SQLHS documentation

Konstantin Pugachev

The Logical Structure

`SQLHS` modules family provide base SQL operations using Haskell syntax. The tables have operation counter and support laziness if possible.

`CountableTable` module introduces tables that store their computation complexity. The table is its constant complexity and a list of rows. The row consists of a computation complexity value and a list of its cells. The cell is a variant type value. It could be either a wrapped String, Int or Double. As a hack, it could be also a fake value or so-called descending value. The first one is used to expess complexity of figuring out that all the rows have been read (like you have to process 90 more lines when evaluating `take 11 . filter (<=10) $ [1..100]` instead of `take 10 . filter (<=10) $ [1..100]` and you have to store this "90" somewhere), is to be replaced by a better solution in the future. The latter is used for sorting in the inversed order. A table complexity is its complexity plus the sum of its rows complexities including rows storing fake values.

For ones who like monads, it would be happy to know that tables and rows are monads (however, their conforming the monad laws haven't been proved yet), having counting complexity as their side effect (see `count` function).

`SQLHS` provides base SQL operations as Haskell-like functions. `SQLHSSugar` wraps them in order to achieve SQL-like syntax making use of Haskell operators and their priorities supporting text names like `mytable.mycolumn`. You can drop `SQLHSSugar` if you think your abstraction level is too high to increase it even more.

Finally, you are provided `DBReader` that reads the course database as a 4-tuple of tables described above and `SQLHSExample` containing some operative code to start with.

Setting up the Environment

The modules described depend on some third party packages. Everything seems to be OK when using the NSU PCs unless you forget to update cabal.

```
cabal update
cabal install sqlite-simple
cabal install multimap
```

Composing Queries

You appear to know Haskell too well to find out how to use the majority of modules in question. So let me show you just the top of the code iceberg, the sugared version.

Given tables $a = (a_1, a_2)$ and $b = (b_1, b_2)$ having rows numbers A and B respectively, `SQLHSSugar` provides

- several ways to join tables
  - ``a `cjoin` b`` – cross join aka comma (lazy, up to $O(AB)$)
  - ``a `njoin` b `on` "a1" `jeq` "b2"`` – naive join aka cross join with filtering (lazy, up to $O(AB)$)
  - ``a `hjoin` b `on` col "a1"`` – hash join (lazy, up to $O(A\ log(B))$) where b is an indexed table
  - ``a `mjoin` b `on` "a1" `jeq` "b2"`` – merge join (lazy, up to $O(A + B)$) where a and b are sorted ones (note you can pass an unsorted table to `mjoin` or join on wrong column that could be erroneous)

- indexing tables
  - `` b `indexby` col "b2" `` – indexing a table (strict, zero cost[1])
  - `flatten b` – getting a sorted table from indexed one (strict, zero cost)
- sorting tables
  - `` a `orderby` ["a1":asc, "a2":desc] `` – ordering a table by the columns specified (strict, $O(A\ log(A))$)
- selecting/projecting tables
  - `` a `select` ["a1", "a2"] `` – select the columns specified (lazy, up to $O(A)$)
  - `` a `wher` col "a1" `eq` str "Blackbriar" `` – select the rows specified (lazy, up to $O(A)$) – you can use things like `int 100`, `double 3.14` or `col 8` as `eq` operands
  - `distinct a` – selecting unique rows (lazy, up to $O(A^2)$)
  - `limit m n a` – skip m rows and select n ones (lazy, up to $O(n)$)
  - `enumerate a` – zip the natural numbers list with the table rows (lazy, up to $O(A)$)
- accessing fields by name and renaming tables
  - `a // "aaa"` – let a be called aaa so one can use `"aaa.a1"`, `"aaa.a2"` field names
  - `"a"`, `"a.a1"` – access column a_2 (e.g. `` a // "a" `njoin` b // "b" `on` "a.a2" `jeq` "b1" ``)

`SQLHSSugar` exports the pipe operator (`Data.Function.&`) so the latter ones could be rewritten as `a & limit m n` and `a & enumerate` respectively.

In theory, every relation operation yields a relation. So does the module except for `indexby` that yields an index. You can combine as complex query as your computer can handle and you can wait for. Note that strict operations break laziness of their operands, the lazy ones don't.

The sugared version isn't supposed to be painful, however it could be a little bit. `njoin` and `mjoin` get columns numbers of the tables provided **in the same order you provide the tables** (logical, huh?). `hjoin` expects you to pass it the column name of the unindexed table, the indexed one using the column it was indexed by (a bit unclear but still logical, isn't it?).

Given tables $a = (a_1, a_2)$ and $b = (b_1, b_2)$, `a JOIN b ON a2=b1` could be either

- `` a `njoin` b `on` "a2" `jeq` "b1" ``
- `` a `hjoin` (b `indexby` col "b1") `on` col "a2" ``
- `` (a `orderby` ["a2":asc]) `mjoin` (b `orderby` ["b1":asc]) `on` "a2" `jeq` "b1" ``
- `` a `cjoin` b `wher` col "a2" `eq` col "b1" ``

Call `test "title to show" a` to print your resulting table `a`. The output is the title provided, up to ten first table rows, the rows count and the complexity of evaluating the table.

See the examples in `SQLHSExample.hs`

Compiling and Executing

There are two ways we know to run your code. You can compile it and run:

```
ghc -O2 -outputdir bin SQLHSExample
SQLHSExample
```

Although compiled code is faster and the compilation is incremental, you could be a relational jedi creating extremely fast query plans. Interpret your code if it's true for you:

```
runhaskell SQLHSExample
```

[1]It definitely takes some time to compute. But indexing is supposed to be performed before running queries so its complexity does not count. The lecturer would give you some penalty points for indexing intermediate tables.

Please make sure you have the appropriate version of `wares_???.sqlite3` in the current directory and it has `wares.sqlite3` name.

So, we wish you fast queries![2]

[2]Compiled using Pandoc: `pandoc readme.md -V geometry:margin=2cm -o readme.pdf`