Programowanie funkcyjne

Bazy danych – Haskell

Autorzy:
Budziło Maria,
Kubański Marcin,
Stypuła Hiacynta

18.01.2022

1. Wstęp

Wszystko, od forów internetowych do podcatcherów (oprogramowanie przeznaczone do „łapania" podcastów), a nawet programów do tworzenia kopii zapasowych, często wykorzystuje bazy danych do trwałego przechowywania. Bazy danych oparte na SQL są często dość wygodne: są szybkie, mogą skalować się od niewielkich do ogromnych rozmiarów, mogą działać w sieci, często pomagają w obsłudze blokowania i transakcji, a nawet mogą zapewniać usprawnienia w zakresie przełączania awaryjnego i nadmiarowości aplikacji. Bazy danych przybierają różne kształty: duże komercyjne bazy danych, takie jak Oracle, silniki typu open source, takie jak PostgreSQL lub MySQL, a nawet silniki osadzane, takie jak Sqlite. Ponieważ bazy danych są tak ważne, wsparcie Haskella dla nich jest również ważne. Przedstawimy teraz jeden z frameworków Haskella do pracy z bazami danych.

2. Przegląd HDBC *(Haskell Database Connectivity)*

Na dole stosu bazy danych znajduje się silnik bazy danych, który odpowiada za faktyczne przechowywanie danych na dysku. Dobrze znane silniki baz danych to PostgreSQL, MySQL i Oracle.

Większość nowoczesnych aparatów baz danych obsługuje **SQL** *(Structured Query Language)* jako standardowy sposób pobierania danych do i z relacyjnych baz danych.

Gdy ma się już silnik bazy danych, który obsługuje SQL, potrzebujemy sposobu na komunikację z nim. Każda baza danych ma swój własny protokół. Ponieważ SQL jest dość stały we wszystkich bazach danych, możliwe jest stworzenie ogólnego interfejsu, który używa sterowników dla każdego indywidualnego protokołu.

Haskell udostępnia kilka różnych frameworków bazodanowych, z których niektóre zapewniają warstwy wysokiego poziomu, a inne. Omówiony zostanie system **HDBC** *(Haskell DataBase Connectivity)*. HDBC to biblioteka abstrakcji bazy danych. Oznacza to, że można napisać kod, który korzysta z HDBC i może uzyskać dostęp do danych przechowywanych w prawie każdej bazie danych SQL z niewielkimi modyfikacjami lub bez nich. Nawet jeśli nie występuje potrzeba przełączania podstawowych silników baz danych, system sterowników HDBC daje dostęp do wielu opcji do ciebie za pomocą jednego interfejsu.

Inną abstrakcyjną biblioteką baz danych dla Haskella jest HSQL, który ma podobne przeznaczenie jak HDBC. Istnieje również struktura wyższego poziomu o nazwie HaskellDB, która znajduje się na szczycie HDBC lub HSQL i ma na celu odizolowanie programisty od szczegółów pracy z SQL. Jednak nie ma tak szerokiego zastosowania, ponieważ jego konstrukcja ogranicza ją do pewnych — choć dość powszechnych — wzorców dostępu do baz danych. Wreszcie, Takusen to framework, który wykorzystuje podejście „w lewo" do

odczytywania danych z bazy danych.

3. Instalacja HDBC oraz Sterowników

Aby połączyć się z daną bazą danych za pomocą HDBC, potrzeba co najmniej dwóch pakietów: ogólnego interfejsu i sterownika dla konkretnej bazy danych. Ogólny pakiet HDBC i wszystkie inne sterowniki można uzyskać z witryny Hackage (`http://hackage.haskell.org/`).

Będzie także potrzebny backend bazy danych i sterownik backendu. Dlatego też używa się Sqlite w wersji 3. **Sqlite** jest wbudowaną bazą danych, więc nie wymaga osobnego serwera i jest łatwa w konfiguracji. Wiele systemów operacyjnych jest już dostarczanych z Sqlite w wersji 3. Można go pobrać z `http://www.sqlite.org/`. Strona główna HDBC zawiera łącze do znanych sterowników zaplecza HDBC. Konkretny sterownik dla Sqlite w wersji 3 można uzyskać z Hackage.

Jeśli chce się używać HDBC z innymi bazami danych, sprawdzamy stronę znanych sterowników HDBC pod adresem `http://software.complete.org/hdbc/wiki/KnownDrivers`. Znajdziemy tam link do powiązania ODBC, który pozwala na połączenie z praktycznie każdą bazą danych na dowolnej platformie (Windows, POSIX i inne). Znajdziemy również powiązanie PostgreSQL. MySQL jest obsługiwany przez powiązanie ODBC, a szczegółowe informacje dla użytkowników MySQL można znaleźć w dokumentacji API HDBC-ODBC (`http://software.complete.org/static/hdbc-odbc/doc/HDBC-odbc/`).

4. Podłączanie się do baz danych

Aby połączyć się z bazą danych, używamy funkcji połączenia ze sterownika backendowego bazy danych. Każda baza danych ma swoją unikalną metodę łączenia. Początkowe połączenie jest zazwyczaj jedynym momentem, w którym wywołujemy cokolwiek bezpośrednio z modułu sterownika backendowego.

Funkcja połączenia z bazą danych zwróci identyfikator bazy danych. Dokładny typ tego identyfikatora może się różnić w zależności od sterownika, ale zawsze będzie to instancja typeklasy IConnection. Wszystkie funkcje, których użyjesz do operowania na bazach danych, będą działać z dowolnym typem, który jest instancją IConnection. Kiedy skończymy, wywołujemy funkcję rozłączenia, aby się z nią rozłączyć. Oto przykład nawiązania połączenia z bazą danych Sqlite:

```
marcinstudia@MarcinPC:~/Desktop$ ghci
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> :module Database.HDBC Database.HDBC.Sqlite3
Prelude Database.HDBC Database.HDBC.Sqlite3> conn <- connectSqlite3 "test1.db"
Prelude Database.HDBC Database.HDBC.Sqlite3> :type conn
conn :: Connection
Prelude Database.HDBC Database.HDBC.Sqlite3> disconnect conn
Prelude Database.HDBC Database.HDBC.Sqlite3>
```

Rysunek 1: Przykład nawiązania połączenia z bazą danych.

5. Transakcje

Większość nowoczesnych baz danych SQL zawiera pojęcie transakcji. Transakcja ma na celu zapewnienie, że wszystkie składniki modyfikacji zostaną zastosowane lub że żaden z nich nie zostanie zastosowany. Ponadto transakcje pomagają zapobiegać wyświetlaniu części danych pochodzących z trwających modyfikacji innym procesom uzyskującym dostęp do tej samej bazy danych.

Wiele baz danych wymaga jawnego zatwierdzenia wszystkich zmian, zanim pojawią się na dysku, lub uruchomienia w trybie automatycznego zatwierdzania. Tryb automatycznego zatwierdzania uruchamia niejawne zatwierdzenie po każdej instrukcji. Może to ułatwić dostosowanie do transakcyjnych baz danych programistom do nich nieprzyzwyczajonym, ale jest to tylko utrudnienie dla osób, które faktycznie chcą korzystać z transakcji wielostanowiskowych.
HDBC celowo nie obsługuje trybu automatycznego zatwierdzania. Kiedy modyfikujesz dane w swoich bazach danych, musisz jawnie spowodować ich zatwierdzenie na dysku. W HDBC można to zrobić na dwa sposoby: możemy wywołać zatwierdzenie, gdy jesteśmy gotowi do zapisania danych na dysku, lub możemy użyć funkcji withTransaction, aby otoczyć kod modyfikacji. withTransaction spowoduje przekazanie danych po pomyślnym zakończeniu funkcji.

Czasami podczas pracy nad zapisem danych w bazie danych wystąpi problem. Przykładowo otrzymamy błąd z bazy danych lub odkryjemy problem z danymi. W takich przypadkach można „cofnąć” zmiany. Spowoduje to, że wszystkie zmiany wprowadzone od ostatniego zatwierdzenia lub wycofania zostaną zapomniane. W HDBC można w tym celu wywołać funkcję wycofania. Jeśli używamy withTransaction, każdy nieprzechwycony wyjątek spowoduje wycofanie.

6. Proste zapytania

Niektóre z najprostszych zapytań w SQL zawierają instrukcje, które nie zwracają żadnych danych. Zapytania te mogą służyć do tworzenia tabel, wstawiania danych, usuwania danych i ustawiania parametrów bazy danych.

Uruchamiana jest najbardziej podstawowa funkcja wysyłania zapytań do bazy danych. Ta funkcja pobiera IConnection, String reprezentujący samo zapytanie oraz listę parametrów. Używana jest do ustawienia niektórych rzeczy w bazie danych.

```
Prelude Database.HDBC Database.HDBC.Sqlite3> conn <- connectSqlite3 "test1.db"
Prelude Database.HDBC Database.HDBC.Sqlite3> run conn "CREATE TABLE test (id INTEGER NOT NULL, desc VARCHAR(80))" []
0
Prelude Database.HDBC Database.HDBC.Sqlite3> run conn "INSERT INTO test (id) VALUES (0)" []
1
Prelude Database.HDBC Database.HDBC.Sqlite3> commit conn
Prelude Database.HDBC Database.HDBC.Sqlite3> disconnect conn
```

Rysunek 2: Najbardziej podstawowa funkcja wysyłania zapytań do bazy danych, używana do ustawienia niektórych rzeczy w bazie danych.

W tym przykładzie po połączeniu się z bazą danych najpierw utworzono tabelę o nazwie test. Następnie wstawiono do niej jeden wiersz danych. Zatwierdzono zmiany i odłączyliśmy się od bazy danych. Zauważmy, że gdybyśmy nie wywołali zatwierdzenia, żadna ostateczna zmiana nie zostałaby w ogóle zapisana w bazie danych.

Funkcja run zwraca liczbę wierszy zmodyfikowanych przez każde zapytanie. W przypadku pierwszego zapytania, które utworzyło tabelę, nie zmodyfikowano żadnych wierszy. Drugie zapytanie wstawiło pojedynczy wiersz, więc funkcja run zwróciła 1.

7. SqlValue

Zarówno Haskell, jak i SQL są systemami silnie typowanymi, HDBC stara się zachować jak najwięcej informacji o typach. Jednocześnie typy Haskell i SQL nie odzwierciedlają się dokładnie. Co więcej, różne bazy danych mają różne sposoby przedstawiania takich rzeczy, jak daty lub znaki specjalne w ciągach.

SqlValue to typ danych, który ma wiele konstruktorów, takich jak SqlString, SqlBool, SqlNull, SqlInteger i nie tylko. Pozwala to reprezentować różne typy danych na listach argumentów w bazie danych i wyświetlać różne typy danych w przychodzących wynikach, a jednocześnie przechowywać je wszystkie na liście. Istnieją wygodne funkcje toSql i fromSql, z których będzie można normalnie korzystać.

8. Parametry Zapytania

HDBC, podobnie jak większość baz danych, obsługuje w zapytaniach pojęcie parametrów zastępowalnych. Istnieją trzy główne zalety używania parametrów zastępczych: zapobiegają atakom typu SQL lub problemom, gdy dane wejściowe zawierają znaki cudzysłowu, poprawiają wydajność przy wielokrotnym wykonywaniu podobnych zapytań oraz umożliwiają łatwe i przenośne wstawianie danych do zapytań.

Załóżmy, że chcemy dodać tysiące wierszy do naszego nowego testu tabeli. Możemy wysyłać zapytania, które wyglądają jak INSERT INTO test VALUES (0, „zero") i INSERT INTO test VALUES (1, „jeden"). Zmusza to serwer bazy danych do indywidualnej analizy każdej instrukcji SQL. Gdyby można było zastąpić te dwie wartości symbolem zastępczym, serwer mógłby raz przeanalizować zapytanie SQL i po prostu wykonać je wiele razy z różnymi danymi.

```
Prelude Database.HDBC Database.HDBC.Sqlite3> conn <- connectSqlite3 "test1.db"
Prelude Database.HDBC Database.HDBC.Sqlite3> run conn "INSERT INTO test VALUES (?, ?)" [toSql 0, toSql "zero"]
1
Prelude Database.HDBC Database.HDBC.Sqlite3> commit conn
Prelude Database.HDBC Database.HDBC.Sqlite3> disconnect conn
```

Rysunek 3: Przykład nawiązania połączenia z bazą danych.

9. Prepared Statement – sparametryzowana instrukcja

HDBC definiuje funkcję przygotowania, która przygotuje zapytanie SQL, ale nie wiąże jeszcze parametrów z zapytaniem. przygotowanie zwraca instrukcję reprezentującą skompilowane zapytanie.

Kiedy już mamy tą instrukcję, można z nią zrobić wiele rzeczy. Można wywołać na nim execute raz lub więcej razy. Po wywołaniu wykonywania zapytania, które zwraca dane, możesz użyć jednej z funkcji pobierania, aby pobrać te dane. Funkcje takie jak run i quickQuery używają instrukcji i wykonują wewnętrznie; są to po prostu skróty umożliwiające szybkie wykonywanie typowych zadań. Gdy potrzebujemy większej kontroli nad tym, co się dzieje, można użyć instrukcji zamiast funkcji, takiej jak run.

```
Prelude Database.HDBC Database.HDBC.Sqlite3> conn <- connectSqlite3 "test1.db"
Prelude Database.HDBC Database.HDBC.Sqlite3> stmt <- prepare conn "INSERT INTO test VALUES (?, ?)"
Prelude Database.HDBC Database.HDBC.Sqlite3> execute stmt [toSql 1, toSql "one"]
1
Prelude Database.HDBC Database.HDBC.Sqlite3> execute stmt [toSql 2, toSql "two"]
1
Prelude Database.HDBC Database.HDBC.Sqlite3> execute stmt [toSql 3, toSql "three"]
1
Prelude Database.HDBC Database.HDBC.Sqlite3> execute stmt [toSql 4, SqlNull]
1
Prelude Database.HDBC Database.HDBC.Sqlite3> commit conn
Prelude Database.HDBC Database.HDBC.Sqlite3> disconnect conn
```

Rysunek 4: Najbardziej podstawowa funkcja wysyłania zapytań do bazy danych, używana do ustawienia niektórych rzeczy w bazie danych.

```
Prelude Database.HDBC Database.HDBC.Sqlite3> conn <- connectSqlite3 "test1.db"
Prelude Database.HDBC Database.HDBC.Sqlite3> stmt <- prepare conn "INSERT INTO test VALUES (?, ?)"
Prelude Database.HDBC Database.HDBC.Sqlite3> executeMany stmt [[toSql 5, toSql "five's nice"], [toSql 6, SqlNull]]
Prelude Database.HDBC Database.HDBC.Sqlite3> commit conn
Prelude Database.HDBC Database.HDBC.Sqlite3> disconnect conn
```

Rysunek 5: Najbardziej podstawowa funkcja wysyłania zapytań do bazy danych, używana do ustawienia niektórych rzeczy w bazie danych.

10. Odczytywanie wyników

Do tej pory omówiliśmy zapytania, które wstawiają lub zmieniają dane. Przejdziemy teraz do pobierania danych z bazy danych. Typ funkcji quickQuery wygląda bardzo podobnie do run, ale zwraca listę wyników zamiast liczby zmienionych wierszy. quickQuery' jest zwykle używany z instrukcjami SELECT.

```
Prelude Database.HDBC Database.HDBC.Sqlite3> conn <- connectSqlite3 "test1.db"
Prelude Database.HDBC Database.HDBC.Sqlite3> quickQuery' conn "SELECT * from test where id < 2" []
[[SqlInt64 0,SqlNull],[SqlInt64 0,SqlByteString "zero"],[SqlInt64 1,SqlByteString "one"]]
```

Rysunek 6: Najbardziej podstawowa funkcja wysyłania zapytań do bazy danych, używana do ustawienia niektórych rzeczy w bazie danych.

```haskell
import Database.HDBC.Sqlite3 (connectSqlite3)
import Database.HDBC

query :: Int -> IO ()
query maxId =
    do
       conn <- connectSqlite3 "test1.db"

       r <- quickQuery' conn
            "SELECT id, desc from test where id <= ? ORDER BY id, desc"
            [toSql maxId]

       let stringRows = map convRow r

       mapM_ putStrLn stringRows

       disconnect conn

    where convRow :: [SqlValue] -> String
          convRow [sqlId, sqlDesc] =
              show intid ++ ": " ++ desc
              where intid = (fromSql sqlId)::Integer
                    desc = case fromSql sqlDesc of
                             Just x -> x
                             Nothing -> "NULL"
          convRow x = fail $ "Unexpected result: " ++ show x
```

Rysunek 7: Przykład kodu do łatwiejszego odczytywania wyników.

Następnie kompilujemy plik query.hs: Wyniki z bazy danych są teraz bardziej przejrzyste.

```
marcinstudia@MarcinPC:~/Desktop$ ghci
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> :load query.hs
[1 of 1] Compiling Main             ( query.hs, interpreted )
Ok, one module loaded.
*Main> query 2
0: NULL
0: zero
1: one
2: two
*Main>
[2]+  Stopped                 ghci
```

Rysunek 8: Wyświetlenie wyników za pomocą pliku query.hs w GHCi.

quickQuery' działa z wymiennymi parametrami, jak właśnie omówiliśmy. W tym przypadku nie używamy żadnego, więc zestaw wartości do zastąpienia to pusta lista na końcu wywołania quickQuery. quickQuery' zwraca listę wierszy, gdzie każdy wiersz jest reprezentowany jako [SqlValue]. Wartości w wierszu są wymienione w kolejności zwracanej przez bazę danych. W razie potrzeby możesz użyć fromSql, aby przekonwertować je na zwykłe typy Haskell.

11. Czytanie danych za pomocą instrukcji

Istnieje wiele sposobów odczytywania danych z wypowiedzi, które mogą być przydatne w określonych sytuacjach. Podobnie jak run, quickQuery' jest wygodną funkcją, która w rzeczywistości używa instrukcji do wykonania swojego zadania.

Aby utworzyć zestawienie do odczytu, używamy przygotowań tak samo, jak przy zestawieniu, które posłuży do zapisywania danych. Wykonujemy go również na serwerze bazy danych. Następnie możemy wykorzystać różne funkcje do odczytania danych z Wyciągu. Funkcja fetchAll Rows zwraca [[SqlValue]], podobnie jak quickQuery'. Istnieje również funkcja o nazwie sFetchAllRows', która konwertuje dane każdej kolumny na Maybe String przed ich zwróceniem. Wreszcie istnieje fetchAllRowsAL', który zwraca pary (String, SqlValue) dla każdej kolumny. String to nazwa kolumny zwrócona przez bazę danych.

12. Lazy Reading

Lazy Reading może być szczególnie przydatne w przypadku zapytań zwracających wyjątkowo dużą ilość danych. Czytając dane powoli, nadal możemy korzystać z wygodnych funkcji, takich jak ***fetchAllRows***, zamiast ręcznie odczytywać każdy wiersz, gdy się pojawia. Jeśli będziemy ostrożni w korzystaniu z danych, możemy uniknąć buforowania wszystkich wyników w pamięci.

„Leniwe" czytanie z bazy danych jest jednak bardziej złożone niż czytanie z pliku. Kiedy skończymy odczytywać dane z pliku, plik jest zamykany - co na ogół jest w porządku. Kiedy natomiast skończymy odczytywać dane z bazy danych, połączenie z bazą danych jest nadal otwarte — można na przykład przesyłać za jego pomocą inne zapytania. Niektóre bazy danych mogą nawet obsługiwać wiele jednoczesnych zapytań, więc HDBC nie może po prostu zamknąć połączenia, gdy skończymy zadanie.

W przypadku korzystania z powolnego odczytywania niezwykle ważne jest, aby zakończyć odczytywanie całego zestawu danych przed próbą zamknięcia połączenia lub wykonania nowego zapytania. Najlepiej korzystać ze ścisłych funkcji lub przetwarzania wiersz po wierszu, gdy tylko jest to możliwe, aby zminimalizować złożone interakcje z powolnym odczytaniem.

Aby powoli odczytywać z bazy danych, używamy tych samych funkcji, których używaliśmy wcześniej, bez apostrofu. Na przykład: fetchAllRows zamiast ***fetchAllRows'***. Rodzaje powolnych funkcji są takie same, jak ich surowych kuzynów. Oto przykład:

```
Prelude Database.HDBC Database.HDBC.Sqlite3> conn <- connectSqlite3 "test1.db"
Prelude Database.HDBC Database.HDBC.Sqlite3> stmt <- prepare conn "SELECT * from test where id < 2"
Prelude Database.HDBC Database.HDBC.Sqlite3> execute stmt []
0
Prelude Database.HDBC Database.HDBC.Sqlite3> results <- fetchAllRowsAL stmt
Prelude Database.HDBC Database.HDBC.Sqlite3> mapM_ print results
[("id",SqlInt64 0),("desc",SqlNull)]
[("id",SqlInt64 0),("desc",SqlByteString "zero")]
[("id",SqlInt64 1),("desc",SqlByteString "one")]
Prelude Database.HDBC Database.HDBC.Sqlite3> disconnect conn
```

Rysunek 9: Przykład Lazy Reading.

13. Metadane baz danych

Czasami może być przydatne dla programu, aby dowiedzieć się jakichś informacji o samej bazie danych. Na przykład - program może chcieć zobaczyć, jakie tabele istnieją, aby automatycznie utworzyć brakujące tabele lub zaktualizować schemat bazy danych. W niektórych przypadkach program może wymagać zmiany swojego zachowania w zależności od używanego backendu bazy danych.

Po pierwsze, istnieje funkcja getTables, która uzyskuje listę zdefiniowanych tabel w bazie danych. Można również skorzystać z funkcji defineTable, która dostarczy informacji o zdefiniowanych kolumnach w danej tabeli.

Aby dowiedzieć się czegoś o aktualnie używanym serwerze bazy danych, można wywołać na przykład dbServerVer i proxiedClientName. Za pomocą funkcji dbTransactionSupport można określić, czy dana baza danych obsługuje transakcje.

```
Prelude Database.HDBC Database.HDBC.Sqlite3> conn <- connectSqlite3 "test1.db"
Prelude Database.HDBC Database.HDBC.Sqlite3> getTables conn
["test"]
Prelude Database.HDBC Database.HDBC.Sqlite3> proxiedClientName conn
"sqlite3"
Prelude Database.HDBC Database.HDBC.Sqlite3> dbServerVer conn
"3.31.1"
Prelude Database.HDBC Database.HDBC.Sqlite3> dbTransactionSupport conn
True
```

Rysunek 10: Funkcji *dbTransactionSupport*, za pomocą której można określić, czy dana baza danych obsługuje transakcje.

Ponadto można dowiedzieć się o wynikach konkretnego zapytania, pobierając informacje z jego wypowiedzi. Funkcja **defineResult** zwraca [(String, SqlColDesc)], listę par. Pierwsza pozycja podaje nazwę kolumny, a druga informacje o kolumnie: typ, rozmiar i czy może być NULL. (Gdybyśmy czegoś potrzebowali to pełna specyfikacja jest podana w referencji API HDBC).

14. Zgłaszanie błędów

HDBC zgłosi wyjątki, gdy wystąpią błędy. Wyjątki mają typ SqlError. Przekazują informacje z bazowego silnika SQL, takie jak stan bazy danych, komunikat o błędzie i numeryczny kod błędu bazy danych, jeśli taki istnieje.

GHCi nie wie, jak wyświetlić na ekranie błąd SqlError, gdy się pojawi. Chociaż wyjątek spowoduje zakończenie programu, nie wyświetli użytecznego komunikatu.

```
Prelude Database.HDBC Database.HDBC.Sqlite3> conn <- connectSqlite3 "test1.db"
Prelude Database.HDBC Database.HDBC.Sqlite3> quickQuery' conn "SELECT * from test2" []
*** Exception: SqlError {seState = "", seNativeError = 1, seErrorMsg = "prepare 20: SELECT * from test2: no such table: test2"}
```

Rysunek 11: Informacja o błędzie w bazie danych

Próbowaliśmy poleceniem SELECT wybrać dane z tabeli, która nie istniała. Otrzymany komunikat o błędzie nie był pomocny. Istnieje funkcja narzędziowa handleSqlError, która przechwyci SqlError i ponownie wywoła go jako IOError. W tej formie będzie można go wyświetlić na ekranie, ale trudniej będzie programowo wyodrębnić określone informacje.

```
Prelude Database.HDBC Database.HDBC.Sqlite3> handleSqlError $ quickQuery' conn "SELECT * from test2" []
*** Exception: user error (SQL error: SqlError {seState = "", seNativeError = 1, seErrorMsg = "prepare 20: SELECT * from test2: no such table: test2"})
Prelude Database.HDBC Database.HDBC.Sqlite3>
```

Rysunek 12: Informacja o błędzie w bazie danych

Tutaj dostaliśmy więcej informacji, w tym komunikat, że nie ma takiej tabeli jak test2. Takie coś jest o wiele bardziej przydatne. Wielu programistów HDBC stosuje standardową praktykę, aby uruchamiać swoje programy z main = handleSqlError do, co zapewni, że każdy nieprzechwycony SqlError zostanie wyświetlony w pomocny sposób. Istnieją również catchSql i handleSql — podobne do standardowych funkcji catch i handle. catchSql i handleSql przechwycą tylko błędy HDBC.

15. Bibliografia

Materiały udostępnione w pliku „3Database_Haskell.pdf”
https://hackage.haskell.org/
http://www.sqlite.org/