed FROM Car WHERE topSpeed > ?
)
      Code: runStmt
              dataSourceName
              "SELECT carMake, topSpeed FROM Car WHERE topSpeed > ?"
                  conn 127
    In the expression:
        do conn <- connectODBC dataSourceName
           ...
```

Figure 10: Non-existing column error message

both do compile time syntax and type -check. I am unsure if any embedded SQL tools do type inference, but it is conceivable that they could, just like MetaHDBC uses type inference.

Embedded SQL error reporting should also be comparable to MetaHDBC.

It is easier to extend MetaHDBC than it is to extend embedded SQL, as the latter requires modifying a preprocessor and not just extending a (Template Haskell) function (section 7).

Finally, MetaHDBC has better integration with the host language (Haskell), by giving access to host-language bound variables in SQL statements (section 4.2).

9.3 HaskellDB

In the easily read yet still inspiring paper: "Domain specific embedded compilers" Leijen et al.[7] introduces HaskellDB[3], which shares many of MetaHDBC's advantages. Both are strictly typed. Both has type inference. Both has Haskell as their host language. Nevertheless, their designs are fundamentally different, as HaskellDB completely hides the SQL language. Instead programmers write queries in ordinary Haskell code, which HaskellDB translates to SQL and parses on to the RDMS. This is also know as a *DSEL* (Domain Specific Embedded Language)[4], as it is a domain specific language embedded and written in the host language.

Being a DSEL gives HaskellDB several advantages relative to MetaHDBC (list inspired by [7] and a haskell-cafe post by Björn Bringert[2]):

- User-build query combinators
- Independent of RDMS
- Programmers do not have to learn SQL
- Programmers do not have to mentally swap between Haskell and SQL while programming

Imperative programming languages typically has multiple *build-in* control flow constructs, such as while, for, and for-each statements. Functional languages have no use for these constructs, as they are imperative in nature, but they have no *build-in* functional counterparts either. Instead functional languages typically rely on user (or library) defined functions, such as map or fold, which means that programmers can define their own "control-flow" functions suited to their specific needs. This is a powerful concept, which HaskellDB carries on to query languages by letting programmers write their own query combinators.

The biggest issue for MetaHDBC is its RDMS dependence. Not only do different RDMSs have different SQL dialects, but their ability to do type inference is also different (section 6). HaskellDB sidesteps this issue by adding a layer between SQL and the programmer. A layer that abstracts away differences between RDMSs.

By being a DSEL, HaskellDB also sidesteps the issue of having to learn and use SQL. However, as relational databases is ubiquitous and as SQL is the "standard" query language, supported by all major database vendors, it is conceivable that most programmers already knows SQL. Another advantage of hiding SQL is that programmers do not have to mentally swap between two different languages while programming, which may become cumbersome (and thus inefficient).

In Björn's haskell-cafe post[2] he also mentions disadvantages of HaskellDB relative to MetaHDBC:

- Poorly optimised queries
- Missing SQL features (e.g. outer joins)
- Poor support for DB-specific features
- Difficult to understand type errors

FIXME: I am unsure of when HaskellDB produces "poorly optimised queries"

MetaHDBC supports all features of a giving RDMS, as queries are evaluated unaltered (except for bound variables, see section 4.2). HaskellDB is restricted as it neither include any RDMS specific features, nor do it support all of the standard SQL features. The latter is unfortunately, but avoiding RDMS specific features is a two edged sword, as it do preclude the programmer from useful (RDMS specific) features, but also shields the programmer from accidently becoming dependent on one particular RDMS.

HaskellDB error messages is sometime verbose and difficult to understand[3]. In comparison MetaHDBC generally provides useful error messages (section 8), though their usefulness depends upon RDMS. However, if native extensible records are added to Haskell then HaskellDB's error messages will likely improve.

The few layers in MetaHDBC and its reliance on RDMSs and Template Haskell makes for a simple implementation. In contrast, HaskellDB needs a complicated representation of queries making extensive use of the Haskell type system, functions mimicking SQL and transformation code from its internal representation to SQL. In lack of a better measure I have looked at lines of code in HaskellDB and MetaHDBC. At the time of writing HaskellDB uses approximately 4100 lines (excluding the multitude of driver backends). MetaHDBC uses approximately 640. Both measures includes blank lines and comments.

FIXME: measures needs to be updated before publication

10 Future work

10.1 Marshalling

MetaHDBC uses HDBC to marshall and unmarshall database attributes to and from Haskell data types. When marshalling rows we have used Haskell's build-in tuples, which has worked for our examples, as the tables only had a few attributes. However, tables often have many attributes and using 10-tuples or 20-tuples will become unwieldy.

I could extend MetaHDBC to handles tables with many columns by using Haskell's user-defined data types. For each database table MetaHDBC could create a matching Haskell data type, including functions to insert, update, delete and select rows. All using the primary key (and possibly other keys) to select which row to manipulate. To avoid unnecessary boilerplate MetaHDBC should use Template Haskell to create the functions and the Haskell data type.

10.2 Type inference

It was disappointing to discover the lacks in type inference for popular RDMSs (section 6). Two options to remedy this would be:

- Extending RDMSs to do type inference. MySQL and PostgreSQL both has poor type inference, but both are open source and I could therefore extend them with type inference.
- Implement my own SQL parser as part of MetaHDBC. It could do type inference at compile-time, but let the RDMS handle the SQL at run-time.

The first option would not only benefit MetaHDBC, but also other users of a type inference extended RDMS. The second option would only benefit MetaHDBC, but would work for all RDMSs.

With the second option, it would be difficult to support both type inference and RDMS specific features. This limitation is not a problem for the first option.

10.3 Preparing statements for multiple RDMSs

As MetaHDBC uses the SQL directly it also gets dependent upon the dialect of SQL a particular RDMS vendor is using. But MetaHDBC could become less dependent by letting two or more RDMSs parse the same SQL statement and:

- See that the SQL was syntactically and semantically valid on all used RDMSs
- See that all used RDMSs expect the same number of parameters and will produce rows containing the same number of results
- Either see that all RDMSs expect the same type of parameters and results or MetaHDBC could use the most precise type inference and thus alleviate the problem with incomplete type inference described in section 6.

This type of cross-platformness is different from what HaskellDB offers. Where HaskellDB tries to provide general cross platform behaviour by using an intermediate layer, here MetaHDBC would check that a specific SQL statement behave the same on multiple specific RDMSs.

11 Conclusion

I did show how to build a database access library, which is simple, extensible, provides static implicit typing (to some degree), has readable compile-time error messages and exposes all features a RDMS (Relational Database Management System) has to offer.

MetaHDBC's ability to do static implicit typing is only as good as the underlying RDMSs ability to infer types. Unfortunately, three of the explored RDMSs did a poor job of type inference and the rest (**FIXME: or one if we can never test on SQL Server**) was good, but not perfect. MetaHDBC do provide a variety of features, but as static implicit typing is the most important one, I were disappointed with RDMSs ability to do type inference.

12 Acknowledgements

References

[1] *Report on the programming language haskell 98*, Tech. report, Yale University, CS Dept., feb 1999.

[2] Björn Bringert, *Haskell-cafe mailing list post by björn bringert*, `http://www.mail-archive.com/haskell-cafe@haskell.org/msg41029.html`.

[3] Björn Bringert, Anders Höckersten, Conny Andersson, Martin Andersson, Mary Bergman, Victor Blomqvist, and Torbjörn Martin, *Student paper: Haskelldb improved*, Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell (New York, NY, USA), ACM Press, 2004, pp. 108–115.

[4] Paul Hudak, *Modular domain specific languages and tools*, in Proceedings of Fifth International Conference on Software Reuse, IEEE Computer Society Press, 1998, pp. 134–142.

[5] ISO/IEC 9075-10:2003, *Part 10: Information technology – database languages – sql – part 10: Object language bindings (sql/olb)*, ISO, Geneva, Switzerland.

[6] ISO/IEC 9075-2:2003, *Information technology – database languages – sql – part 2: Foundation (sql/foundation)*, ISO, Geneva, Switzerland.

[7] Daan Leijen and Erik Meijer, *Domain specific embedded compilers*, 2nd USENIX Conference on Domain Specific Languages (DSL'99) (Austin, Texas), October 1999, Also appeared in ACM SIGPLAN Notices 35, 1, (Jan. 2000), pp. 109–122.

[8] IBM Redbooks, *Db2 for z/os and os/390: Ready for java*, ch. 8 - Getting started with SQLJ, pp. 141–148, 2003.

[9] Tim Sheard and Simon Peyton Jones, *Template metaprogramming for Haskell*, ACM SIGPLAN Haskell Workshop 02 (Manuel M. T. Chakravarty, ed.), ACM Press, October 2002, pp. 1–16.