Using DataBases on Haskell

Damian Łącz, Arkadiusz Jędruch
Politechnika Krakowska

February 2021

Spis treści

1 Wstęp

Wszystko, od forów internetowych po podcatchery, a nawet programy do tworzenia kopii zapasowych w większości korzysta z bazy danych do trwałego przechowywania. Bazy danych oparte na języku SQL są często dość wygodne:

- są szybkie,
- można je skalować od niewielkich do ogromnych rozmiarów,
- mogą działać w sieci,
- często obsługują blokady i transakcje pomocnicze,
- a mogą zapewnić aplikacjom poprawę w zakresie przełączania awaryjnego i nadmiarowości.

Bazy danych mają wiele różnych kształtów: duże komercyjne bazy danych, takie jak Oracle, silniki open source, takie jak PostgreSQL lub MySQL, a nawet silniki osadzalne, takie jak Sqlite. Ponieważ bazy danych są tak ważne, wsparcie Haskell dla nich jest również ważne. W tym rozdziale przedstawimy jeden z frameworków Haskell do pracy z bazami danych

2 Przegląd Haskell DataBase Con-nectivity (HDBC)

Na dole stosu bazy danych znajduje się silnik bazy danych, który jest odpowiedzialny za faktyczne przechowywanie danych na dysku. Do znanych

silników baz danych należą PostgreSQL, MySQL i Oracle. Większość nowoczesnych silników baz danych obsługuje Structured Query Language (SQL) jako standardowy sposób pobierania danych do i z relacyjnych baz danych.

Gdy mamy już silnik bazy danych, który obsługuje SQL, potrzebujesz sposobu, aby się z nim komunikować. Każda baza danych ma swój własny protokół. Ponieważ SQL jest w miarę stały w różnych bazach danych, możliwe jest stworzenie ogólnego interfejsu, który będzie używał sterowników dla każdego indywidualnego protokołu.

Haskell ma kilka różnych struktur bazodanowych, z których niektóre zapewniają wyższy poziom nad innymi. Skoncentrujemy się na systemie
Haskell DataBase Con-nectivity (HDBC)
HDBC to biblioteka abstrakcji bazy danych. Oznacza to, że możesz pisać kod wykorzystujący HDBC i mieć dostęp do danych przechowywanych w prawie każdej bazie danych SQL z niewielkimi modyfikacjami lub bez żadnych modyfikacji. pojedynczy interfejs. Inną biblioteką abstrakcji bazy danych dla Haskell jest HSQL, który ma podobny cel co HDBC. Istnieje również struktura wyższego poziomu o nazwie HaskellDB, która znajduje się na poziomie HDBC lub HSQL i została zaprojektowana w celu odizolowania programisty od szczegółów pracy z SQL. Jednak nie ma tak szerokiego odwołania, ponieważ jego konstrukcja ogranicza go do pewnych - choć dość powszechnych - wzorców dostępu do bazy danych. Wreszcie Takusen to framework, który wykorzystuje podejście „lewostronne" do odczytywania danych z bazy danych.

3 Instalowanie HDBC i sterowników

Aby połączyć się z daną bazą danych za pomocą HDBC, potrzebujemy co najmniej dwóch pakietów: interfejsu ogólnego i sterownika do konkretnej bazy danych. Ogólny pakiet HDBC i wszystkie inne sterowniki można pobrać z firmy Hackage (http://hackage.haskell.org/). W dalszej części będziemy używać HDBC w wersji 1.1.3. Będziemy także potrzebować zaplecza bazy danych i sterownika zaplecza. W tym rozdziale użyjemy SQLite w wersji 3. Sqlite to wbudowana baza danych, więc nie wymaga oddzielnego serwera i jest łatwa w konfiguracji. Wiele systemów operacyjnych jest już dostarczanych z Sqlite w wersji 3.

```
ghci> :module Database.HDBC Database.HDBC.Sqlite3
ghci> conn <- connectSqlite3 "test1.db"
Loading package array-0.1.0.0 ... linking ... done.
Loading package containers-0.1.0.2 ... linking ... done.
Loading package bytestring-0.9.0.1.1 ... linking ... done.
Loading package old-locale-1.0.0.0 ... linking ... done.
Loading package old-time-1.0.0.0 ... linking ... done.
Loading package mtl-1.1.0.1 ... linking ... done.
Loading package HDBC-1.1.4 ... linking ... done.
Loading package HDBC-sqlite3-1.1.4.0 ... linking ... done.
ghci> :type conn
conn :: Connection
ghci> disconnect conn
```

Rysunek 1

4 Połącznie z bazą danych

Aby połączyć się z bazą danych, użyjemy funkcji połączenia ze sterownika zaplecza bazy danych. Każda baza danych ma swoją własną, unikalną metodę łączenia. Pierwsze połączenie to na ogół jedyny przypadek, w którym będziemy korzystać bezpośrednio z modułu sterownika zaplecza. Funkcja połączenia z bazą danych zwróci 'uchwyt' bazy danych. Dokładny typ tego uchwytu może się różnić w zależności od sterownika, ale zawsze będzie to instancja klasy typu IConnection. Wszystkie funkcje, których będziemy używać do operacji na bazach danych, będą działać z każdym typem, który jest instancją IConnection.

Po zakończeniu pracy z bazą danych wywołaj funkcję rozłączania, aby się z nią rozłączyć.

4.1 Przykład nawiązywania połączenia z bazą danych Sqlite

5 Transakcje

Większość współczesnych baz danych SQL ma pojęcie transakcji. Transakcja ma na celu zapewnienie, że wszystkie składniki modyfikacji zostaną zastosowane lub że żaden z nich tego nie zrobi.Ponadto transakcje pomagają

uniemożliwić innym procesom dostęp do tej samej bazy danych przed wyświetleniem częściowych danych z trwających modyfikacji.

Wiele baz danych wymaga jawnego zatwierdzenia wszystkich zmian, zanim pojawią się na dysku, lub uruchomienia w trybie automatycznego zatwierdzania. Tryb autocommit uruchamia niejawne zatwierdzenie po każdej instrukcji.
Może to sprawić, że dostosowanie się do transakcyjnych baz danych będzie łatwiejsze dla nieprzyzwyczajonych do nich programistów, ale jest tylko przeszkodą 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, że zostaną zapisane na dysku.
W HDBC można to zrobić na dwa sposoby:

1. możemy wywołać commit, kiedy będziemy gotowi do zapisania danych,

2. możemy użyć funkcji withTransaction, aby otoczyć swój kod modyfikacji.

Funkcja withTransaction spowoduje, że dane zostaną zatwierdzone po pomyślnym zakończeniu funkcji.
Czasami pojawia się problem podczas pracy nad zapisem danych do bazy danych, być może otrzymamy błąd z bazy danych lub odkryjemy problem z danymi.

W tych przypadkach możemy „cofnąć" swoje zmiany. Spowoduje to, że wszystkie zmiany, które wprowadziliśmy od ostatniego zatwierdzenia lub wycofania, zostaną zapomniane.
W HDBC można w tym celu wywołać funkcję **Therollback**. Jeśli używamy funkcji **withTransaction**, każdy nieprzechwycony wyjątek spowoduje wydanie wycofania. Zauważmy, że operacja wycofywania wycofuje tylko zmiany od ostatniego zatwierdzenia, wycofania lub withTransaction. Baza danych nie posiada obszernej historii, takiej jak system kontroli wersji.

6 Proste zapytania

Niektóre z najprostszych zapytań w SQL obejmują 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

```
ghci> :module Database.HDBC Database.HDBC.Sqlite3
ghci> conn <- connectSqlite3 "test1.db"
Loading package array-0.1.0.0 ... linking ... done.
Loading package containers-0.1.0.2 ... linking ... done.
Loading package bytestring-0.9.0.1.1 ... linking ... done.
Loading package old-locale-1.0.0.0 ... linking ... done.
Loading package old-time-1.0.0.0 ... linking ... done.
Loading package mtl-1.1.0.1 ... linking ... done.
Loading package HDBC-1.1.4 ... linking ... done.
Loading package HDBC-sqlite3-1.1.4.0 ... linking ... done.
ghci> run conn "CREATE TABLE test (id INTEGER NOT NULL, desc VARCHAR(80))" []
0
ghci> run conn "INSERT INTO test (id) VALUES (0)" []
1
ghci> commit conn
ghci> disconnect conn
```

Rysunek 2

bazy danych. Ta funkcja przyjmuje IConnection, ciąg znaków reprezentujący samo zapytanie oraz listę parametrów. Użyjmy go do skonfigurowania niektórych rzeczy w naszej bazie danych

6.1 Tabela i wiersz danych w bazie

W tym przykładzie po połączeniu się z bazą danych utworzyliśmy najpierw tabelę o nazwie test, a następnie wstawiliśmy do niej jeden wiersz danych.
Ostatecznie zatwierdziliśmy zmiany i odłączyliśmy się od bazy danych.
Zauważmy, że gdybyśmy nie wywołali commit, żadna ostateczna zmiana nie została w ogóle zapisana w bazie danych.

Funkcja run zwraca liczbę wierszy modyfikowanych przez każde zapytanie. W pierwszym zapytaniu, które utworzyło tabelę, żadne wiersze nie zostały zmodyfikowane. Drugie zapytanie wstawiło pojedynczy wiersz, więc run zwrócił 1.

7 SqlValue - wartości

Zanim przejdziemy dalej, musimy omówić typ danych wprowadzony w HDBC: SqlValue.
Ponieważ zarówno Haskell, jak i SQL są systemami silnie wpisanymi na typie, HDBC stara się zachować informacje o typach w jak największym stopniu. W tym samym czasie typy Haskell i SQL nie tworzą dokładnie lustrzanego odbicia. Ponadto różne bazy danych mają różne sposoby przedstawiania takich elementów, jak daty lub znaki specjalne w łańcuchach.

SqlValue to typ danych, który ma wiele konstruktorów, takich jak

- SqlString
- SqlBool
- SqlNull
- SqlInteger
- i inne

Umożliwia to reprezentowanie różnych typów danych na listach argumentów w bazie danych i wyświetlanie różnych typów danych w powracających wynikach, a także przechowywanie ich wszystkich na liście. Istnieją wygodne funkcje, toSql i fromSql, z których zwykle będziemy mogli korzystać.

8 Parametry zapytania

HDBC, podobnie jak większość baz danych, obsługuje pojęcie parametrów wymienialnych w zapytaniach. Istnieją trzy główne zalety używania parametrów wymiennych:

1. zapobiegają atakom SQLinjection lub problemom, gdy dane wejściowe zawierają znaki cudzysłowu,
2. poprawiają wydajność podczas wielokrotnego wykonywania podobnych zapytań
3. pozwalają na łatwe i przenośne wstawianie danych do zapytań.

Załóżmy, że chcesz dodać tysiące wierszy do naszego nowego testu tabeli. Możesz zadawać zapytania, które wyglądają jak WSTAWIANIE WARTOŚCI testowych (INSERT INTO test VALUES (0, „zero")) i WSTAWIANIE WARTOŚCI testowych (INSERT INTO test VALUES (1, „jeden")). Zmusza to serwer bazy danych do oddzielnego analizowania każdej instrukcji SQL. Gdyby można było zastąpić te dwie wartości symbolem zastępczym, serwer mógłby raz sparsować zapytanie SQL i po prostu wykonać je wiele razy z różnymi danymi.

Drugi problem dotyczy ucieczki znaków. A co jeśli chcesz wstawić ciąg „I don't like 1"? SQL używa pojedynczego cudzysłowu, aby pokazać koniec pola. Większość baz danych SQL wymagałaby wpisania tego jako „I don"t like 1". Jednak zasady dotyczące innych znaków specjalnych, takich jak ukośniki odwrotne, różnią się w poszczególnych bazach danych. Zamiast próbować samemu to kodować, HDBC poradzi sobie z tym wszystkim za Ciebie.

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

Rysunek 3

8.1 Przykład - znaki specjalne

Znaki zapytania w zapytaniu INSERT w tym przykładzie to symbole zastępcze. Następnie przekazujemy parametry, które mają tam trafić. run pobiera listę SqlValue, więc my używamy toSql, aby przekonwertować każdy element na SqlValue.
HDBC automatycznie obsługuje konwersję łańcucha „zero" do odpowiedniej reprezentacji używanej bazy danych. Takie podejście nie przyniesie w rzeczywistości żadnych korzyści w zakresie wydajności podczas wstawiania dużych ilości danych.
W tym celu potrzebujemy większej kontroli nad procesem tworzenia zapytania SQL.

9 Przygotowanie sprawozdania

HDBC definiuje funkcję przygotowania, która przygotuje zapytanie SQL, ale nie wiąże jeszcze parametrów z zapytaniem. preparat zwraca instrukcję reprezentującą skompilowane zapytanie.
Gdy mamy już sprawozdanie, możemy z nim zrobić wiele rzeczy. Możemy wykonać na nim callexecute raz lub więcej razy. Po wywołaniu polecenia execute dla zapytania, które zwraca dane, można użyć jednej z funkcji pobierania, aby pobrać te dane.
Funkcje takie jak run i quickQuery 'używają instrukcji i wykonują się wewnętrznie; są to po prostu skróty, które pozwalają szybko wykonywać typowe zadania. Gdy potrzebujesz większej kontroli nad tym, co się dzieje, możesz użyć instrukcji zamiast funkcji takiej jak run.

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

Rysunek 4

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

Rysunek 5

9.1 Instrukcja do wstawiania wielu wartości za pomocą jednego zapytania

Tutaj tworzymy przygotowaną instrukcję i nazywamy ją stmt. Następnie wykonujemy tę instrukcję cztery razy i za każdym razem przekazujemy inne parametry. Parametry te są używane w celu aby zamienić znaki zapytania w oryginalnym ciągu zapytania. Na koniec zatwierdzamy zmiany i odłączamy bazę danych.

9.2 funkcja executeMany

HDBC zapewnia również funkcję executeMany, która może być przydatna w takich sytuacjach. executeMany po prostu pobiera listę wierszy danych do wywołania instrukcji.

```
ghci> conn <- connectSqlite3 "test1.db"
ghci> quickQuery' conn "SELECT * from test where id < 2" []
[[SqlString "0",SqlNull],[SqlString "0",SqlString "zero"],
[SqlString "1",SqlString "one"],[SqlString "0",SqlNull],
[SqlString "0",SqlString "zero"],[SqlString "1",SqlString "one"]]
ghci> disconnect conn
```

Rysunek 6

10 Odczyt rezultatów

Do tej pory omawialiśmy zapytania, które wprowadzają lub zmieniają dane.
Przejdźmy teraz do odzyskiwania danych z bazy danych. Typ funkcji quickQuery 'wygląda bardzo podobnie do run, ale zwraca listę wyników zamiast liczby zmienionych wierszy. quickQuery 'jest zwykle używane z instrukcjami SELECT.

10.1 Przykład z quickQuery

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

10.2 Kod programy do odczytania wyników

Ten program robi w większości to samo, co nasz przykład z ghci, ale z nowym dodatkiem: funkcją convRow. Ta funkcja pobiera wiersz danych z bazy danych i konwertuje go na łańcuch. Ten ciąg można następnie łatwo wydrukować.

Zwracamy uwagę, jak pobraliśmy wartość intid z fromSql bezpośrednio, ale przetworzyliśmy fromSql sqlDesc jako typ String typu Maybe.
Zadeklarowaliśmy, że pierwsza kolumna w tej tabeli nigdy nie może zawierać wartości NULL, ale druga kolumna może. Dlatego możemy bezpiecznie zignorować potencjał NULL w pierwszej kolumnie, ale nie w drugiej.
Jest możliwe użycie fromSql do bezpośredniego przekonwertowania drugiej

```
 -- file: ch21/query.hs
 import Database.HDBC.Sqlite3 (connectSqlite3)
 import Database.HDBC


{- | Define a function that takes an integer representing the maximum
id value to look up.  Will fetch all matching rows from the test database
and print them to the screen in a friendly format. -}
query :: Int -> IO ()
query maxId =
    do -- Connect to the database
       conn <- connectSqlite3 "test1.db"

       -- Run the query and store the results in r
       r <- quickQuery' conn
            "SELECT id, desc from test where id <= ? ORDER BY id, desc"
            [toSql maxId]

       -- Convert each row into a String
       let stringRows = map convRow r

       -- Print the rows out
       mapM_ putStrLn stringRows

       -- And disconnect from the database
       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

kolumny na łańcuch, a to nawet działałoby - aż do napotkania wiersza z wartością NULL na tej pozycji. Spowodowałoby to bezwzględny wyjątek.

11 Odczyt sprawozdania

Istnieje wiele sposobów odczytywania danych ze stwierdzeń, 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 odczyt do sprawozdania, używamy przygotowania tak samo, jak w przypadku instrukcji, która będzie używana do zapisywania danych. Używamy również execute, aby wykonać to na serwerze bazy danych, a następnie możemy użyć różnych funkcji do odczytu danych z instrukcji. Funkcja fetchAllRows 'zwraca [[SqlValue]], podobnie jak quickQuery'. Istnieje również funkcja o nazwie sFetchAllRows ', która konwertuje dane każdej kolumny na Maybe String przed ich ponownym przekształceniem. Wreszcie istnieje funkcja fetchAllRowsAL ', która zwraca (String, SqlValue) pary dla każdej kolumny. Ciąg to nazwa kolumny zwrócona przez bazę danych.

Możemy również odczytywać dane po jednym wierszu na raz, wywołując funkcję fetchRow, która zwraca IO (Maybe [SqlValue]). Będzie to Nothing (nic), jeśli wszystkie wyniki zostały już przeczytane lub jeden wiersz w przeciwnym razie.

12 "Leniwe czytanie"

Możliwe jest 'leniwe' odczytywanie danych z baz danych. Może to być szczególnie przydatne w przypadku zapytań zwracających wyjątkowo dużą ilość danych. Czytając dane leniwie, nadal możemy korzystać z wygodnych funkcji, takich jak fetchAllRows, zamiast konieczności ręcznego odczytywania każdego wiersza w momencie, gdy się pojawi. Jeśli będziemy ostrożnie obchodzić się z danymi, możemy uniknąć buforowania wszystkich wyników w pamięci.

Leniwe czytanie z bazy danych jest jednak bardziej złożone niż czytanie z pliku. Kiedy skończymy leniwie odczytywać dane z pliku, plik jest zamykany - co zazwyczaj jest w porządku.
Gdy skończymy leniwie odczytywać dane z bazy danych, połączenie z bazą danych jest nadal otwarte - możemy na przykład przesyłać do niej inne zapytania .

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

Rysunek 8

Niektóre bazy danych mogą nawet obsługiwać wiele jednoczesnych zapytań, więc HDBC nie może po prostu zamknąć połączenia, gdy skończymy.

Podczas korzystania z leniwego czytania niezwykle ważne jest, aby zakończyć czytanie całego zestawu danych, zanim spróbujemy zamknąć połączenie lub wykonać nowe zapytanie. Zachęca się do korzystania ze ścisłych funkcji lub przetwarzania wiersz po wierszu, jeśli to możliwe, aby zminimalizować złożone interakcje z leniwym czytaniem.

Aby leniwie czytać z bazy danych, używamy tych samych funkcji, których używaliśmy wcześniej, bez apostrofu. Na przykład fetchAllRows zamiast fetchAllRows '. Rodzaje leniwych funkcji są takie same jak ich ścisłych kuzynów.

12.1 Przykład leniwego czytania

13 Metadane Bazy Danych

Czasami przydatne może być poznanie przez program informacji o samej bazie danych, na przykład program może chcieć zobaczyć, jakie tabele istnieją, aby mógł automatycznie utworzyć brakujące tabele lub zaktualizować schemat bazy danych. W niektórych przypadkach program może potrzebować

```
ghci> conn <- connectSqlite3 "test1.db"
ghci> getTables conn
["test"]
ghci> proxiedClientName conn
"sqlite3"
ghci> dbServerVer conn
"3.5.6"
ghci> dbTransactionSupport conn
True
ghci> disconnect conn
```

Rysunek 9

zmienić swoje zachowanie w zależności od używanego zaplecza bazy danych.

Po pierwsze istnieje funkcja getTables, która pobierze listę zdefiniowanych tabel w bazie danych, możesz też skorzystać z funkcji opiszTable, która dostarczy informacji o zdefiniowanych kolumnach w danej tabeli.

Możemy dowiedzieć się o używanym serwerze bazy danych, wywołując na przykład dbServerVer iproxiedClientName. Za pomocą funkcji dbTransactionSupport można określić, czy dana baza danych obsługuje transakcje.

13.1 Przykład

Możemy również dowiedzieć się o wynikach określonego zapytania, uzyskując informacje z jego wyciągu. Funkcja describeResult zwraca [(String, SqlColDesc)], listę par. Pierwsza pozycja podaje nazwę kolumny, a druga zawiera informacje o kolumnie: typ, rozmiar i czy może mieć wartość NULL. Pełną specyfikację podano w dokumentacji HDBC API.

14 Obsłga błędów

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

ghci nie wie, jak wyświetlić błąd SqlError na ekranie, gdy wystąpi. Jeśli wyjątek spowoduje zakończenie działania programu, nie wyświetli użytecznego komunikatu.

```
ghci> conn <- connectSqlite3 "test1.db"
ghci> quickQuery' conn "SELECT * from test2" []
*** Exception: (unknown)
ghci> disconnect conn
```

Rysunek 10

```
ghci> conn <- connectSqlite3 "test1.db"
ghci> handleSqlError $ quickQuery' conn "SELECT * from test2" []
*** Exception: user error (SQL error: SqlError {seState = "", seNativeError =
seErrorMsg = "prepare 20: SELECT * from test2: no such table: test2"})
ghci> disconnect conn
```

Rysunek 11

14.1 przykład

Tutaj próbowaliśmy wybrać [SELECT] dane z tabeli, która nie istniała. Komunikat o błędzie, który otrzymaliśmy, nie był pomocny. Istnieje funkcja narzędzia handleSqlError, która przechwyci SqlError i ponownie zgłosi go jako IOError. W tej formie będzie można go wydrukować na ekranie, ale programowe wyodrębnienie określonych informacji będzie trudniejsze.

14.1.1 handleSqlError Przykład

Tutaj otrzymaliśmy więcej informacji, w tym komunikat informujący, że nie ma takiej tabeli jak test2. Jest to znacznie bardziej pomocne. Wielu programistów HDBC stosuje standardową praktykę, aby uruchamiać swoje programy za pomocą "main = handleSqlError S do",
co zapewni, że każdy niezłapany SqlError zostanie wydrukowany w pomocny sposób.

15 Bibliografia

- Bryan O'Sullivan, John Goerzen, Don Stewart - "Real World Haskell- O Reilly Media", 2008 - Chapter 21 - Using Database