DBConnect

Simple Accessing Postgresql Databases from haskell

Author Stefan Lack
Freiburg, April 2004

Inhaltsverzeichnis

1 Einleitung

`DBConnect` ist eine kleine Sammlung von Funktionen mit denen es ermöglicht wird, von Haskell aus auf eine *Postgresql* Datenbank zuzugreifen. `DBConnect` verwendet dazu die libpg-Bibliothek.

1.1 Vorraussetzungen

Um BDConnect verwenden zu können, müssen einige andere Packete auf dem System installiert sein, die im Folgenden kurz genannt werden:

Die libpq Bibliothek auf die zugegriffen wird. Die libpq ist in C geschrieben. Zusätzlich zu der Bibliotek werden auch die Headerdateien benötigt. Für debian lautet das Packet z.B. *postgresql-dev*

C->Haskell Compiler , ein Interface Generator mit welchem auf die C-Bibliotheken zugegriffen wird, erhältlich unter *http://www.cse.unsw.edu.au/∼ chak/haskell/c2hs/*

haddock *(optional)* ein Werkeug, welches aus kommentierten HaskellSourcen eine übersichtliche, html-formatierte api erstellt. (Ähnlich zu *javadoc*) Download: http://www.haskell.org/haddock/

haskell Verwendet wird der Glasgow Haskell Compiler ghc, Version 6.

2 Installation von DBConnect

Zur Installation entpacke man die tar Datei `dbconnect.tgz`

```
user@leo $ tar xfz dbconnect.tgz
user@leo $ cd dbconnect/lib
```

Nach dem entpacken prüfe man, ob die beiden Pfade `PGLIB` und `PGINCLUDE` im Makefile im Verzeichnis `dbconnect/lib` richtig gesetzt sind. Im Makefile werden diese Pfade mitels dem Postgresql-Programm `pg_config` gesetzt.
Nun können wir mit dem Compilieren der Bibliothek beginnen:

```
user@leo ~$ cd dbonnect/lib
user@leo ~/dbconnect/lib$ make depend
user@leo ~/dbconnect/lib$ make dblib
user@leo ~/dbconnect/lib$ make doc
```

Damit werden zuerst die Abhängigkeiten analysiert und am Ende des Makefiles eingefügt. `make dblib` startet dann die Übersetzung und erzeugt `libDBCONNECT.a`. `make doc` ruft haddock auf.
Wenn `make doc` folgende Fehlermeldung ausgibt:

```
Fail: does not exist
Action: openFile
Reason: No such file or directory
File: haddock/index.html

make: *** [doc] Fehler 1
```

muss noch ein neues Verzeichniss für die Dokumentation erstellt werden:

```
user@leo ~/dbconnect/lib$$ mkdir haddock
```

3 Installation des Beispiels

Das Verzeichnis `dbconnect/examples` enthält eine kleine WASH-Anwendung, welche auf eine Datenbank namens `ducktales` zugreift. Um das Beispiel anwenden zu können, müssen wir zuerst mittels

```
user@leo $$ createdb ducktales
CREATE DATABASE
```

eine Datenbank anlegen. Danach füllen wir sie mit einigen Beispieldatensätzen:

```
user@leo $ cd dbconnect/examples
user@leo $ psql -d ducktales -f ducktales.sql
psql:ducktales.sql:10: HINWEIS:  CREATE TABLE / PRIMARY KEY erstellt
      implizit einen Index >>persons_pkey<< für Tabelle >>persons<<
CREATE TABLE
```

```
INSERT 17854 1
INSERT 17855 1
INSERT 17856 1
INSERT 17857 1
```

Dabei bedeutet `-d` dass mit der Datenbank *ducktales* verbunden werden soll, `-f` übergibt die Datei ducktales.sql, Dadurch wird eine Relation *persons* angelegt und einige Daten eingefügt.
Nun müssen im Makefile noch einige Pfade gesetzt werden, darunter `PATH_TO_DBCONNECT`, `PATH_TO_WASH*`, `WASHROOT` und `CGIDIR`
Danach starten wir make:

```
user@leo ~/dbconnect/exsamples$ make depend
user@leo ~/dbconnect/exsamples$ make install
user@leo ~/dbconnect/exsamples$ make doc
```

Damit wird das erstellte CGI in das bei *CGIDIR* angegebene Verzeichnis installiert.

4 Beschreibung der Bibliothek

4.1 Verbing einrichten

Anfangs muss man zuerst eine Verbindung zu einer Datenbank öffnen. Dazu verwendet man die Funktion `createDBService`. Sie verlangt 5 Paramter:
`databaseserver databasename user password logfile`
Beispiel (`ducktales`) Die Verbindung in `Main.hs` wird wie folgt geöffnet:

```
dbservice = createDBService "127.0.0.1" "ducktales" "sl" "" Nothing
```

Der letze Parameter ist entweder ‘‘`Just Filename`’’ oder ‘‘`Nothing`’’.Im Falle von Nothing wird defaultmässig /tmp/dbconnect.log gewählt.
Von `createDBService` bekommen wir eine Instanz des Typs `DBService` zurück. Diese Instanz repräsentiert unsere Datenbankverbindung.

4.2 Select-Anfragen

Für SELECT-Queries stehen die Funktionen

`selectReturnTuples, selectReturnRow, selectReturnList, und selectReturnOne`

zur Verfügung. Als erstes Element muss jeweils die Datenbankinstanz von DB-Service übergeben werden, und das zweite Argument enthält das Query.
Beispiel:

```
persons <- io $ selectReturnTuples
    db
    "SELECT anrede, given_name, sur_name, email FROM persons"
```

Die Funktion `io` ist WASH-Spezifisch und kapselt die Datenbankoperation `selectReturnTuples` vom Typ `IO (String)` in den WASH Typen `CGI()`. Wann immer man eine IO Operation innerhalb der CGI-Monade ausführen will, muss man sie mit diesem Wrapper kappseln.

4.3 Insert, Delete und Update

Zur Übergabe von Insert-, Delete- und Update - Queries wird die Funktion `execute` verwendet. Sie gibt die Anzahl der betroffenen Zeilen als `Int` zurück. Aufgerufen wird sie ebenso wie die Select-Funktionen: Der erste Parameter ist die Instanz von `DBService`, der zweite ist das Query.

```
-- | performs the deleting of a person tuple
delPerson ::String -> String -> DBService -> CGI()
delPerson vorname nachname db = do
  io $ execute db $ "DELETE FROM persons WHERE given_name='"++vorname++
                    "' AND sur_name='"++nachname++"'"
  sampleSelect db
```

In diesem Beispiel interessieren wir uns nicht weiter für die Anzahl der gelöschten Tupel, daher wurde das Ergebnis ignoriert.

4.4 Fehlerbehandlung

Innerhalb aller Datenbankfunktionen können Exceptions auftreten. Um die Fehler abzufangen, gibt es die Funktion `catchDB`.

Beispiel:

```
main :: IO ()
main =
  let
    dbservice = createDBService "127.0.0.1" "ducktales" "sl" "" Nothing
  in
   catch
     (catchDB (run $ sampleSelect dbservice)
              (processException))
     processAllException
```

Das erste Argument ist die auszuführende Funktion vom Typ `IO()`.

Das zweite Argument ist eine Funktion vom Typ srcc DBEXception -> IO (). Diese Funktion wird von catchDB ausgewertet, sobald eine entsprechende Exception aufgerufen wurde. Sie dient dazu, auf geeignete Weise auf einen Fehler zu reagieren.
Das Beispiel `processException` zeigt, wie man die Exception weiter auswertet: Eine DBException kann entweder einen `ConnectionError` oder einen `ExecuteError` bedeuteten. In beiden Fällen wird eine Zeichenkette mitgeliefert, in welcher der Fehler näher erleutert wird.

```
processException :: DBException  -> IO ()
processException e = case e of
  DBError  ConnectionError s ->
       itell stdout $ standardCssPage "Datenbankfehler"
       (do (text s)
                   br empty
                   textBackLink "Zurück" empty
                   br empty
                   (hlink (URL "javascript:location.reload()")
                          (text "Erneut versuchen")))
                mystyles
  DBError ExecuteError s ->
       itell stdout $ standardCssPage "Datenbankfehler"
       (do (text $ "Fehler beim Ausführen des Querys: "++ s)
```

```
    br empty
    textBackLink "Zurück" empty)
  mystyles
```

Kommt es zum Beispiel durch die Ausführung eines Queries zu einer Primary-Key Verletzung, wird das Beispiel-cgi diese Fehlermeldung generieren:

```
Fehler beim Ausführen des Querys:
FEHLER: duplizierter Schlüssel verletzt Unique-Constraint
 >>persons_pkey<<
```

Kann das CGI nicht auf die Datenbank zugreifen, gibt es einen `ConnectionError`:

```
Error: unable to open connection to server: 127.0.0.1 database: ducktales
Zurück
Erneut versuchen
```