Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 21.Б11-мм

ORM-библиотека для OCaml

СТАРЦЕВ Матвей Сергеевич

Отчёт по учебной практике
в форме «Производственное задание»

Научный руководитель:
ассистент каф. СП Косарев Д.С.

Санкт-Петербург
2024

Оглавление

Введение

Хранение и обработка внешних данных — ключевые процессы в программировании. Без них программы ограничены в возможностях и той пользе, которую они могут принести пользователю, так как в большинстве своём статичны.

Зачастую внешней информации очень много, вследствие чего сложно её хранить, организовывать и обрабатывать. При этом время жизни этих данных больше времени жизни работающего с ними приложения, а в некоторых случаях доступ к ним должен быть ещё и безопасным. Одним из решений этих проблем являются реляционные базы данных[4].

Работа с ними происходит с помощью декларативного языка SQL[2]. Однако при комбинировании его с исходным кодом приложения, написанного на каком-то из языков программирования, разработчики зачастую встречаются с рядом трудностей, такими как: маловыразительные и со своими особенностями собственные представления и типы данных, различные диалекты самого языка работы с базами данных и использование «магических» строк в коде для составления запросов.

Существует подход, который борется с описанными выше проблемами — ORM (Object–relational mapping) [10]. Он предлагает выражать все SQL-запросы и интерпретировать их результаты с помощью конструкций языка исходного кода. Зачастую этот подход реализуют ORM-фреймворки, например, LINQ2DB [3], SQLAlchemy [17], Hibernate [15], которые выступают прослойкой между кодом и базами данных, генерируют SQL-запросы самостоятельно, а полученные данные транслируют в объекты языка.

Приложения на OCaml ничем не хуже других, написанных на иных языках программирования, и тоже работают с внешними данными и базами данных, для чего сообщество создало ряд библиотек и расширений[1]. Однако в большинстве своём они узкоспециализированы под одну из баз данных, предлагают программисту писать и вызывать SQL-запросы внутри и из кода. Не все из них учитывают специфику языка

[1]https://ocamlverse.net/content/databases.html Дата обращения: 24.12.2023

и ни одна в полной мере не реализует ORM-подход.

Таким образом, возникает задача реализации ORM-библиотеки для языка программирования OCaml, учитывающей его особенности, о которых речь пойдёт позже, и поддерживающей работу с несколькими типами баз данных.

Дата сборки: 3 июня 2024 г.

1. Постановка задачи

Целью данной учебной практики является создание ORM-библиотеки для языка OCaml.

В предыдущей работе на данную тему[2] уже было реализованно следующее:

1. базовые операции работы с данными;
2. отображение простых типов;
3. связь один-ко-одному.

Имплементация следующих вещей стала новой задачей для выполнения поставленной цели:

1. безопасные по типам условия для запросов;
2. агрегаторы и группирующие функции;
3. связи один-ко-многим и многие-ко-многим;
4. инструменты для миграции.

[2]https://github.com/Tozarin/courseworks/blob/main/ORM/First-part-ORM.pdf Дата обращения: 03.06.2024

2. Обзор

2.1. ORM-фреймворки

Обзор ORM-фреймворков проводился с целью выявления общепринятой функциональности и некоторых особенностей её реализации.

В первую очередь были исследованы библиотеки для языка Scala, так как он во многом схож с OCaml: тоже функциональный и тоже располагает инструментарием для описания объектов, а также в достаточной мере популярен, чтобы для него существовали поддерживаемые и общепринятые решения.

В силу того, что большинство ORM-библиотек для объектно-ореинтированных языков во много схожи между собой была рассмотрена только одна из них.

2.1.1. Squeryl

Squeryl[14] — Scala ORM-фреймворк, который позволяет формировать запросы с помощью безопасных по типам функций, и отображает результаты запросов пользователя в объекты и типы языка.

Листинг 1: Выборка записей по переданному id

```
def songs = from(MusicDb.songs)
    (s => where(s.artistId === id) select(s))
```

Отсутствие «голых» строк, представляющих SQL-запросы, исключает возможность ошибиться в синтаксисе запроса и позволяет протипизировать всё выражение, гарантируя тем самым его семантическую корректность.

Для начала работы Squeryl требует описания схемы базы данных через исходный код на языке Scala с использованием классов, которые в последствие должны быть привязаны к библиотечному значению, отвечающему за обращение к базе.

Листинг 2: Объявление таблицы Movie

```
class Movie(
    val imdbId: String,
    val year: Int,
    val name: String
)
val movies = table[Movie]("MOVIES")
```

Внутри классов могут быть описаны значения:

1. базовых типов, возможно внутри option, которые отображаются напрямую в отдельные SQL-типы;

2. перечислений, которые отображаются в целочисленные поля;

3. пользовательских типов, но с условием, что для них определены в соответствии с некоторым соглашением часть функций;

4. других классов или перечислений классов, участвующих в отображении. Для этого создаются промежуточные таблицы, описывающие one-to-many и many-to-many отношения.

В итоге каждому полю из класса соответствует поле такого же имени в таблице, имя которой — имя связанного класса. На некоторые аспекты этого отображения возможно повлиять с помощью аннотаций, например, задать другое имя для поля в базе или изменить максимальную длину строки для него.

При этом обязательно должно быть описано поле, которое однозначно индефицирует объект. Это может быть сделано либо с помощью аннотаций, либо наследуя класс библиотеки, отвечающий за это.

Классам предоставляются функции для манипуляций с данными в базе: insert, update, delete, каждая из которых воздействует на записи, соответствующие объекту, множеству объектов или удовлетворяющие переданному запросу, формируя один относительно оптимальный SQL-запрос. Зачастую каждое ключевое слово из запроса на Scala напрямую транслируется в соответствующее на SQL.

Объекты из базы данных могут быть получены через select или join функции, принимающие условие, по которому отбираются записи. Оно составляется из лямбда-функций и логических функций, например, exist или any, в нём могут участвовать поля, задействованных классов, базовые операторы сравнения, константы и подзапросы.

Возвращаемые значения и их типы могут быть изменены, если применить к запросу агрегирующую или группирующую функции. Также может быть поменян их порядок с помощью orederby. Как и для описанных выше операций, составляемые выражения транслируются в SQL практически напрямую.

Главной особенностью Squeryl разработчики выделяют «ленивость» формирования запросов, что позволяет выбирать оптимальные сценарии их исполнения, например, при возникновении N+1 проблемы[3], для неё фрейморк может составить один SQL-запрос, а затем из полученных данных сложно формировать объекты, либо несколько, которые создадут нагрузку на базу, но облегчат интерпретацию результатов.

2.1.2. Slick

Slick[13] — Scala решение для работы с базами данных. Разработчики определяют его не как ORM, а как FRM (Functional Relational Mapping) в силу своей философии. Их целью при его создании, как они утверждают, являлось создание такой библиотеки, которая бы позволила писать функциональный код (в смысле парадигмы) с сохранением всех присущих аспектов, например, системы типов или отсутствием побочных эффектов (как следствие Slick ничего не кэширует), и при этом обладать полным контролем над самой базой данных как будто бы пользователь работает с ней напрямую через SQL.

В следствие такой философии разработчик получает более гибкий инструмент для работы в обмен на дополнительную нагрузку при написании кода и отображении данных в сравнении с традиционными решениями.

[3]https://www.baeldung.com/spring-hibernate-n1-problem Дата обращения: 06.05.2024

Листинг 3: Объявление таблицы Coffees

```
class Coffees(tag:Tag) extends Table[Coffee](tag,"COFFEES"){
    def name  = column[String]("NAME")
    def price = column[Double]("PRICE")
    def * = (name, price).mapTo[Coffee]
}
val coffees = TableQuery[Coffees]
```

Схема базы данных определяется через расширение пользовательским классом библиотечного с явным указанием всех полей, их имён, типов и ограничений в базе данных. Все дополнительные таблицы, например, для сложных связей между данными пользователь должен описывать самостоятельно. Идентификация записей также ложится на пользователя, однако она может быть упрощена через специальные ограничения на поля, которые будут автоматически инкриминировать значения для новых записей. В схеме могут быть описаны все базовые типы, которые напрямую, как и многое в Slick, перекладываются на SQL.

Для манипуляций с данными Slick определяет списки, каждый из которых соответствует отдельной таблице, и переопределяет функции работы с ними.

Листинг 4: Простые операции в Slick

```
coffees += Coffee("Latte", 2.50)
coffees.map(_.name)
```

Так вставка реализуется через оператор добавления элемента: +=, а выборка через filter. Через эти функции, которые соответствуют функциональному стилю, можно выражать все операции SQL, чего и добивались разработчики.

Каждый составленный запрос не исполняется сразу, а только по вызову специальной функции, это позволяет объединять вложенные условия, чтобы в дальнейшем сгенерировать достаточно оптимальный единственный SQL-запрос, и переиспользовать уже составленные.

В Slick определены операторы и функции, участвующие в составлении условий запросов, агрегаторы и группирующие функции, все они напрямую транслируются в SQL. Возвращаемые значения, а точнее то, какие поля и в каком порядке они должны быть представлены, определяет пользователь.

В итоге, как было описано выше, Slick в следствие философии разработчиков представляет собой функциональное переложение SQL на язык Scala без крупной потери функциональности с минимальными оптимизациями. Пользователь получает полный контроль над базой данных, все действия над ней для него прозрачны и однозначны, но требуют больших усилий нежели, например, при использовании Squeryl.

2.1.3. Entity Framework

Entity Framework[9] — крупный C# ORM-фреймворк организующий работу с базой данных и конвертацию результатов запросов в типы C# (преимущественно объекты) и обратно. В отличие от описанных выше библиотек Entity Framework представляет собой не только множество функций для работы с данными, но и различные решения по организации таблиц, отображений, миграций, которые зачастую реализуются вне кода.

Так, схема базы данных и модель могут быть описаны различными способами:

1. с помощью других средств база может быть уже организована, тогда от пользователя требуется описать классы, соответствующие таблицам, и определить классы-контексты для работы с каждым из них, а фреймворк самостоятельно решит, как отображать данные;

2. внутри кода пользователь может описать классы и с помощью аннотаций сконфигурировать часть полей. Эта модель будет использована для создания схемы базы данных.

Так как модель может быть получена из схемы и схема из модели, все

SQL-типы могут быть отображены в C#-типы и наоборот с сохранением всех ограничений, которые в коде задаются с помощью аннотаций, и связей.

Листинг 5: Выборка по условию

```
using(ApplicationContext db = new ApplicationContext()) {
    var users = db.Users.Where(p => p.Company!.Name=="Google");
    foreach (User user in users)
        Console.WriteLine("{user.Name}:({user.Age})");
}
```

Выполнение запросов должно происходить обязательно внутри конструкции using с использованием класса-контекста. Только там могут производиться все манипуляции над данными, а также их выборка.

Entity Framework поддерживает все базовые операции с данными, которые задаются с помощью отдельных методов классов-таблиц, либо с участием LINQ-операторов или методов расширения. Каждый из которых представляет аналог ключевого слова из SQL, принимающий объекты для добавления или условия для выборки и модификации, заданные, например, с помощью лямбда-функций. В рамках одного запроса на C# формируется один SQL-запрос.

Фреймворк позволяет использовать почти все операторы сравнения, базовые функции, агрегаторы и группирующие функции SQL внутри запросов.

Большая часть возвращаемых объектов кэшируется, что облегчает нагрузку на базу данных, так как в случае повторения запроса или его части, объекты могут быть возвращены мгновенно, минуя физический запрос и отображение в объект.

Помимо базовой функциональности Entity Framework предоставляет инструменты для миграции баз данных. Они могут быть осуществлены как с помощью среды разработки, так из кода. Для этого необходимо изменить модель и определить класс, который наследуется от библиотечного класса миграции, а в нём методы up и down, описывающие изменения с базой данных. Возможные изменения отвечают

функциональности выражения ALTER в SQL, то есть, например, могут быть добавлены новые таблицы или изменены ограничения на поля. Все новые добавленые поля инициализируются NULL.

Как было описано выше, Entity Framework — крупный индустриальный фреймворк позволяющий организовывать работу с базой данных различными способами как на этапе инициализации, так и при работе с данными.

2.2. SQL библиотеки для OCaml

Обзор OCaml библиотек для работы с SQL проводился для того, что бы актуализировать важность данной работы и обосновать ряд принятых решений, связанных с общим видением проекта. Поэтому он не столь подробен и затрагивает только базовую функциональность без конкретных деталей реализации.

При этом стоит уточнить, что выбор одной из них для работы с базами данных был обоснован в предыдущей работе.

Были рассмотрены популярные (с более чем 100 звёздами на github) библиотеки, а также схожие либо аналогичные решения.

2.2.1. Caqti и ppx_rapper

Caqti[1] — монадическая типобезопасная библиотека, которая позволяет работать с PostgreSQL, SQLite, MariaDB.

Её интерфейс обязывает описывать типы пользовательских запросов, чтобы затем проверять их на корректность в коде. Сами же запросы внутри кода реализуются как строки.

Для облегчения написания сопутствующего кода используется препроцессорное расширение ppx_rapper[16], которое на этапе компиляции по статическому запросу генерирует необходимые функции и структуры, что обеспечивает его синтаксическую и типовую проверку.

2.2.2. PG‘OCaml и PPX

PG‘OCaml[6] — PostgreSQL клиент, написанный на чистом OCaml без использования языка C. Он позволяет создавать запросы-структуры, которые представляют собой «магические» строки с некоторыми пропущенными значениями. В дальнейшем они могут быть переиспользованы уже с конкретными данными.

Аналогично Caqti для PG‘OCaml реализованно препроцессорное расширение PPX, валидирующее статические запросы и способное специфицировать, в каком виде должны быть возвращены строки: записи, тюплы и тому подобное.

2.2.3. PGX

PGX[7] — несостоявшееся, в том смысле, что это отдельная библиотека, хотя и во многом схожая, ответвление от PG‘OCaml. Подобно последней это PostgreSQL клиент, реализованный на OCaml.

Основным же её отличием является то, что она направленна на использование без препроцессорных расширений[4]. В связи с чем для поддержания типовой безопасности обязывает подобно Caqti описывать типы запросов, которые представлены как строки.

2.2.4. SQLite3-OCaml

SQLite3-OCaml[12] — OCaml SQL-клиент для SQLite3. Использует «магические» строки для запросов. Не предоставляет возможностей как-либо их проверять.

2.2.5. orm

orm[8] — ORM-библиотека для OCaml над SQLite3, использует записи для описания схемы базы данных и для работы с результатами запросов.

[4]https://discuss.ocaml.org/t/ann-first-release-of-pgx-pure-ocaml-postgresql-client/2068/6 Дата обращения: 03.06.2024

Она типобезопасная в том смысле, что созданное для чтения подключение не позволяет на уровне типов вносить изменения в базу данных и тому подобное.

По выбранным пользовательским типам-записям генерируется дополнительный необходимый код для отображения и работы с базой данных. При этом исключаются запросы непосредственно на SQL в коде, ограничивая гибкость: пользователю предлагается выбирать записи только по условиям на конкретные поля, которе не могут содержать ничего кроме базовых операторов и констант и в целом круг возможностей ограничен.

Также стоит отметить, развитие и поддержка библиотеки остановились семь лет назад, в личной переписке с автором выяснилось, что это произошло из-за устаревания camlp4[18] — одной из технологий, от которой сильно зависит библиотека.

2.2.6. DBCaml

DBCaml[11] — новая библиотека, возникшая сравнительно недавно (3-его мая 2024-го года[5]). Реализует сериализацию и десериализацию пользовательских записей для использования при составлении запросов.

Запросы реализуются как «магические» строки и обязаны сопровождаться дополнительным кодом, который описывает преобразования данных из базы данных в OCaml и обратно.

2.3. Выводы

Большинство OCaml библиотек для работы с базами данных предлагают пользователю самостоятельно писать SQL-запросы, а затем проверять их с использованием некоторых инструментов. Однако они применимы только к статическим запросам, так как работают во время компиляции.

[5]https://priver.dev/blog/dbcaml/dbcaml-project/ Дата обращения: 03.06.2024

То есть эти решения дают гибкость при работе, не давая полной гарантии безопасности в случае составления запросов непосредственно во время работы приложения.

При этом стоить отметить, что библиотек, предлагающих отображение составных типов, будь то записи или модули, сравнительно немного. И они либо не поддерживаются, либо не позволяют валидировать запросы.

В связи с чем было принято решение реализовывать собственную библиотеку подобно Squery, то есть с применением базового ORM подхода. Максимально упрощая опыт использования и гарантируя корректность работы, жертвуя гибкостью и возможностями при работе. Так как подобных решений не было обнаружено.

3. Реализация

Основой библиотеки стали first-class модули, которые позволяют агрегировать в себе именованные поля различных типов. В следствие чего для неё было выбрано имя Mrm.

3.1. Генерация кода

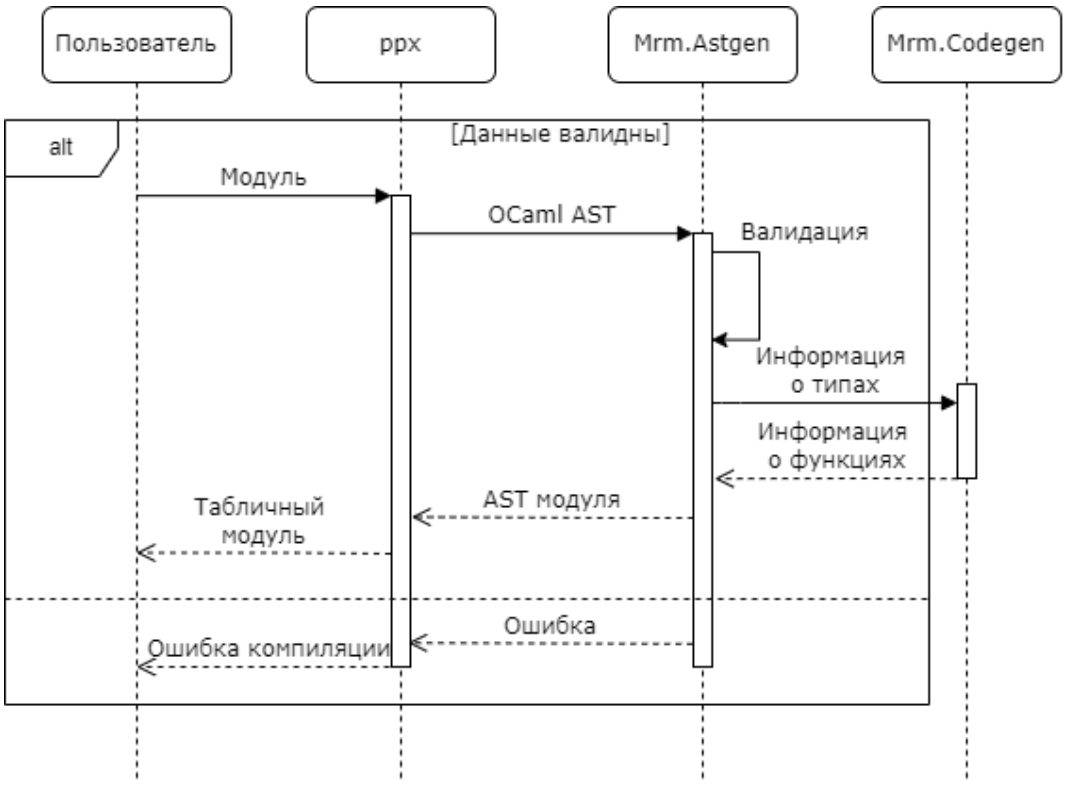

Рис. 1: Схема генерации

Для кодогенерации была выбрана стандартная библиотека препроцессинга — ppxlib[5], она позволяет определять собственные директивы, которые по общему контексту кода могут изменять дерево абстрактного синтаксиса OCaml-программы во время компиляции.

Для тех пользовательских first-class модулей, которые содержат включение библиотечного модуля Uuid, необходимого для идентификации, и отмечены библиотечной директивой mrm, генерируются отдельные модули для запросов к базе данных вне зависимости от её типа.

Листинг 6: Модуль для отоброжения в таблицу

```
module type Point = sig
    include Uuid

    val x : int
    val y : int
end
[@@deriving mrm]
```

Это происходит следующим образом (Рис. 1). Препроцессор предоставляет информацию об отмеченном модуле фукнции-расширению, объявленной в библиотеке.

На первом этапе эта функция валидирует модуль, проверяя, что поля допустимых типов и что модуль содержит включение модуля Uuid. В случае неудачи возвращается ошибка компиляции.

Полученная часть дерева абстрактного синтаксиса конвертируется во внутренние типы, которые содержат всю необходимую информацию для генерации кода. Это необходимо для того, чтобы унифицировать имена, а также иметь возможность вносить изменения с минимальными усилиями, так как вся необходимая информация собирается и заносится в единственном месте в коде.

По полученным внутренним структурам генерируются OCaml код, который представлен на данном этапе в виде строк. Затем он передаётся функции OCaml-парсера для получения части дерева абстрактного синтаксиса. Генерация может быть осуществлена сразу во второе представление, однако на данном этапе развития библиотеки, когда иногда приходится вносить крупные изменения в создаваемый код, проще работать со строками нежели с деревом OCaml. Опасность же генерации синтаксически некорректного кода, которая отсутствует при альтернативном подходе, предотвращается тестами.

Сгенерированные сущности, представляющие собой модуль с функциями для работы с конкретной таблицей, возвращаются препроцессору, который модифицирует программу во время компиляции.

3.2. Генерируемый код

Для каждого пользовательского модуля генерируется модуль со следующими структурами и функциями, которые работают с соответствующей таблицей:

1. вспомогательные функции и типы для работы с Caqti;
2. модуль Fields, который содержит значения-поля модуля, участвующие в составлении запросов, будь то условие или группировка;
3. in_condition — специальная функция для составления подзапросов, которые могут участвовать в условиях для функций других модулей;
4. отдельная структура, содержащая информацию об уже использованных идентификаторах. Необходима для определения уже занесённых модулей и выдаче ещё не занятых идентификаторов;
5. get_uuid — функция получения уникального не использованного в таблице идентификатора;
6. модуль Q для внутреннего пользования, реализующий базовые функции-запросы, например select, insert и так далее. Они по переданным параметрам способны составлять безопасный в смысле Caqti запрос любой выразимой библиотечными средствами сложности;
7. connect_inited и connect_not_inited — функции создания обращения к конкретной таблице в базе данных, связанной с модулем. Выбор между ними обуславливается фактическим состоянием таблицы: была та объявлена или нет. При этом они реализуют разную логику относительно структуры идентификаторов: первая обновит её значениями из таблицы, а вторая инициализирует пустым значением. Непосредственно доступны пользователю и необходимы для безопасной работы;

8. init — высокоуровневая функция физической инициализации таблицы в базе. Инициализирует вспомогательные таблицы, если это требуется;

9. drop — функция физического удаления основной и вспомогательных таблиц в базе;

10. add — физическое добавление переданного модуля в таблицу. Это добавление глубокое в том смысле, что если полученный модуль содержит значения, которые также могут храниться в базе (различные связи между таблицами), то они рекурсивно будут занесены в базу. Если модуль уже содержится в таблице, то соответствующие записи будут обновлены;

11. update — глубокое обновление связанных с модулем записей. Если модуль ещё не был добавлен, то он просто заносится в таблицу;

12. delete — удаление записей по условию, которое составлено из библиотечных значений, не затрагивает подмодули;

13. update_by_function — обновление конкретного поля модуля в таблице по заданной библиотечными значениями формуле, которая может содержать как константы, так и другие поля;

14. select — выборка модулей по полученному условию, которая позволяет сортировать получаемые значения по одному из полей, выбирать и пропускать произвольное их количество. Если модуль связан связями, то возможно исполнение дополнительных запросов;

15. select_with_aggr — получение результата агрегирующей функции по модулям, соответствующим условию, возможно с применением группировки и ключевого слова HAVING из SQL. Возвращается список пар, где первое значение это непосредственно результат агрегации, а второе — значение, по которому произошла группировка.

3.3. Поддерживаемые типы и отображение

На данный момент библиотека поддерживает следующие типы и связи:

1. int — отображается в INT, так как он 64-ёх битный и вмещает его в себя;

2. string — отображается в VARCHAR(256), для больших по длине строк предлагается объявлять собственный тип;

3. float — отображается в REAL, так как он тоже схож по количеству битов;

4. bool — отображается в INT, потому что SQLite не поддерживает булевы типы;

5. uuid — непосредственно в OCaml представлен как int64, а в схеме как INTEGER;

6. option — всякий тип, за исключением других модулей и пользовательских, может быть опционален, это означает, что в схеме для него не будет применено ограничение NOT NULL;

7. определённые пользователем типы также могут быть сохранены в таблице как VARCHAR(1024), однако требуется, чтобы были объявлены функции с именами encoder_<name> и decoder_<name>, где <name> — имя типа в нижнем регистре. Первая должна конвертировать значения в строки, а вторая соответственно из строк получать их. Поля подобных типов не могут участвовать в составлении запросов и могут быть обновлены только с помощью update, так как отображаются в SQL не напрямую;

8. one-to-one связь — реализуется через дополнительное поле типа uuid, в котором содержится значение идентификатора связанного модуля;

9. one-to-many и many-to-many связи — реализуются с помощью дополнительных таблиц, что является стандартной практикой для реляционных баз данных. Код, работающий с задействованными модулями, учитывает эту специфику. Для поддержания целостности данных используются каскадные ограничения на удаление в дополнительных таблицах.

3.4. Используемые значения

Для составление корректных в синтаксическом плане, а также по типам запросов используются четыре типа значений: поле, условие, аргумент, агрегатор. Каждый из них параметризуется фантомными типами, которые представляют необходимую информацию. Это позволяет ещё до компиляции с помощью системы типов выявлять некорректные условия или запросы.

Так, типы не позволят сравнить строку и число, просуммировать булевы значения или обновить неопциональное поле опциональными значениями.

Значение типа поле следует получать только из сгенерированных модулей Fields. Это позволяет их типизировать непосредственно тем модулем, в котором они находятся, вследствие чего в условии для одного модуля не могут участвовать поля, в него не входящие, и тому подобное.

Условия составляются с использованием библиотечных конструкторов. В них могут быть задействованы базовые операторы сравнения, вхождения строк и проверки на NULL, они могут быть объедены булевыми операторами или кванторами. Значения типа условие могут содержать подзапросы, для проверки вхождения поля или значения.

Листинг 7: Базовый запрос с простым условием

```
select (field x ≥ int_const 42 && field y == int_const 0)
```

Аргументы выступают как участники условий или как значения для обновления полей функцией update_by_function. Они могут быть пред-

ставлены константой, полем, либо какой-то базовой функцией над другими аргументами, например, сложением, объединением строк и так далее.

Библиотека реализует все доступные в SQLite3 агрегаторы.

Значения всех описанных типов напрямую отображаются в SQL, перекладывая вопросы оптимизации на базу данных.

3.5. Безопасная комбинация запросов

Любая высокоуровневая функция запроса, которую должен использовать пользователь при работе с базой, принимает и возвращает значения, обёрнутые в монаду, содержащую информацию о подключении, а также параметризующуюся типами, описывающими состояние конкретной таблицы и саму таблицу.

Листинг 8: Базовыая комбинация запросов

```
connect_not_inited ▷ init ▷ add p1 ▷ add p2 ▷ select ctrue
```

Это позволяет безопасно комбинировать запросы при условии, что пользователь не пытается намеренно обойти ограничения.

Так, если пользователь начал обращение к базе с connect_not_inited, что предполагает физическое несуществование таблицы, типовая система не позволит ему ничего сделать кроме как применить init, и только потом уже начать работать с таблицей с помощью других функций.

Или же, если у какого-то модуля есть подмодули, то для начала работы с его таблицей в функции connect необходимо передать произвольные значения подтаблиц, которые обязательно параметризованы типом, сообщающим, что в них можно писать.

OCaml не позволяет описать типы так, чтобы гарантировать безопасность в подобном случае, для этого бы потребовались линейные типы[6], которые в нём не реализованы. Они бы позволили запретить на уровне типов создавать несколько значений обращения к таблице. Вследствие чего можно было бы гарантировать состояние таблицы, опи-

[6]https://runebook.dev/ru/docs/haskell/users_guide/exts/linear_types Дата обращения: 03.06.2024

санное в типе, так как это значение могло бы существовать в единственном экземпляре, а измениться оно может только вследствие применения функций запросов, по контролируемым правилам.

Но даже в таком случае всё же пришлось бы надеяться на сознательность пользователя при выборе connect_inited и connect_not_inited.

3.6. Миграции

Предполагается, что пользователь взаимодействует с базой данных только через код, при этом схема базы данных сильно привязана к нему, начиная с имён полей. Любое изменение, затрагивающее модуль, приведёт к расхождениям. Для решения данной проблемы реализованы в отдельном модуле следующие функции:

1. add — добавление поля в таблицу с некоторым базовым значением;
2. remove — удаление поля из таблицы;
3. rename — задание нового имени для поля.

Сценарий работы с ними следующий. Пусть у пользователя уже есть какой-то модуль, соответствующая ему таблица объявлена и содержит записи. Ему потребовалось добавить новое поле в модуль, тогда он просто вносит его туда, заново компилирует эту часть кода, чтобы получить возможность использовать значение типа поле. Применяет функцию add с новым полем и каким-то значением для начальной инициализации, чем изменяет таблицу в соответствии с новой структурой модуля.

Работа с remove и rename аналогична.

Изменения внутри базы библиотекой осуществляются с помощью ключевого слова ALTER, вся дополнительная информация содержится в значениях типа поле.

При возникновении расхождений, которые не были исправлены, в момент исполнения запросов пользователь получит ошибку.

3.7. Тестирование

Для библиотеки были написаны два вида тестов:

1. in-line — покрывающие внутренние функции и сложно воспроизводимые ветви кода;

2. cram — представляющие собой сценарий использования утилиты.

Покрытие кода измерялось с использованием утилиты bisect_ppx. И проект был покрыт на 74 процента, которые затрагивали основные сценарии использования.

В репозитории настроен CI, запускающий тесты на каждый коммит, проверяющий форматирование исходного кода и полученные результаты с помощью Coveralls[7] превращающий в бейджики .

[7]https://coveralls.io Дата обращения: 24.12.2023

Заключение

В результате данной работы была реализована ORM-библиотека для OCaml — mrm.

Были выполнены следующие задачи:

1. реализованы безопасные по типам условия для запросов;
2. реализованы агрегаторы и группирующие функции;
3. реализованы связи один-ко-многим и многие-ко-многим;
4. реализованы инструменты для миграции.

Код проекта вместе с примерами доступен в GitHub-репозитории, имя пользователя Tozarin. https://github.com/Tozarin/mrm/tree/dev03

Список литературы

[1] A. Urkedal Petter. Caqti's documentation. — URL: https://github.com/paurkedal/ocaml-caqti (дата обращения: 2023-12-22).

[2] Chamberlin Donald, Boyce Raymond. SEQUEL: A structured English query language. — URL: https://doi.org/10.1145/800296.811515 (дата обращения: 2023-12-21).

[3] Codd Edgar Frank. LINQ2DB Documentation. — URL: https://linq2db.github.io (дата обращения: 2023-12-21).

[4] Codd Edgar Frank. A Relational Model of Data for Large Shared Data Banks. — URL: https://doi.org/10.1145/362384.362685 (дата обращения: 2023-12-21).

[5] Group Jane Street. ppxlib's documentation. — URL: https://github.com/ocaml-ppx/ppxlib (дата обращения: 2023-12-22).

[6] Jones Richard, other authors. PG'OCaml's documentation. — URL: https://github.com/darioteixeira/pgocaml (дата обращения: 2023-12-22).

[7] Jones Richard, other authors. PGX's documentation. — URL: https://github.com/arenadotio/pgx (дата обращения: 2023-12-22).

[8] Madhavapeddy Anil, Gazagnaire Thomas. orm's documentation. — URL: https://github.com/mirage/orm (дата обращения: 2023-12-22).

[9] Microsoft. Entity Framework's documentation. — URL: https://learn.microsoft.com/ru-ru/ef (дата обращения: 2023-12-22).

[10] Object-Relational Mapping Revisited - A Quantitative Study on the Impact of Database Technology on O/R Mapping Strategies / Martin Lorenz, Jan-Peer Rudolph, Günter Hesse et al. — URL: https://doi.org/10.24251/hicss.2017.592 (дата обращения: 2023-12-21).

[11] Privér Emil. DBCaml's documentation. — URL: https://priver.dev/blog/dbcaml/dbcaml-project/ (дата обращения: 2024-06-03).

[12] SQLite3-OCaml's documentation. — URL: https://github.com/mmottl/sqlite3-ocaml/?tab=readme-ov-file (дата обращения: 2024-06-03).

[13] Slick's documentation. — URL: https://github.com/slick/slick (дата обращения: 2024-05-03).

[14] Squeryl's documentation. — URL: https://github.com/squeryl/squeryl (дата обращения: 2024-05-03).

[15] for Idiomatic Java Relational Persistence. Hibernate's documentation. — URL: https://hibernate.org (дата обращения: 2023-12-22).

[16] ppx_rapper's documentation. — URL: https://github.com/roddyyaga/ppx_rapper (дата обращения: 2024-06-03).

[17] the SQLAlchemy authors, contributors. SQLAlchemy's documentation. — URL: https://www.sqlalchemy.org (дата обращения: 2023-12-22).

[18] the camlp4 authors, contributors. camlp4's documentation. — URL: https://github.com/camlp4/camlp4 (дата обращения: 2023-12-23).