HaskellDB

Type-safe declarative database combinators

Björn Bringert
bringert@cs.chalmers.se

Chalmers University of Technology

Current approach

Most database interfaces use SQL strings embedded in the program:

```
$res = mysql_query("select articlekey, title "
     . "from users, articles"
     . "where users.uid = articles.uid "
     . "and username = '" . $name . "'")
     or die("Invalid query: " . mysql_error());
while ($row = mysql_fetch_array($res)) {
     printf("%s, %s", $row['articlekey'],
                      $row['username']);
}
mysql_free_result($res);
```

Can you find the bugs?

Current approach

Most database interfaces use SQL strings embedded in the program:

```
$res = mysql_query("select articlekey, title "
     . "from users, atricles"
     . "where users.uid = articles.uid "
     . "and username = '" . $name . "'")
     or die("Invalid query: " . mysql_error());
while ($row = mysql_fetch_array($res)) {
     printf("%s, %s", $row['articlekey'],
                      $row['username']);
}
mysql_free_result($res);
```

Problems with SQL strings

Some problems with using embedded SQL strings:

- Invalid queries are not detected until runtime. Potential problems include syntax errors, type errors, non-existing tables and fields.
- Types of query result rows are not statically known. The compiler doesn't know which fields we get, nor their types.
- Unescaped input causes security problems.
  What if $name = “foo'; drop table users”?

The somewhat less nasty way

The same fragment (less the bugs) using HaskellDB:

```
do
q = do u <- table users
       a <- table articles
       restrict (u!U.uid .==. a!A.uid .&&.
                 username .==. constant name)
       project (articlekey << a!articlekey
                # title << a!title)
mapM_ (putStrLn . showRow) (query db q)

showRow r = r!articlekey ++ ", " ++ r!title
```

HaskellDB

Using HaskellDB, all of the bugs in the PHP/SQL example are found by the compiler.

HaskellDB is a domain specific embedded language for specifying database queries and operations. It was created by Daan Leijen and Erik Meijer.

As a student project at Chalmers, we have modified HaskellDB to work with current compilers and more platforms. Changes include a new record type, reworked back-end interface, new back-ends, table creation support, new query operators, bounded string support, API documentation, bug fixes etc.

Haskell crash course

Types:

```
replicate :: Int -> a -> [a]
```

any type

list

Application:

```
replicate 5 'x'  ⇒  “xxxxx”
```

I/O actions:

```
putStrLn :: String -> IO ()
```

I/O action

“nothing”

do-notation:

```
do
x <- readLine
let y = map toUpper x
putStrLn y
```

apply a function to each element of a list

HaskellDB basics

Record construction:

(field1 << expr1 # field2 << expr2)

Field selection:

r!field1

Queries are constructed in the Query monad. The monad is used to hold the currently constructed query and supply unique names for field renaming.

Running queries (somewhat simplified):

query :: Database -> Query (Rel r) -> IO [r]

Database description

To assign types to queries, we need a database description:

- A Haskell module declaring all the tables and fields.
- Can be hand-written or generated from the database.
- Can be used to create a new database.

Relations

In a database context, a relation is essentially a table with labeled columns.

Two of the tables from the WASH Blogger:

topic	*owner*
wifi	peteg
estonian	bjorn

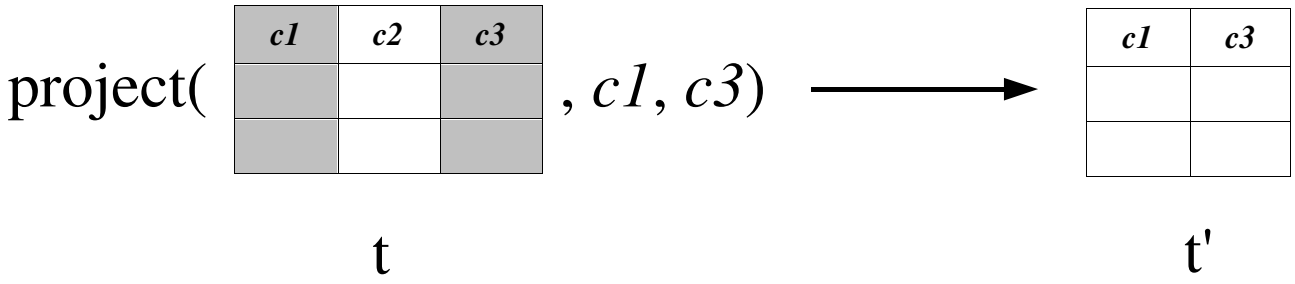
The *topics* table

topic	*date*	*owner*	*body*
estonian	04/08/18	bjorn	üks, kaks, kolm
wifi	04/08/20	peteg	Free wifi rocks!
estonian	04/08/20	bjorn	I still don't understand a word.

The *messages* table

Projection

Projection: select a subset of the *columns* in a relation.

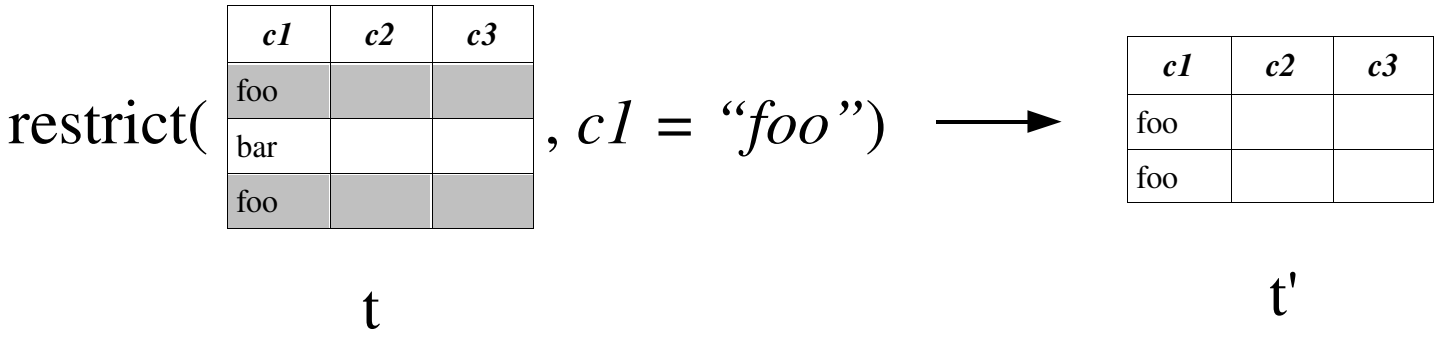


In HaskellDB, projection and renaming is done with the *project* function, e.g.:

```
t' <- project (c1 << t!c1 # c2 << t!c2)
```

Restriction

Restriction (or selection): select a subset of the *rows* in a relation.



In HaskellDB, restriction is done using the *restrict* function, e.g.:

restrict (t!c1 .==. constant "foo")

Note that we don't say which relation to restrict since the Query monad keeps track of the current relation.

Multiplication

Multiplication: given two relations, create a relation with all columns from both relations and all pairwise combinations of values.

multiply(

c1	*c2*
A	B
C	D

t1

,

c3	*c4*
E	F
G	H

t2

) →

c1	*c2*	*c3*	*c4*
A	B	E	F
A	B	G	H
C	D	E	F
C	D	G	H

t3

Multiplication in HaskellDB is somewhat different:

```
r1 <- table t1
r2 <- table t2
r3 <- project (c1 << r1!c1 # c2 << r1!c2
               # c3 << r2!c3 # c4 << r2!c4)
```

← implicit multiplication

Constructing a query

```
-- get topic name and date of last message,
-- topic with newest message first
recentTopics :: Database
             -> IO [(String,CalendarTime)]
```

We want to construct this query, run it and get the results in the right format.

Constructing a query

```
-- get topic name and date of last message,
-- topic with newest message first
recentTopics :: Database
             -> IO [(String,CalendarTime)]
recentTopics db = do
  let q = do
          t <- table topics
          m <- table messages
```

multiplies the *topics* and *messages* tables

Constructing a query

```
-- get topic name and date of last message,
-- topic with newest message first
recentTopics :: Database
             -> IO [(String,CalendarTime)]
recentTopics db = do
  let q = do
          t <- table topics
          m <- table messages
          restrict (t!T.topic .==. m!M.topic)
```

equality in query expressions

We join the tables on the *topic* fields, i.e. multiply them and keep only the rows where the *topic* fields have the same value.

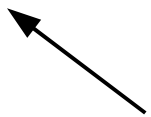
Constructing a query

```
-- get topic name and date of last message,
-- topic with newest message first
recentTopics :: Database
             -> IO [(String,CalendarTime)]
recentTopics db = do
  let q = do
          t <- table topics
          m <- table messages
          restrict (t!T.topic .==. m!M.topic)
          r <- project (T.topic  << t!T.topic
                        # M.date << _max(m!M.date))
```

We get the topic name and the maximum date for every topic.

Constructing a query

```
-- get topic name and date of last message,
-- topic with newest message first
recentTopics :: Database
             -> IO [(String,CalendarTime)]
recentTopics db = do
  let q = do
          t <- table topics
          m <- table messages
          restrict (t!T.topic .==. m!M.topic)
          r <- project (T.topic  << t!T.topic
                        # M.date << _max(m!M.date))
          order [desc r M.date]
          return r
```



descending order by the *date* field

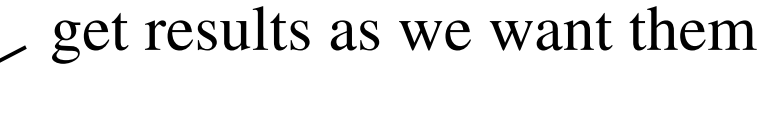
Constructing a query

```
-- get topic name and date of last message,
-- topic with newest message first
recentTopics :: Database
             -> IO [(String,CalendarTime)]
recentTopics db = do
  let q = do
          t <- table topics
          m <- table messages
          restrict (t!T.topic .==. m!M.topic)
          r <- project (T.topic  << t!T.topic
                        # M.date << _max(m!M.date))
          order [desc r M.date]
          return r
  rs <- query db q
```

run the query

Constructing a query

```
-- get topic name and date of last message,
-- topic with newest message first
recentTopics :: Database
             -> IO [(String,CalendarTime)]
recentTopics db = do
  let q = do
          t <- table topics
          m <- table messages
          restrict (t!T.topic .==. m!M.topic)
          r <- project (T.topic  << t!T.topic
                        # M.date << _max(m!M.date))
          order [desc r M.date]
          return r
  rs <- query db q
  return $ map (\r -> (r!T.topic, r!M.date)) rs
```



Insert & Delete

Insert a row:

```
insert :: Database -> Table r -> Record r -> IO ()

insert db topics (topic <<- “lecture”
                  # owner <<- “nibro”)
```

Delete rows:

```
delete :: Database -> Table r
        -> (Rel r -> Expr Bool) -> IO ()


delete db topics (\r -> r!owner .==.
                        constant “peteg”)
```

Update

Updating rows:

```
update :: Database -> Table r
        -> (Rel r -> Expr Bool)
        -> (Rel r -> Record s) -> IO ()
```

```
update db topics
   (\r -> r!topic `like` constant “foo”)
   (\r -> topic <<- “bar” # owner << r!owner)
```

Transactions

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

- Performs some I/O action as a transaction on a database.
- I/O action may include database operations and any other I/O.
- Does rollback automatically if an exception is thrown.

Extensible records

HaskellDB uses record types for relations.

Typing arbitrary database queries requires that the type system supports constructing new record types without declaring them.

To do this, the original HaskellDB used the *Trex* extension, which is only available in Hugs.

We have constructed a system for extensible records using more common extensions: multi-parameter type classes and overlapping instances.

Problems / future work

- Record type errors are hard to understand.
- All field labels must be declared.
- Mutually incompatible back-ends.
- No support for database-specific features,
- Does not use all optimization features (e.g. prepared statements).
- Might be too SQL-specific.

Conclusions

- Quick development.
- Query correctness.
- Security.
- Can use the power of Haskell.
- Platform independence.
- Still has some rough edges.

Further information

http://haskelldb.sourceforge.net/

New developers are very welcome!