Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

РОБОТА З БАЗАМИ ДАНИХ У HASKELL

Текстова частина до курсової роботи
за спеціальністю „Інженерія програмного забезпечення”

Керівник курсової роботи
к.ф-м.н., доцент Проценко В. С.

_____
(підпис)
“____” _____ 2020 р.

Виконав студент Теліженко С.О.
“____” _____ 2020 р.

Київ 2020

Зміст

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ

Зав. кафедри інформатики,

доцент, к.ф.-м.н.

_____ С. С. Гороховський

(підпис)

“____”_____ 2020 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на курсову роботу

студенту Теліженку Станіславу

____4_____курсу факультету інформатики

ТЕМА: Робота з базами даних у Haskell

Вихідні дані:

- Web API, розроблений на мові Haskell, що взаємодіє з СКБД PostgreSQL

Зміст ТЧ до курсової роботи:

Вступ
1. Аналіз предметної області та наявних проблем і задач
2. Теоретичні засади роботи з базами даних у Haskell
3. Реалізація практичної частини роботи
Висновки
Список джерел
Додатки (за необхідністю)

Дата видачі “___” _____ 2019 р.

Керівник _____ Завдання отримано _____

Календарний план виконання курсової роботи

Тема: Робота з базами даних у Haskell

№ п/п	Назва етапу курсового проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу	листопад 2019 р.	
2.	Огляд літератури за темою роботи	листопад - грудень 2019 р.	
3.	Оволодіння навичками роботи з базами даних у Haskell	грудень - лютий 2019 р.	
4.	Створення власного застосунку на Haskell, який працюватиме з базою даних	лютий - березень 2020 р.	
5.	Написання пояснювальної роботи	лютий - березень 2020 р.	
6.	Створення слайдів для доповіді та написання доповіді	березень 2020 р.	
7.	Надання роботи керівнику для перевірки	березень 2020 р.	
8.	Корегування роботи за результатами перевірки керівником	березень 2020 р.	
9.	Остаточне оформлення пояснювальної роботи та слайдів	березень - квітень 2020 р.	
10.	Захист курсової роботи	19 квітня 2020 р.	

Студент Теліженко С.О. _____

Керівник Проценко В. С. _____

“_____” _____ _____ р.

Вступ

Однією з основних задач при розробці промислових застосунків є взаємодія з базами даних. Вона пов’язана з рядом цікавих задач, таких як підтримка транзакцій та обробка мережевого з’єднання з СКБД, моделювання предметної області тощо.

Окрім того, чисто функціональна мова програмування Haskell має низку особливостей роботи з базами даних. Адже взаємодія з мережею та дисковим простором – це сторонні ефекти, які у Haskell вимагають особливої уваги.

Для вирішення цих задач Haskell пропонує низку інструментів, які різняться своїми підходами та методологіями. Є такі, що спеціалізуються на конкретній СКБД, є ті, що намагаються охопити ряд різних систем. Такі, що пропонують працювати з базою даних на низькому рівні і такі, що дають можливості, наближені до популярної у об’єктно-орієнтованих мовах програмування технології Object-Relational Mapping.

Ця робота присвячена цим підходам та методологіям, а також їх використанню на практиці.

Анотація

Роботу присвячено задачам, які виникають при роботі з базами даних на мові програмування Haskell: управління підключенням до СКБД, транзакції, формування запитів та обробка їх результатів. Також у теоретичній частині розглянуті такі техніки, як міграції та генерація типів на основі схеми, статична перевірка коректності запитів, у контексті функціональної парадигми. Для демонстрації результатів дослідження розроблено консольне застосування мовою Haskell, що взаємодіє з СКБД PostgreSQL.

1. Аналіз предметної області та наявних проблем і задач

1.1 Персистентність у Haskell

Робота майже будь-якого прикладного застосунку так чи інакше пов’язана з маніпуляціями над персистентним станом – таким, який має можливість існувати довше, ніж процес який його створив. Якщо таких маніпуляцій не виконувати, уся інформація, надана користувачем під час взаємодії з застосунком, зникатиме разом із завершенням його роботи. Для забезпечення персистентності доводиться програмно взаємодіяти із «зовнішнім світом» - дисковим простором, мережею. Іншими словами, персистентність вимагає від програми читання та модифікації її середовища виконання, з чим у Haskell є декілька особливостей.

Справа в тому, що Haskell – чисто функціональна мова програмування, що з одного боку дає ряд переваг, серед яких: можливість підтримки відкладених обчислень на рівні системи програмування, розробка паралельних програм без затрат на синхронізацію, а з іншого боку накладає ряд обмежень. Ці обмеження визначаються необхідними властивостями чистих функцій [1]:

- Функція має бути детермінованою – для одного набору аргументів повертати один результат.
- Функція не повинна мати сторонніх ефектів – не змінювати середовище виконання.

Ці обмеження напряму вказують на те, що чиста функція не взаємодіє з середовищем виконання – результат його читання є недетермінованим, а його модифікація – сторонній ефект. Насправді, чистота функції обмежує роботу не лише з персистентністю: читання та запис у консоль, відображення вікон, обробка дій клавіатури та миші – усе це вимагає взаємодіяти із середовищем виконання, «зовнішнім світом».

Поки що ці обмеження сформульовані суто з точки зору теорії. Подивимося, які існують проблеми на рівні реалізації, на прикладі наступної програми:

```
import qualified Data.ByteString as B

readFromSocket :: String -> Int -> Int -> B.ByteString
readFromSocket ip port amount = undefined

readFromSocketTwice :: String -> Int -> Int -> B.ByteString
readFromSocketTwice ip port amount =
      B.append    (readFromSocket ip port amount)
                  (readFromSocket ip port amount)
```

У цього коду, якби він міг працювати, були б дві проблеми:

- Мають місце два виклики однієї функції з однаковим набором аргументів. З огляду на детермінованість чистих функцій, система виконання може визначити другий виклик як надлишковий та одразу підставити замість нього результат виконання першого, хоча в даному випадку це неприйнятно. Назвемо це проблемою кешування.
- Навіть якщо виконаються два виклики, невідомо в якому саме порядку. З точки зору функціонального програмування, у виразі вигляду *f x y* порядок обчислення *x* та *y* не важливий [1]. Але в цьому випадку, зміна порядку вплине на результат. Назвемо це проблемою перевпорядкування.

Очевидно, має існувати рішення цієї проблеми. Основою цього рішення є монади – абстракція зв'язаних обчислень. Так виглядає клас монад у Haskell[2]:

```
class Monad m where
  (>>=)  :: m a -> (a -> m b)   -> m b
  (>>)   :: m a ->  m b         -> m b
  return ::   a                 -> m a
  fail   :: String -> m a
```

Монади не є основною темою цієї роботи, тому я зупинюсь лише на тих аспектах, що стосуються реалізації функцій з побічним ефектом.

Оператор *(>>)* зв'язує дві дії: на виході отримаємо дію, що полягає у виконанні спочатку першої, потім другої. Результатом обчислень є результат другої дії, результат першої відкидається. Це вирішує проблему перевпорядкування: для обчислення результату функції *act2* у виразі *(act1 >> act2)* необхідне обчислення функції *act1*.

В свою чергу, оператор *(>>=)* виконує дію, та передає її результат як аргумент до функції, що повертає другу дію. Як і у випадку з *(>>)*, результатом обчислення цілого виразу є результат другої дії. За допомогою цього оператора з'являється можливість неявно передавати між рядом дій стан обчислень: кожна дія, можливо, модифікує цей стан та передає його як аргумент наступній. Так вирішується проблема кешування: якщо одна дія змінює стан обчислень (у нашому випадку – стан середовища виконання), то наступна гарантовано отримає інший набір аргументів.

Як результат, монади є зручним механізмом для управління сторонніми ефектами, при чому цей механізм не порушує чисто функціональну природу мови. Стандартна бібліотека Haskell містить монаду *IO* (введення-виведення), яка і реалізує сторонні ефекти. Ця монада має декілька особливостей[3]:

- *IO t* – тип, що описує дію, яка взаємодіє з середовищем виконання та результат якої має тип *t*.
- Конструктори монади приховані від користувача. Це унеможливлює її використання всередині чистих функцій.
- Монада продукує сторонній ефект не в момент обчислення (наприклад, при зв'язуванні двох дій), а саме в момент виконання.

Усі інші типи та бібліотеки, що мають сторонні ефекти, як-от взаємодія з мережею (*network, conduit*), з системним оточенням (*System.Environment*) побудовані на монаді *IO*.

1.2 Основні задачі при роботі з базами даних

1.2.1 Системи керування базами даних

Система керування базою даних (далі СКБД) – програмне забезпечення, за допомогою якого користувачі можуть визначати, створювати та підтримувати базу даних, а також здійснювати контрольований доступ до неї. Слід зауважити, що матеріал цього розділу стосуватиметься насамперед реляційних СКБД.

Фактично СКБД – це та система, що приймає запит користувача (у нашому випадку – запит мовою SQL), інтерпретує його, виконує відповідні інструкції та відповідає на запит. Це складний механізм з багаторівневою архітектурою, але щоб охопити основні принципи програмної взаємодії з ним достатньо розглянути лише декілька основних його компонентів [4]:

- *Підсистема визначення даних.* Складається з інструментів для означення та зміни структури бази даних: реляцій, відношень, обмежень, профілів користувачів. Взаємодія з нею відбувається за допомогою мови визначення даних (Data Definition Language, DDL) – нею визначають усі об’єкти бази даних та формують концептуальну схему.
- *Системний каталог.* Містить метадані бази даних: її структуру, залежності між її об’єктами, права доступу та ін.
- *Менеджер комунікацій.* Окрема система, під’єднана до СКБД, що виконує функції обробки з’єднань з віддаленими користувачами у розподіленому середовищі, обробки повідомлень що надходять до бази даних, відправки відповідей. Менеджер комунікацій відповідає за ефективну роботу бази даних у клієнт-серверному середовищі.
- *Двигун бази даних.* Відповідає за зв’язок усіх інших підсистем з фізичним пристроєм за посередництвом операційної системи: доступ до дискової пам’яті, операції введення/виведення, передача даних між пам’яттю та системним буфером.

1.2.2 Підключення до СКБД

За способом підключення СКБД класифікують як клієнт-серверні та вбудовані. Вбудованими називають такі СКБД, що напряму прив’язані до клієнтського застосунку та виконуються у одному процесі з ним. Клієнт-серверні в свою чергу виконуються в окремому, автономному процесі, можливо навіть на окремому від клієнтського застосунку пристрої[5].

Вбудовані СКБД мають низку переваг над клієнт-серверними: вони більш прості у розробці, підтримці та експлуатації, комунікація з ними пов’язана з набагато меншим обсягом накладних витрат. Втім, у розробці промислових систем для збереження даних, до яких необхідно організувати спільний доступ, найчастіше обирають клієнт-серверні через вищий рівень безпеки, менший ризик втрат та пошкодження даних та можливість розподілу обчислювального навантаження. Вбудовані ж виявляються більш прийнятними для під’єднання до клієнтського (мобільного чи настільного) застосунку та збереження локальних, менш чутливих до різного роду збоїв даних: інформація про авторизованого користувача, уподобання в компонуванні користувацького інтерфейсу та ін.

Дана робота присвячена програмній взаємодії саме з клієнт-серверними СКБД. Особливість такої взаємодії полягає в тому, що усі запити виконуються за допомогою передачі повідомлень через мережу згідно деякого, специфічного для конкретної СКБД, протоколу.

Отже, будь-яка маніпуляція з базою даних в такому випадку відбувається у наступному циклі:

1. Відкрити підключення до серверного сокету. Для цього може знадобитись не лише IP-адреса та порт, а й ім’я бази даних (один екземпляр СКБД-сервера може обслуговувати декілька екземплярів баз даних), ім’я користувача та пароль.
2. Відправити запит, коректний с точки зору протоколу обміну повідомленнями.
3. Отримати та обробити повідомлення з результатом.

4. Закрити підключення або повернутися до кроку 2.

Якщо кроки 2 та 3 вимагають лише мати інформацію про протокол обміну повідомленнями, то кроки 1 та 4 мають деякі особливості. По-перше, не всі СКБД дозволяють виконувати декілька запитів в рамках одного підключення.

По-друге, як було зазначено раніше, у порівнянні із вбудованими, взаємодія з клієнт-серверними СКБД пов’язана з більшими накладними витратами. Ці витрати полягають у тому, що дані передаються мережею та у необхідності кожного запиту відкривати та закривати підключення.

Пул підключень до бази даних (Database Connection Pool, DBCP) – інструмент, задача якого – мінімізувати ці накладні витрати. DBCP – одна з варіацій класичного шаблону проектування Pool of Objects, що призначений для управління циклом життя об’єктів, які пов’язані з деякими ресурсами, захоплення та звільнення яких вимагає серйозних затрат у часі[6].

Отже, DBCP зберігає набір відкритих підключень та надає інтерфейс доступу до них. Якщо користувачу необхідне підключення, то DBCP поверне одне з вже відкритих вільних. Якщо в даний момент вільних немає, створюється нове. Як тільки користувач завершує роботу з підключенням, воно стає доступним для інших. Якщо підключення не використовується протягом певного часу, воно закривається.

1.2.3 Транзакції

Транзакція – одне з основоположних понять у базах даних, що означає атомарну (неподільну) сукупність операцій над даними. Транзакції володіють наступними властивостями, відомими як ACID (Atomicity-Consistency-Isolation-Durability)[7]:

- *Атомарність*. Операції, з яких складається транзакція, мають або виконатися усі, або не виконається жодна.
- *Цілісність*. Транзакція переводить дані з одного цілісного стану у інший цілісний стан.

- *Ізоляція*. Ефект транзакції невидимий для інших транзакцій поки вона не буде повністю завершена.
- *Довговічність*. Результат завершеної транзакції є постійним.

Механізм транзакцій є необхідним для повноцінної роботи будь-якої бази даних. Атомарність, цілісність та довговічність гарантують, що дані залишаться цілісними навіть у випадку помилок і збоїв, регулюють коректне відновлення у виключних ситуаціях. Ізоляція забезпечує коректну одночасну взаємодію декількох користувачів з даними таким чином, щоб кожен користувач не міг бачити деяку операцію у її проміжному стані, а лише коли вона повністю завершиться та дані будуть цілісними.

З точки зору програмної взаємодії з реляційними СКБД, існують два варіанти управління транзакціями: явний та неявний(так званий автокоміт режим). У першому випадку після одної чи декількох SQL-інструкцій необхідно виконати оператор *COMMIT*, після якого транзакція завершиться, або оператор *ROLLBACK*, який скасує усі ефекти транзакції. У другому, кожна інструкція неявно супроводжується відповідним викликом оператору *COMMIT*. Варто зазначити, що не усі СКБД підтримують режим автокоміт.

2. Теоретичні засади роботи з базами даних у Haskell

2.1 Інструменти Haskell для роботи з СКБД

2.1.1 Інструменти, специфічні для конкретної СКБД

Цей розділ присвячено різноманіттю бібліотек мови Haskell для взаємодії з СКБД. Умовно їх можна класифікувати як такі, що надають інтерфейс роботи з однією конкретною СКБД та такі, що впроваджують абстракцію над декількома. У даному підрозділі мова піде про перший тип.

Такі бібліотеки мають беззаперечну перевагу – вони мають змогу охопити усі особливості конкретної СКБД. Як приклад, бібліотека *postgregsql-simple* містить модуль *Database.PostgreSQL.Simple.Arrays,* що надає інтерфейс розпізнавання та перетворення масивів об'єктно-реляційної СКБД PostgreSQL. В свою чергу, бібліотеки, призначені для роботи з певним набором різних СКБД (які будуть детально розглянуті у підрозділі 2.3.2), як-от *hdbc*, не в змозі забезпечити такий рівень деталізації, адже деякі з цих СКБД не підтримують абстракцію масивів. Але у цієї переваги є ціна – використовуючи такі бібліотеки, буде складніше мігрувати з однієї СКБД на іншу. [8]

Як приклад такої бібліотеки можна розглянути *sqlite-simple* – інструмент взаємодії із вбудованою реляційною СКБД SQLite. Незважаючи на те, що вона не клієнт-серверна, з її допомогою усе ж можна відтворити основні аспекти роботи з реляційними базами даних.

Для того, щоб почати роботу з бібліотекою, необхідно розглянути декілька її базових типів: *Connection, SQLData, Query, ToField, ToRow, FromField* та *FromRow*.

Connection – абстракція підключення до СКБД. Будь-які функції, що виконують маніпуляції з базою даних, вимагають екземпляр цього типу як аргумент. Бібліотека надає функцію *open*, що за рядком підключення до бази даних повертає відкрите підключення, та функцію *close,* що підключення закриває.

Очевидно, що кожен виклик *open* повинен супроводжуватися викликом *close*. Більше того, функція *close* має бути викликана навіть у випадку виникнення виключних ситуацій. Це призводить до рутинного дублювання коду, тому бібліотека також надає функцію *withConnection*, аргументами якої є рядок підключення та дія над підключенням. Вона відкриває підключення, виконує дію та закриває підключення навіть у випадку помилок. [9]

```
newtype Connection = Connection
      { connectionHandle :: Database.SQLite3.Database }
open :: String -> IO Connection
close :: Connection -> IO ()
withConnection :: String -> (Connection -> IO a) -> IO a
```

SQLData – обгортка над типами, які підтримує база даних. Цей тип є основою конверсії типів мови Haskell у типи SQLite та навпаки. Для кожного типу, що присутній у СКБД, *SQLData* має окремий конструктор [9]:

```
data SQLData
    = SQLInteger    !Int64
    | SQLFloat      !Double
    | SQLText       !Text
    | SQLBlob       !ByteString
```

Клас *ToField* призначений для перетворення типів мови Haskell на відповідні типи, які підтримує база даних. Бібліотека впроваджує його реалізації для основних вбудованих типів у Haskell. Варто зазначити, що значення деякого типу Haskell *t*, що в базі даних можуть бути пустими (*NULL*), відповідають типу *Maybe t*. Тип *t* в цьому випадку повинен бути екземпляром *ToField*. [9]

```haskell
class ToField a where
    toField :: a -> SQLData

-- реалізації для деяких стандартних типів
instance ToField Double where
    toField = SQLFloat

instance ToField Int where
  toField = SQLInteger . fromIntegral

-- реалізація для значень, що можуть містити NULL
instance (ToField a) => ToField (Maybe a) where
  toField Nothing  = SQLNull
  toField (Just a) = toField a
```

ToRow виконує схожу функцію – конвертує типи Haskell у кортежі бази даних, які в контексті бібліотеки представлені списками значень *SQLData*. Для типів, визначених користувачем, доведеться писати окрему реалізацію, однак бібліотека надає реалізацію для кортежів, кожне значення яких є екземпляром *ToField*, чого цілком достатньо для того, щоб ця реалізація була максимально простою: [9]

```haskell
class ToRow a where
    toRow :: a -> [SQLData]

instance (ToField a, ToField b) => ToRow (a,b) where
    toRow (a,b) = [toField a, toField b]
```

Для того щоб почати взаємодіяти з базою даних, необхідно розглянути ще один тип – *Query*. Він представляє собою абстракцію над запитом до СКБД. Ось його визначення: [9]

```haskell
newtype Query = Query {
      fromQuery :: T.Text
    } deriving (Eq, Ord, Typeable)

instance IsString Query where
      fromString = Query . T.pack
```

На перший погляд, кожен запит у такому випадку доведеться обгортати у екземпляр цього типу, хоча він містить єдине поле – текст запиту і у більшості випадків достатньо лише рядкового літералу. На щастя, ця проблема вирішується через розширення синтаксису *OverloadedStrings*. Якщо воно

включене, компілятор дозволяє використовувати рядкові літерали замість типів, що реалізують клас *IsString*.

Тепер ми можемо використати функції *execute* та *execute_*: [9]

```
execute :: ToRow q => Connection -> Query -> q -> IO ()
execute_ :: Connection -> Query -> IO ()
```

Функція execute приймає підключення до СКБД, запит та екземпляр ToRow, який можна перетворити на список позиційних аргументів запиту. execute_ – аналогічна функція, але вона призначена для запитів, які не мають аргументів. Як висновок, тепер ми можемо написати просту функцію, яка додаватиме новий запис до бази даних:

```
{-# LANGUAGE OverloadedStrings #-}

import Database.SQLite.Simple

data User = User {
    _id :: Int,
    name :: String
}

addUser :: String -> User -> IO ()
addUser connStr user = withConnection connStr $ \conn ->
    execute conn "INSERT INTO users VALUES (?) (?)"
            (_id user, name user)
```

Але поки що розглянутих інструментів достатньо лише для запитів, які не повертають жодних результатів. Аби отримувати записи з бази даних, необхідно використати класи *FromField* та *FromRow*.

Вони виконують обернену до класів ToField та ToRow відповідно функцію: конвертують записи в базі даних у типи Haskell. Вони визначаються таким чином: [9]

```
type  FieldParser a = Field -> Ok a

newtype RowParseRO = RowParseRO { nColumns :: Int }

newtype RowParser a = RP
{ unRP :: ReaderT RowParseRO (StateT (Int, [SQLData]) Ok) a }

class FromField a where
      fromField :: FieldParser a

class FromRow a where
      fromRow :: RowParser a
```

Задача монади *Ok* – обробити помилкові ситуації при перетворенні поля у екземпляр типу *a*. *RowParser* – монада, побудована на основі трансформатора монад *ReaderT*. Вона додає до обчислення значення рядка оточення: номер колонки у реляції та її ім'я.

Як і у випадку з *ToField*, бібліотека надає реалізації *FromField* для основних вбудованих типів Haskell. Також надає вона функцію-парсер *field*, яка перетворює значення на поле екземпляра.

У випадку, якщо користувацький тип складається з набору полів, типи яких є вбудованими, а їх ім'я співпадають з назвами відповідних атрибутів реляції, то для того щоб зробити цей тип екземпляром *FromRow*, достатньо реалізувати *fromRow* як комбінацію *field* [10].

Для запитів, що повертають результат, у бібліотеці є дві функції, симетричні до *execute* та *execute_:* [9]

```
query :: (ToRow q, FromRow r) => Connection -> Query -> q -> IO [r]

query_ :: FromRow r => Connection -> Query -> IO [r]
```

Використовуючи ці можливості розглянутої бібліотеки, функцію отримання користувача за його ідентифікатором, можна реалізувати наступним чином:

```
{-# LANGUAGE OverloadedStrings #-}

import Control.Applicative
import Database.SQLite.Simple
import Database.SQLite.Simple.FromRow

data User = User {
    _id :: Int,
    name :: String
}

instance FromRow User where
    fromRow = User <$> field <*> field

fetchUser' :: Int -> Connection -> IO (Maybe User)
fetchUser' userId conn = do
    us <- (query conn "SELECT * FROM users \
                    \ WHERE _id = (?)" (Only userId))::IO [User]
    case us of
        [] -> return Nothing
        users -> return $ Just $ head users

fetchUser :: String -> Int -> IO (Maybe User)
fetchUser connStr userId = withConnection connStr (fetchUser' userId)
```

2.1.2 Абстракції над декількома СКБД

У цьому підрозділі мова піде про інструменти, які не фокусуються на конкретній СКБД, а надають користувачеві шар абстракції над декількома з них: зазвичай – разом зі специфічними з'єднувачами для кожної з них. Набір таких систем варіюється для кожної конкретної бібліотеки: деякі фокусуються на реляційних, деякі покривають і нереляційні теж [11].

Головна перевага таких інструментів у тому, що початковий вибір СКБД може бути переглянутий в процесі розробки і кількість коду, що в такому випадку необхідно буде переписати, буде значно меншою ніж у випадку з першим типом інструментів.

Але є і недоліки: такі бібліотеки зазвичай не покривають специфічний функціонал кожної окремої СКБД, а вимушені обмежуватися тими можливостями, які є спільними для кожної з них.

Розглянемо приклад такого інструменту – Haskell Database Connectivity (HDBC). Для його використання знадобиться як мінімум дві бібліотеки – *hdbc*, що надає узагальнений інтерфейс, та бібліотеку-з'єднувач, для конкретної обраної СКБД. Наприклад, для СКБД SQLite знадобиться *hdbc-sqlite3*, для PostgreSQL – *hdbc-postgresql*.

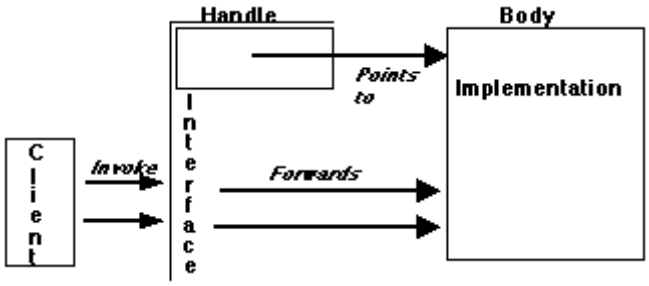
Для розуміння того, як ці дві бібліотеки працюватимуть разом, необхідно розглянути клас *IConnection*, визначений у *hdbc*, що абстрагує підключення до деякої СКБД. Нижче приведені найважливіші частини його визначення[12]:

```
class IConnection conn where
  run :: conn -> String -> [SqlValue] -> IO Integer
  commit :: conn -> IO ()
  rollback :: conn -> IO ()
  disconnect :: conn -> IO ()
  prepare :: conn -> String -> IO Statement
```

Зауважимо, що бібліотека не надає жодної реалізації цього класу. Причина цьому – реалізація буде специфічною для кожної СКБД, тому вона міститься у відповідних бібліотеках-з'єднувачах [11].

Тип *SqlValue* тут – основний для конвертації типів мови Haskell у типи, які підтримує СКБД. Як і у випадку з *sqlite-simple*, він має окремий конструктор для кожного типу бази даних. *Statement* абстрагує підготовлену для виконання інструкцію мовою SQL та надає інтерфейс її виконання.

Таким чином, архітектура таких бібліотек будується за принципом, схожим на шаблон проектування handle-body (дескриптор-тіло). Вона розділяє концептуально цілісний програмний компонент на два модулі: дескриптор, що визначає абстрактний програмний інтерфейс, з яким взаємодіє користувач, та тіло, яке містить реалізацію цього інтерфейсу[13]:



Саме дескриптор обробляє запити користувача, але у більшості випадків він просто пересилає ці запити тілу.

У випадку бібліотеки-дескриптора *hdbc*, вона надає користувачеві низку функцій вищого рівня, що дозволяють зручно взаємодіяти з базою даних, але усі вони спираються функції, визначені класом *IConnection*, реалізація яких знаходиться у бібліотеках-з'єднувачах.

Використовуючи *hdbc*, код для отримання користувача за його ідентифікатором з розділу 1.3.1 можна переписати так:

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

data User = User { _id :: Int, name :: String }

fetchUser :: String -> Int -> IO (Maybe User)
fetchUser connStr userId = do
    conn <- connectSqlite3 connStr
    res  <- quickQuery "SELECT * FROM users WHERE _id = (?)" [toSql userId]
    disconnect conn
    case res of
        [] -> return Nothing
        [[SqlInteger uid, SqlString uname]] -> Just $ User uid uname
```

Припустимо, що нам необхідно мігрувати з SQLite на PostgreSQL. У такому випадку, код вище зазнає наступних змін: замість *connectSqlite3* з модуля *Database.HDBC.Sqlite3* необхідно використати *connectPostgreSQL* з модуля *Database.HDBC.PostgreSQL*. Але при цьому необхідно врахувати ряд обмежень:

- Це можливо, зокрема, завдяки тому, що у коді використаний стандартний SQL і ці конструкції підтримуються обома СКБД. У протилежному випадку, змін було б більше.

- Використання об’єктних особливостей СКБД PostgreSQL залишається недоступним. Для них необхідно використати інструмент, специфічний для цієї СКБД.

2.2 Особливості роботи з СКБД на низькому рівні

Розглянуті у першому розділі інструменти мають низку спільних рис. Це обумовлено тим, що вони пропонують працювати з СКБД (незалежно від того, з однією чи з декількома) на низькому рівні абстракції. Це відображається у наступних особливостях.

По-перше, усі запити до бази даних зберігаються прямо у вихідному коді, у вигляді рядкових літералів. Частково це можна виправити – винести їх у окремий модуль, чи декілька модулів, або навіть зчитувати їх з окремих текстових файлів. Але це несуттєво вплине на ситуацію.

По-друге, схема бази даних відображається в коді неявно, або не відображається зовсім. Ця проблема також вирішується лише частково – можна сформувати окремий модуль з типами, які будуть відповідати реляціям у базі даних. Але ми досі вимушені конвертувати ці типи у записи в базі даних та навпаки, та дублювати ці дані у запитах.

Усі ці особливості спричиняють низку незручностей при розробці:

- Текстові запити, як правило, є джерелом механічних помилок.
- Навіть якщо бібліотека не сильно зв’язана з конкретною СКБД, написання текстових запитів, що містять конструкції, специфічні для цієї СКБД, ускладнює міграцію на інші системи.
- При змінах схеми доведеться продублювати усі зміни у типи, які відповідають реляціям, та запити до цих реляцій.

Найголовніше у цих проблемах те, що вони виникають на етапі виконання програми. Помилка у тексті запиту або його невідповідність останній версій схеми стане помітною, коли цей запит буде виконано: СКБД повідомить про некоректний запит, виникне виключна ситуація. Невідповідність типів при їх конвертації спричинить або виключну ситуацію, або, що гірше, некоректний її результат.

Один зі способів попередити ці проблеми – тестування. Але розробити тести для кожного типу запитів займе, можливо, стільки ж часу, скільки і

розробити самі ці запити. І тестування – це все ще перевірка на етапі виконання, а не на етапі компіляції.

Очевидно, що інфраструктура Haskell, як мови зі строгою статичною типізацією, має запропонувати інструменти, що попереджують ці помилки на етапі компіляції. Таких інструментів чимало, і вони впроваджують високий рівень абстракції при роботі з базою даних.

Цей підрозділ присвячено стандартним підходам та методологіям, що дозволяють використати строгу статичну типізацію для виявлення помилок при роботі з базою даних на етапі компляції.

2.3 Відповідність між схемою бази даних та типами даних у Haskell

2.3.1 Генерація типів на основі схеми бази даних

Перша проблема яку потрібно вирішити – попередити невідповідності між схемою бази даних та типами, які представляють відповідні реляції з цієї схеми. Є два шляхи її вирішення:

- Код для визначення схеми бази даних генерується на основі набору типів мови Haskell
- Типи, що відповідають реляціям, генеруються на основі метаданих існуючої бази даних

Такий підхід дуже схожий на технологію Object-Relative Mapping (ORM), широко використовувану в об’єктно-орієнтованих мовах програмування. Але головна особливість ORM-систем полягає у тому, що маніпуляції над об’єктами типів, що відповідають реляціям, відображаються на відповідних записах у базі даних. Такий підхід не відповідає концепції Haskell як чисто функціональної мови. Тому, як ми побачимо у підрозділі 2.3.2, незважаючи на схожість з ORM-системами, усі маніпуляції з базою даних будуть явно відображатися в коді.

Template Haskell (TH) – розширення синтаксису мови Haskell, що дозволяє метапрограмування на етапі компіляції. З його допомогою можна писати метапрограми, результатом компіляції яких є програми мовою Haskell. Як правило, Template Haskell використовують або для генерації коду, або для розробки вбудованих предметно-орієнтованих мов [14].

Як інструмент генерації коду, TH має два основних компоненти – монада *Q* (з англ. – “quotation”), яка представляє ряд обрахунків, результатом яких є вираз мовою Haskell, та тип *Exp,* який представляє, власне, вирази мовою Haskell. *Exp* – алгебраїчний тип, кожен його конструктор відповідає певному типу виразу – змінна, літерал, кортеж, анонімна функція, зразок тощо.

Для розробки вбудованих предметно-орієнтованих мов, TH пропонує використати клас *QuasiQuoter*, задача якого – перетворити рядок на вираз

мовою Haskell. Наприклад, так виглядатиме визначення вбудованої мови XML та її використання:

```
{-# LANGUAGE TemplateHaskell            #-}
{-# LANGUAGE QuasiQuotes                #-}
-- визначення
xml  :: QuasiQuoter
xml  =  QuasiQuoter { quoteExp = parseXML }

-- використання
sampleXML :: XML
sampleXML = [xml | <root> val </root> ]
```

Одне з рішень для генерації типів Haskell за схемою бази даних пропонує бібліотека *persistent* – вона пропонує вбудований синтаксис *mkPersist*, призначений для опису сутностей бази даних [15]. І першочергово, цей інструмент генерує відповідні типи.

Розглянемо наступний приклад:

```
{-# LANGUAGE TemplateHaskell            #-}
{-# LANGUAGE QuasiQuotes                #-}
import Database.Persist.TH
import Data.Text (Text)

share [mkPersist sqlSettings] [persistLowerCase|
  Supplier
    telNum Text
    name Text
    city Text
    email Text Maybe
]
```

На основі цього коду *persistent* згенерує тип *Supplier*:

```
data Supplier = Supplier {
  supplierTelNum :: Text,
  supplierName :: Text,
  supplierCity :: Text,
  supplierEmail :: Maybe Text
}
```

Також цей тип буде зроблено екземпляром *PersistEntity* та згенеровано декілька допоміжних типів даних, що будуть детальніше розглянуті у розділі 2.4.

2.3.2 Генерація схеми бази даних на основі типів. Міграції

Розглянутого прийому генерації типів вбудованою предметно-орієнтованою мовою недостатньо для гарантування відповідності між схемою та набором типів. Необхідно також мати інструмент, який побудує коректну схему бази даних. Цей інструмент – міграції.

Міграції – це технологія, що дозволяє за наявною схемою бази даних та описом необхідних сутностей згенерувати SQL-код, що модифікує наявну схему, приводячи її у відповідність із заданими сутностями.

Бібліотека *persistent* надає Template Haskell функцію *mkMigrate*, яка за заданим описом сутностей генерує функцію мовою Haskell, яка виконує міграцію. Але перш ніж продемонструвати її можливості, необхідно повернутися до синтаксису *mkPersist*, який дозволяє не тільки визначати властивості типів Haskell (як було показано у підрозділі 2.2.1), а й реляції баз даних. Модифікуємо приклад з підрозділу 2.2.1:

```
share [mkPersist sqlSettings, mkMigrate "migrateDB"] [persistLowerCase|
  Supplier sql=suppliers
    telNum Text sql=tel_num sqltype=char(10)
    name Text sqltype=varchar(64)
    email Text Maybe sqltype=varchar(64)
    Primary telNum
  Tool sql=tools
    name Text sqltype = varchar(32)
    price Int
    supplierTelNum Text sql=supp_tel_num sqltype=char(10)
    Primary name
    Foreign Supplier fkSupp supplierTelNum
]
```

У прикладі вище ми бачимо визначення не тільки двох типів мовою Haskell, а й двох сутностей у базі даних. Явно вказані назви реляцій та назви атрибутів (за допомогою ключового слова *sql*), типи атрибутів (за допомогою ключового слова *sqltype*), первинний та зовнішній ключі (ключові слова *Primary* та *Foreign* відповідно).

Виклик функції *mkMigrate* з аргументом "migrateDB" на етапі компіляції згенерує функцію, що повертає екземпляр типу *Migration* – абстракцією над

процесом міграції. Бібліотека надає монадичну функцію *runMigration*, яка виконує задану міграцію.

Також є можливість побачити код мовою SQL, який буде згенеровано за заданою міграцією. Для приведеного вище визначення сутностей та бази даних з пустою схемою, матимемо наступний код:

```
CREATE TABLE "suppliers" (
    PRIMARY KEY ("tel_num"),
    "tel_num" char(10) NOT NULL,
    "name" varchar(64) NOT NULL,
    "email" varchar(64) NULL
);
CREATe TABLE "tools" (
    PRIMARY KEY ("name"),
    "name" VARCHAR NOT NULL,
    "price" INTEGER NOT NULL,
    "supp_tel_num" char(10) NOT NULL
);
ALTER TABLE "tools" ADD CONSTRAINT "toolfksupp"
FOREIGN KEY("supp_tel_num")
REFERENCES "suppliers"("tel_num");
```

Таким чином, *mkPersist* надає можливість підтримувати відповідність між схемою бази даних та набором типів даних Haskell за допомогою синтаксису визначення сутностей, за яким генеруються і типи, і схема.

2.4 Статична перевірка коректності запитів

Статична перевірка коректності запитів означає, що синтаксичні помилки та невідповідності між запитами та схемою викликатимуть помилки компіляції, а не помилки часу виконання. Єдиний спосіб гарантувати це – впровадити абстракцію над запитами до бази даних. Іншими словами, потрібні конструкції мовою Haskell, результат яких на етапі виконання можна перетворити на запити до СКБД.

Основним недоліком використання такого підходу є те, що насправді доволі складно передбачити, які саме конструкції будуть згенеровані на основі цих конструкцій.

Бібліотека *persistent* дає можливість використовувати такі конструкції.

По-перше, окрім типів, що відповідають сутностям бази даних *persistent* генерує ще ряд конструкцій [15]:

- Для кожної властивості таких типів генерується окремий екземпляр типу *EntityField,* що абстрагує атрибут реляції
- Для кожного такого типу генерується екземпляр типу *Key*, що абстрагує первинний ключ
- Кожен тип, що відповідає реляції стає екземпляром класу *PersistEntity*, що абстрагує сутність

По-друге, маємо набір інструментів, за допомогою яких можна зі згенерованих конструкції утворювати запити до бази даних:

- Монадичні функції, які відповідають SQL-інструкціям: *selectList*, *insert*, *updateWhere* тощо
- Набір комбінаторів, що відповідають SQL-операторам: *(.=)* – присвоювання, *(||.)* – оператор *OR*, (/<-.) – оператор *NOT IN*, *(<.)* – перевірка «менше ніж» тощо

За допомогою цих інструментів ми можемо писати конструкції мовою Haskell, які статично перевірятимуться на етапі компіляції та можуть бути перетворені у SQL-запити. Наприклад, так виглядатиме додавання нового інструмента у таблицю, приведену у розділі 2.2.2:

```
insertTool:: String -> String -> Int -> SqlPersistT (LoggingT IO) (Key Tool)
insertTool toolName supplierId price = insert $ Tool $ toolname supplierId
price
```

А таким чином можна отримати список усіх інструментів, які постачає певний постачальник, відсортований за ціною:

```
toolsOfSupplier :: Supplier -> SqlPersistT (LoggingT IO) [Entity Tool]
toolsOfSupplier supplier = selectList
                [ToolSupplierTelNum ==. (supplierTelNum supplier)]
                [OrderBy ToolPrice Desc]
```

Перший аргумент функції *selectList* – список екземплярів типу *Filter*, які при виконанні запиту перетворюються на частину *WHERE*, другий аргумент – список додатковий опцій вибору *SelectOpt*.

Таким чином, за результатом виклику функції вище згенерується наступний SQL-код, де *telnum* – результат обчислення виразу *(supplierTelNum supplier)*:

```
SELECT * FROM tools
WHERE supplierTelNum = telnum
ORDER BY price DESC
```

Серйозним недоліком *persistent* є те, що ряд СКБД, які він підтримує, включає також і нереляційні бази даних. Тому можливості цієї бібліотеки не покривають конструкції, специфічні для реляційних СКБД, такі як *JOIN* та *GROUP BY*.

3. Реалізація практичної частини

3.1 Опис програми та обґрунтування обраних інструментів

Для демонстрації розглянутих підходів та інструментів розроблено консольний застосунок для управління діяльністю готелю. Програма дозволяє створювати та редагувати бронювання, фіксувати оплату за цими бронюваннями, маніпулювати даними про наявний персонал, номери, клієнтів.

Для збереження даних обрано об’єктно-реляційну СКБД з відкритим вихідним кодом PostgreSQL. Дана система є доволі простою у розробці та підтримці, реалізує клієнт-серверну модель взаємодії. Для визначення та маніпуляції даними система використовує мову SQL, доповнену деякими об’єктними можливостями. Усе це робить СКБД PostgreSQL прийнятною для демонстрації особливостей роботи з реляційними базами даних.

Для маніпуляцій із базою даних обрано бібліотеку *postgresql-simple*. Це доволі простий та гнучкий інструмент, який дозволяє використовувати усі, як реляційні так і об’єктні, можливості обраної СКБД.

Для реалізації консольного інтерфейсу користувача використано стандартні інструменти Haskell, що обробляють введення-виведення, а саме модуль *System.IO*.

3.2 Розробка бази даних

Після аналізу предметної області було виділено ряд сутностей, основні з яких: клієнт, номер, бронювання, оплата та персонал, їх атрибути та зв'язки між ними. Результатом цього аналізу є ER-модель (див. Додаток А). Цю модель було переведено у реляційну (див. Додаток Б). Нижче приведено SQL-код, що створює таблиці Клієнти, Бронювання, Кімнати:

```
create table bookings
(
    book_num        serial          primary key,
    start_date      date            not null,
    end_date        date            not null,
    start_date_real timestamp,
    end_date_real   timestamp,
    booked_price    numeric(8, 2)   not null,
    price_period    numeric(9, 2)   not null,
    sum_fees        numeric(7, 2)   not null        default 0,
    payed           numeric(9, 2)   not null        default 0,
    book_comment    varchar(100),
    complaint       varchar(100),
    room_num        smallint        not null
                    references rooms(room_num)
                    on update cascade on delete restrict,
    cl_tel_num      char(13)        not null
                    references clients(tel_num)
                    on update cascade on delete restrict,
    book_state      booking_states not null         default 'booked'
);

create table clients
(
    tel_num         char(13)        primary key,
    passport        char(8),
    cl_name         varchar(15)     not null,
    surname         varchar(25)     not null,
    patronym        varchar(20)     not null,
    discount        smallint        not null        default 0
);

create table rooms
(
    room_num        smallint        primary key,
    room_floor      smallint        not null,
    room_places     smallint        not null,
    price           numeric(8, 2)   not null,
    type_name       varchar(100)    not null
                    references room_types(type_name)
                    on update cascade on delete restrict
);
```

У наведеному коді атрибут *book_state* має тип *booking_states* – не примітивний, визначений користувачем. Це тому, що *book_state* – рядковий атрибут, що позначає стан бронювання і існує обмеження предметної області: він може мати одне з чотирьох значень – «заброньовано», «скасовано», «заселено», «не заселено вчасно». У реляційній моделі є ще один схожий атрибут: тип оплати, який може мати значення або «готівкою», або «банківською карткою» У базі даних ці обмеження реалізовані за допомогою визначення користувацького типу-переліку. Вони аналогічні до типів-переліків enum, які підтримують ряд мов програмування. У Haskell вони можуть бути представлені типів-сум, що буде продемонстровано у підрозділі 3.3. У даному випадку на рівні бази даних домени цих атрибутів визначені наступним чином:

```
create type booking_states
as enum ('booked', 'settled', 'non_settle', 'canceled');

create type payment_types
as enum ('cash', 'card');
```

Ми маємо ще декілька обмежень предметної області, які варто контролювати на рівні бази даних. Атрибут *booked_price* реляції *bookings* повинен містити ціну за добу проживання у номері, яка була встановлена на момент вставки кортежу з цим бронюванням у базу даних. Це необхідно для того, щоб зміни у ціні на номер не впливали на суму до сплати за бронювання, які були створені до цих змін.

Також є обчислювані атрибути, наприклад атрибут *price_period* реляції *bookings*, що повинен зберігати загальну суму до сплати за проживання, не враховуючи штрафів, визначається як кількість днів у періоді між початковою та кінцевою датою бронювання, помножена на ціну за добу проживання у номері. А атрибути *sum_fees* та *payed* визначаються як сума розмірів усіх штрафів, що зареєстровані за цим бронюванням та сума усіх оплат, які за цим бронюванням проведено, відповідно.

PostgreSQL не має вбудованої підтримки обчислюваних атрибутів, але надає інструменти для визначення SQL-процедур та тригерів, за допомогою яких можна розробити механізм, який підтримуватиме цілісність даних при кожному їх оновленні.

У даному випадку нам необхідна процедура, яка знаходитиме відповідний кортеж реляції *rooms* та встановлюватиме значення атрибуту *booked_price* у значення атрибуту *price* у цьому кортежі, а також обчислювати значення *price_period*:

```
create function set_booked_price() returns trigger as
$$;
BEGIN
      NEW.booked_price :=
               (SELECT price FROM rooms
                WHERE room_num = NEW.room_num);
      NEW.price_period :=
                NEW.booked_price * (NEW.end_date - NEW.start_date);
      RETURN NEW;
END
$$;
```

Для процедур, що виконуються як тригер рівня запису при командах *INSERT* та *UPDATE*, PostgreSQL створює змінну *NEW*. Змінна має тип RECORD та зберігає кортеж, який вставляють, або кортеж, який буде результатом оновлення, відповідно. Користувацький код може читати та модифікувати значення атрибутів цього кортежу, як видно у коді вище.

Тепер необхідно зареєструвати тригер, що виконуватиме цю процедуру при вставці нового кортежу у реляцію *bookings:*

```
CREATE TRIGGER set_booked_price
BEFORE INSERT ON booking
FOR EACH ROW EXECUTE PROCEDURE set_booked_price();
```

Для підтримки цілісності атрибутів *sum_fees* та *payed* розроблено наступні тригери, що мають виконуватися при вставці кортежів у реляції Штрафи та Оплати відповідно:

```
create function increase_fees_sum() returns trigger as
$$
BEGIN
     UPDATE bookings
        SET sum_fees = sum_fees + NEW.price
        WHERE book_num = NEW.book_num;
     RETURN NEW;
END;
$$;

CREATE TRIGGER increase_fees_sum
BEFORE INSERT ON fees
FOR EACH ROW EXECUTE PROCEDURE increase_fees_sum();

create function increase_payed() returns trigger as
$$
BEGIN
      UPDATE bookings
         SET payed = payed + NEW.amount
         WHERE book_num = NEW.book_num;
      RETURN NEW;
END;
$$;

CREATE TRIGGER increase_payed
BEFORE INSERT ON payments
FOR EACH ROW EXECUTE PROCEDURE increase_payed();
```

3.3 Реалізація взаємодії з базою даних на мові Haskell

Перш за все, якщо у схемі бази даних використовуються користувацькі типи, необхідно реалізувати механізм їх конверсії у типи мови програмування, адже бібліотеки не зможуть забезпечити коректну роботу з ними. У випадку строго типізованої мови Haskell, робота з ними без гарантії типової безпеки завершуватиметься помилкою.

Отже, нам потрібні типи, що відповідатимуть перелікам *booking_states* та *payment_types*. Як було зазначено вище, для цієї задачі найкраще підійдуть типи-суми:

```
data PaymentType = Cash | Card
    deriving (Eq)

data BookingState = Booked | Settled | NonSettle | Cancelled
    deriving (Eq)
```

Отже, маємо типи, що явно відображають наші обмеження предметної області. Для зручності, варто зробити їх екземплярами класів *Read* та *Show* таким чином, щоб це відображало спосіб, у який відповідні значення зберігаються у базі даних. І якщо реалізація *Show* буде тривіальною, то для реалізації *Read* було додано допоміжну функцію *tryParse:*

```
tryParse :: String -> [(String, a)] -> [(a, String)]
tryParse _ [] = []
tryParse val ((attempt, result):xs) =
        if (take (length attempt) val) == attempt
            then [(result, drop (length attempt) val)]
            else tryParse val xs
```

Нижче наведено реалізацію *Read* та *Show* для *BookingState*, де перетворення у рядок та навпаки ставлять у відповідність значенню *BookingState* значення з визначення типу у базі даних:

```
booked, settled, non_settle, cancelled :: String -- booking_states values
booked = "booked"
settled = "settled"
non_settle = "non_settle"
cancelled = "cancelled"

instance Show BookingState where
    show Booked    = booked
    show Settled   = settled
    show NonSettle = non_settle
    show Cancelled = cancelled

instance Read BookingState where
  readsPrec _ value =
      tryParse value
           [(booked, Booked), (cancelled, Cancelled),
            (settled, Settled), (non_settle, NonSettle)]
```

Для того, щоб можна було вільно конвертувати значення цих типів у атрибути реляцій, залишилося реалізувати класи *FromField* та *ToField*, детально розглянуті у теоретичній частині. У нашому випадку це виглядатиме наступним чином:

```
import qualified Data.ByteString.Char8 as B
import qualified Data.ByteString.UTF8 as BSU

instance FromField BookingState where
    fromField f data =
        case fmap B.unpack data of
                Nothing  -> returnError UnexpectedNull f ""
                Just dat ->
                    case [ x | (x,t) <- reads dat, ("","") <- lex t ] of
                        [x] -> return x
                        _   -> returnError ConversionFailed f dat

instance ToField BookingState  where
    toField = Escape . BSU.fromString . show
```

Перетворення зі значення атрибуту виконується наступним чином: рядок байтів конвертується у звичайний рядок, далі у випадку пустого результату створюється виключна ситуація, яка повідомляє про те, що значення *NULL* неможливо перетворити у значення *BookingState*, а у випадку успіху, виконується спроба перетворити рядок у *BookingState* на основі реалізації класу *Read*.

Щоб перетворити *BookingState* на значення атрибуту, воно конвертується у рядок відповідно до реалізації *Show*, а цей рядок перетворюється у рядок байтів. Результат цього виразу «обгортається» у конструктор *Escape*, що означає, що при формуванні запиту, вихідний рядок необхідно заключити у одинарні лапки.

Тепер можна реалізовувати виконання запитів та обробку їх результатів. Запити у проекті винесено у окремий модуль *Queries*, який містить функції, що повертають значення типу *Query*. Так, наприклад, виглядає запит, що отримує список номерів, які будуть вільні у заданий період:

```
module Queries where

import Database.PostgreSQL.Simple
import Data.String

availiableRoomsQ :: Query
availiableRoomsQ = fromString $
    "SELECT room_num, room_floor, type_name, price " ++
    "FROM rooms " ++
    "WHERE NOT EXISTS
          (SELECT * " ++
          "FROM bookings " ++
          "WHERE rooms.room_num = room_num " ++
                "AND((?) BETWEEN start_date AND (end_date - integer '1') " ++
                "OR (?) BETWEEN start_date AND (end_date - integer '1') " ++
                "OR start_date BETWEEN (?) AND (?)" ++
                "OR (end_date - integer '1') BETWEEN (?) AND (?))" ++
                "AND book_state <> 'canceled') " ++
          "AND room_places >= (?) " ++
    "ORDER BY room_num;"
```

Функції введення-виведення, що безпосередньо взаємодіють з базою даних, зібрані у модулі *Repository*. Так виглядає функція, яка, використовуючи запит *availiableRoomsQ* отримує список номерів:

```
module Repository where

import Queries
import Data.Time.Calendar

withConn :: (Connection -> IO a) -> IO a
withConn action = do
    conn <- connectPostgreSQL connStr
    a <- action conn
    close conn
    return a

fetchAvailiableRooms :: Day -> Day -> Int -> IO [RoomDisplayModel]
fetchAvailiableRooms from to guests =
      withConn $  \conn -> query conn availiableRoomsQ
            (from, to, from, to, from, to, guests)
```

Аби використовувати тип *RoomDisplayModel* у цій функції, він має бути екземпляром *ToRow*. У даному випадку це реалізовано наступним чином:

```
data RoomDisplayModel = RoomDisplayModel {
      num :: Int
    , floor_ :: Int
    , type_ :: String
    , price :: Rational
}

instance FromRow RoomDisplayModel where
    fromRow = RoomDisplayModel <$> field <*> field <*> field <*> field
```

3.4 Реалізація консольного інтерфейсу користувача

Для взаємодії з користувачем впроваджено декілька типів даних, що відображають елементи управління:

```
data NextStep = Continue | Exit
    deriving Eq
type Prompt = String

type CanOperate a = (a -> Bool)

type Action = IO NextStep
type Choice = (Prompt, Action)

type BookingAction = BookingDisplayModel -> Action
type BookingChoice =
      (Prompt, BookingAction, CanOperate BookingDisplayModel)
```

Кожна дія користувача – це операція введення-виведення, результатом якої є мітка наступного кроку – вийти чи продовжити. Вибір користувача складається з рядка запрошення та дії, яка має виконатися при цьому виборі. Окремо визначено дію над бронюванням – дія, аргументом якої є значення типу *BookingDisplayModel*. Відповідно, окреме визначення є і для користувацького вибору дії на бронюванням – окрім рядка запрошення та дії, воно містить предикат, який позначає, чи застосовна дана дія до даного бронювання.

Основний цикл застосунку полягає у тому, що користувач отримує список дій, що він може виконати, дія виконується, а залежно від її результату програма або завершує роботу, або виконується заново:

```
main :: IO ()
main = do
    act <- choose choices fst "Select action:"
    next <- snd act
    if next == Continue then main else return ()
```

Тут *choices* – глобально визначений список доступних виборів користувача, що буде наведений далі, а *choose* – функція-утиліта, що за списком значень дає користувачу можливість обрати одне з них.

Як приклад дії користувача, розглянемо операцію додавання нового бронювання:

```
addBooking :: Action
addBooking = do
    putStrLn "Enter From date (YYYY-MM-DD):"
    from <- readDay
    putStrLn "Enter To date (YYYY-MM-DD):"
    to <- readDay
    putStrLn "Enter number of guests:"
    guests <- readInt
    rooms <- fetchAvailiableRooms from to guests
    case rooms of
        [] -> putStrLn "No free rooms for this period" >> pure Continue
        _ -> do
            room <- chooseShow rooms
                    "We have these rooms free for that days. Choose one:"
            putStrLn "Enter Comment:"
            comm <- getLine
            cls <- fetchClients
            cl <- chooseShow cls "Here are existing clients. Choose one:"
            createBooking from to comm (num room) (tel_num cl)
            return Continue
```

Адміністратор вводить дати, на які йому необхідно забронювати номер, та кількість гостей. Відповідно до цього з бази даних отримуються номери, вільні на цей період та такі, що можуть вмістити необхідну кількість гостей. У випадку, якщо такі є, адміністратор обирає клієнта зі списку та залишає коментар. Після цього викликається функція створення нового бронювання з модуля *Repository*, результат дії – мітка «Продовжити».

Аби додати цю дію до інтерфейсу користувача, треба додати відповідний вибір у список *choices:*

```
choices :: [Choice]
choices =
   [("Operate Bookings",        operateBookings),
    ("Add Booking",             addBooking),
    ("Show client analytics",   showClientAnalytics),
    ("Show debpts",             showDebpts),
    ("Show payments",           showPayments),
    ("Add client",              addClient >> pure Continue),
    ("Exit",                    pure Exit)]
```

Для операцій над бронюваннями маємо окрему дію – *operateBookings*. Вона виглядає наступним чином:

```
operateBookings :: Action
operateBookings = do
    putStrLn "Enter From date (YYYY-MM-DD):"
    from <- readDay
    return ()
    putStrLn "Enter To date (YYYY-MM-DD):"
    to <- readDay
    bs <- fetchBokings from to
    case bs of
        [] -> putStrLn "No booking for this period" >> pure Continue
        _ -> do
            b <- chooseShow bs "Select booking to operate: "
            let ops = filter
                  (\o -> bookingChoiceCanOperate o $ b) bookingChoices
            case ops of
                [] ->
                  putStrLn "Nothing to do with this booking" >>
                  pure Continue
                _ -> do
                    op <- choose ops bookingChoiceName "Select operation"
                    _ <- bookingChoiceAction op $ b
                    return Continue
```

Після введення періоду часу, користувач отримує список бронювань, що з ним перетинаються. Обравши певне з них, отримує список дій над бронюваннями, що для нього доступні. Далі ця дія виконується. Як приклад, можна розглянути проведення оплати за бронюванням:

```
payAct :: BookingAction
payAct b = do
    putStrLn "Enter sum:"
    sum <- readDouble
    if sum <= (fromRational $ debpt b)
        then do
            ptype <- chooseShow [Cash, Card] "Select payment type:"
            _ <- pay (book_num b) ptype sum
            return Continue
        else do
            putStrLn $ "Too much: debpt is just " ++ (show $ debpt b)
            payAct b
```

Адміністратор вводить суму та тип оплати, після перевірки введених даних відбувається додавання нового рядка до таблиці з оплатами.

Щоб ця дія з’явилася в інтерфейсі користувача, її необхідно додати до списку *bookingChoices* з відповідним рядком запрошення та предикатом, який перевірятиме, чи є у цього бронювання ненульовий борг:

```
bookingChoices :: [BookingChoice]
bookingChoices =
   [("Cancel", cancelBookingAct, (== Booked) . book_state),
    ("Mark as Settled", settleBookingAct, (== Booked) . book_state),
    ("Add payment", payAct, (> 0) . debpt),
    ("Add fee", feeAct, (== Settled) . book_state)]
```

Висновки

В рамках роботи було розглянуто ряд задач, які виникають при роботі з базами даних, зокрема у Haskell, та підходи до їх вирішення. Було проаналізовано переваги та недоліки взаємодії з СКБД на рівні текстових запитів та явної конвертації їх результатів у типи даних, з одного боку, та високорівневої роботи з базами даних, з іншого.

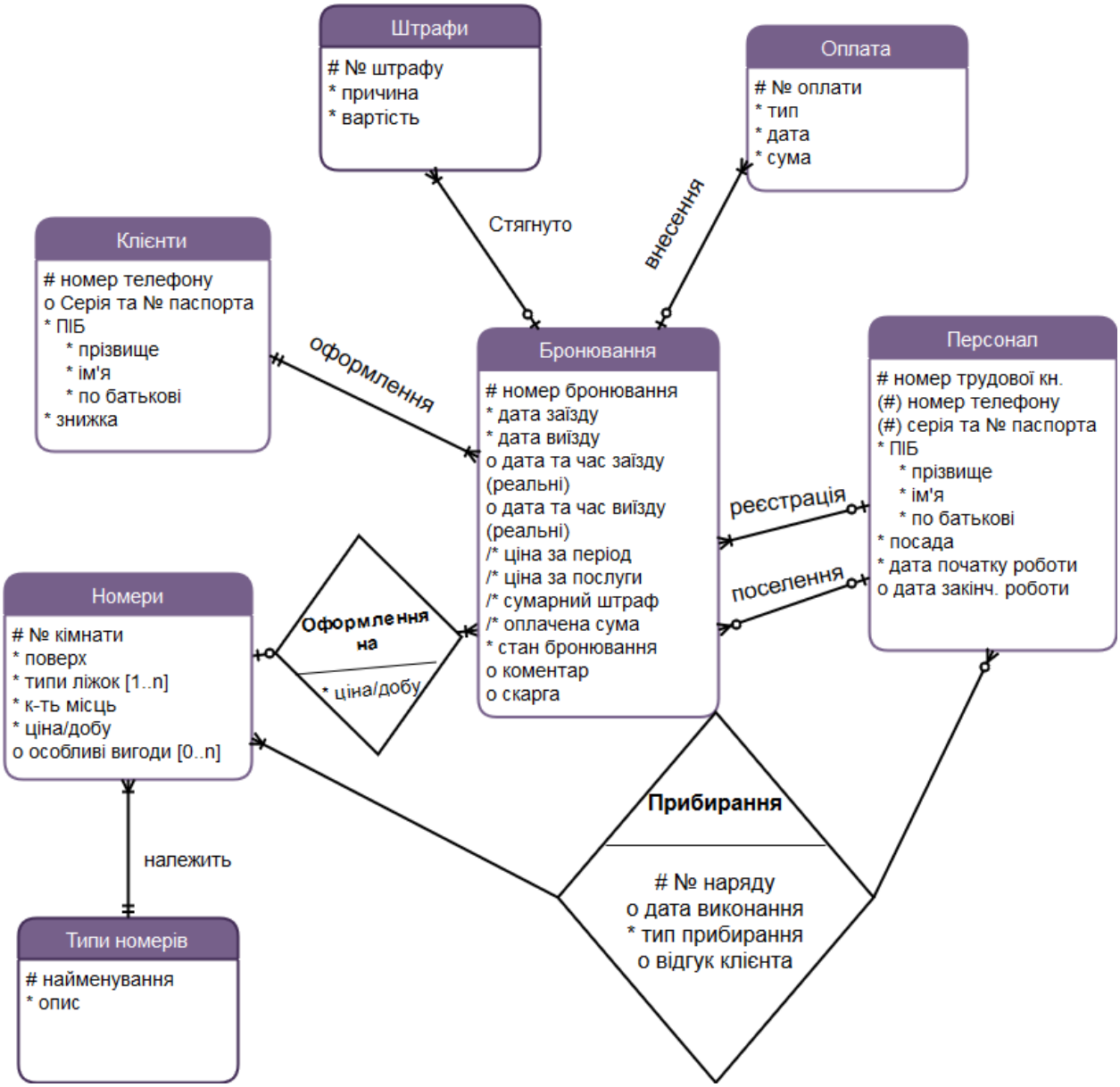
Розглянуто ряд інструментів мови Haskell, які дозволяють реалізувати ці підходи: монади, класи типів, розширення синтаксису Template Haskell. Окрім того, у теоретичній частині розглянуто базові операції при роботі з базами даних: підключення, транзакції.

За результатами дослідження реалізовано консольне застосування на мові Haskell, що взаємодіє з об’єктно-реляційною СКБД PostgreSQL за допомогою бібліотеки *postgresql-simple*.

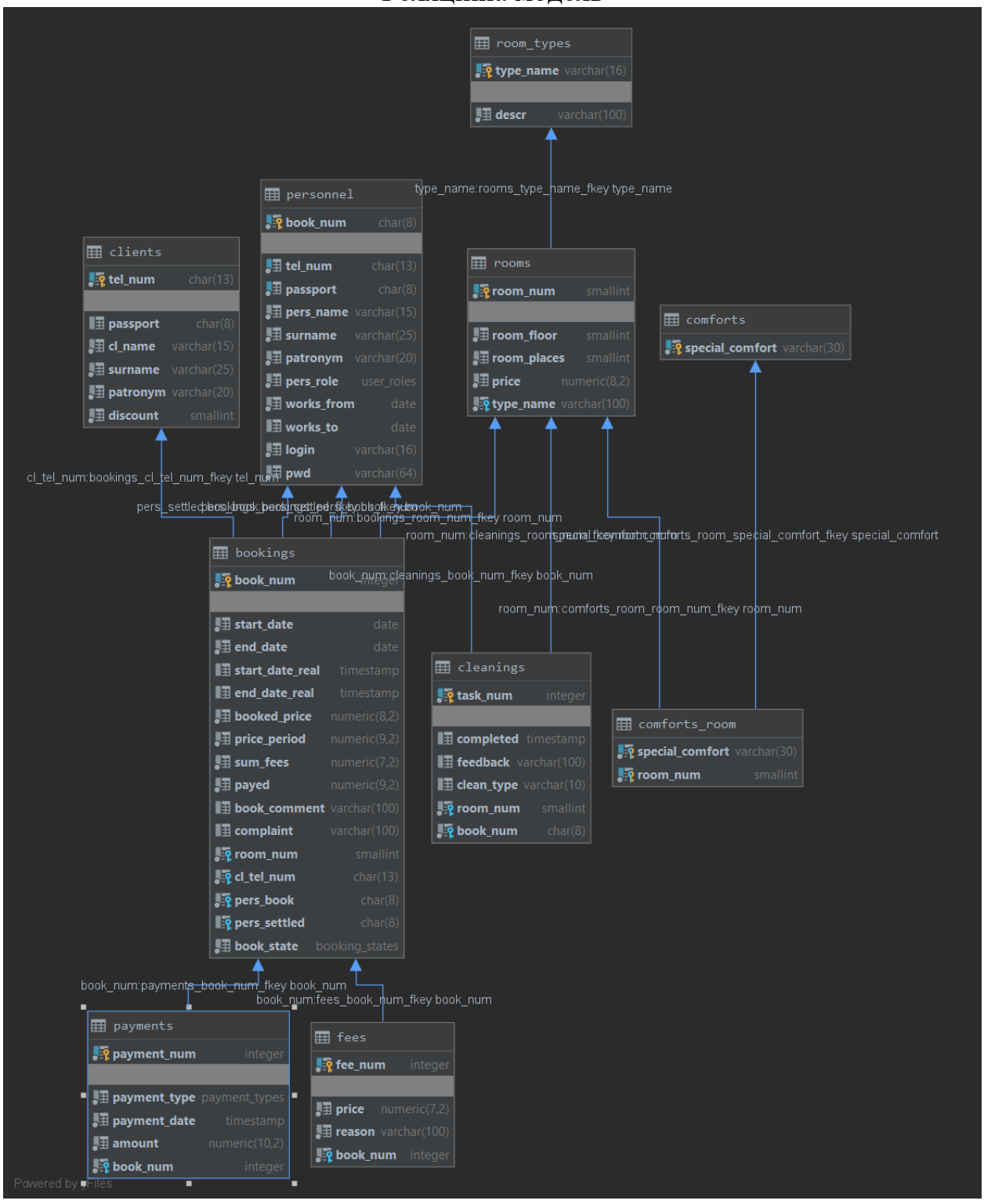
Список джерел

1. What is a Purely Functional Language? Автор А. Себрі. Рік видання – 1998.
2. Learn You a Haskell for Great Good. Автор М. Липовача. Рік видання – 2011.
3. IO inside [Електронний ресурс]: wiki.haskell.org/IO_inside
4. Database systems: a pragmatic approach. Автори Е. Фостер, С. Гадбоул. Рік видання – 2016.
5. An Introduction to Database Systems. Автор К. Дейт. Рік видання – 2003.
6. Replication, Clustering, and Connection Pooling [Електронний ресурс]: http://wiki.postgresql.org/wiki/Replication,_Clustering,_and_Connection_Pooling
7. The Transaction Concept: Virtues and Limitations. Автор Д. Грей. Рік видання – 1981.
8. Beginning Haskell: a project-based approach. Автор А. С. Мена. Рік видання – 2014.
9. sqlite-simple: Mid-Level SQLite client library [Електронний ресурс]: https://hackage.haskell.org/package/sqlite-simple
10. Get programming with Haskell. Автор У. Курт. Рік видання – 2018.
11. Real World Haskell: code you can believe in. Автори Б. Саліван, Дж. Горзен, Д. Стюарт. Рік видання – 2008.
12. HDBC: Haskell Database Connectivity [Електронний ресурс]: https://hackage.haskell.org/package/HDBC
13. Pattern Languages of Program Design 5: (Software Patterns). Автор Д. Манолеску. Рік видання – 2006.
14. Template Meta-programming for Haskell. Автори Т. Шерд, С. П. Джонс. Рік видання – 2002.
15. persistent: Type-safe, multi-backend data serialization. [Електронний ресурс]: https://hackage.haskell.org/package/persistent

Додаток А
ER модель



Додаток Б
Реляційна модель



Додаток В

Модуль із запитами до бази даних

```
module Queries where

import Database.PostgreSQL.Simple
import Data.String

bookingsQ :: Query
bookingsQ = fromString $
    "SELECT rooms.room_num, room_floor, start_date, end_date, book_state, boo
k_num, " ++
    "(price_period * (1 - (" ++
        "SELECT discount " ++
        "FROM clients " ++
        "WHERE tel_num = B.cl_tel_num)/100.0) + sum_fees-payed) AS debt " ++
    "FROM ( " ++
        "select * " ++
        "FROM bookings " ++
        "WHERE (end_date BETWEEN (?) AND (?) " ++
        "OR start_date BETWEEN (?) AND (?)) " ++
        "AND book_state <> 'canceled' " ++
            ") AS B INNER JOIN rooms ON B.room_num = rooms.room_num " ++
    "ORDER BY room_num, start_date; "

availiableRoomsQ :: Query
availiableRoomsQ = fromString $
    "SELECT room_num, room_floor, type_name, price " ++
    "FROM rooms " ++
    "WHERE NOT EXISTS(SELECT * " ++
                    "FROM bookings " ++
                    "WHERE rooms.room_num = room_num " ++
                                "AND((?) BETWEEN start_date AND (end_date - i
nteger '1') " ++
                                    "OR (?) BETWEEN start_date AND (end_date
- integer '1') " ++
                                    "OR start_date BETWEEN (?) AND (?)" ++
                                    "OR (end_date - integer '1') BETWEEN (?)
AND (?))" ++
                            "AND book_state <> 'canceled') " ++
            "AND room_places >= (?) " ++
        "ORDER BY room_num;"

analyticsQ :: Query
analyticsQ = fromString $
    "SELECT clients.surname, clients.discount, " ++
            "COUNT(bookings.book_num) as count_booked, " ++
            "coalesce(SUM(bookings.payed), 0) as sum_payed " ++
    "FROM clients LEFT OUTER JOIN bookings ON bookings.cl_tel_num = clients.t
el_num " ++
```

```
    "GROUP BY clients.tel_num, clients.cl_name, clients.surname, clients.disc
ount " ++
    "ORDER BY sum_payed DESC; ";

debptsQ :: Query
debptsQ = fromString $
    "SELECT * " ++
    "FROM(SELECT book_num, clients.tel_num, surname, room_num, " ++
                "((price_period + sum_fees) * (1 - clients.discount / 100.0))
 - payed AS debpt " ++
        "FROM bookings INNER JOIN clients ON bookings.cl_tel_num = clients.te
l_num " ++
        "WHERE bookings.book_state <> 'canceled') AS X " ++
    "WHERE debpt > 0;"

createBookingQ :: Query
createBookingQ = fromString $
    "INSERT INTO bookings (start_date, end_date, book_comment, room_num, cl_t
el_num, pers_book) " ++
    "VALUES ((?), (?), (?), (?), (?), 'KB111111');"

cancelBookingQ :: Query
cancelBookingQ = fromString $
    "UPDATE bookings " ++
    "SET book_state = 'canceled' " ++
    "WHERE book_num = (?);"

settleQ :: Query
settleQ = fromString $
    "UPDATE bookings " ++
    "SET book_state = 'settled' " ++
    "WHERE book_num = (?);"

clientsQ :: Query
clientsQ = fromString $
    "SELECT tel_num, surname, discount " ++
    "FROM clients"

payQ :: Query
payQ = fromString $
    "INSERT INTO payments (payment_type, payment_date, amount, book_num) " ++
    "VALUES ((?), current_timestamp, (?), (?))"

feeQ :: Query
feeQ = fromString $
    "INSERT INTO fees (price, reason, book_num) " ++
    "VALUES ((?), (?), (?))"

createClientQ :: Query
createClientQ = fromString $
```

```
    "INSERT INTO clients " ++
    "VALUES ((?), (?), (?), (?), (?), (?));"

paymentsQ :: Query
paymentsQ = fromString $
    "SELECT payment_type, amount, room_num " ++
    "FROM payments INNER JOIN bookings ON payments.book_num = bookings.book_n
um " ++
    "WHERE date(payments.payment_date) = (?);"
```