Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 21.Б15-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
2023

Оглавление

Введение **3**

1. Постановка задачи **5**

2. Обзор **6**

2.1. ORM-фреймворки 6

2.2. Существующие решения для OCaml 8

3. Реализация **10**

3.1. First-class модули 10

3.2. mrm библиотека 10

3.3. Генерация кода 11

3.4. Сценарий работы с библиотекой 13

3.5. Тестирование 14

Заключение **15**

Список литературы **16**

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) [11]. 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 [15], Hibernate [13], 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-

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

запросы внутри и из кода. Не все из них учитывают специфику языка и ни одна в полной мере не реализует ORM-подход в полной мере.

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: 23 декабря 2023 г.

1. Постановка задачи

Целью этой учебной практики является реализация ORM-библиотеки для OCaml.

Для ее выполнения поставлены следующие задачи:

1. провести обзор ORM-фрейморков и ORM-библиотек для других языков;
2. провести обзор существующих решений для работы с базами данных для OCaml;
3. реализовать ORM-библиотеку.

2. Обзор

2.1. ORM-фреймворки

Обзор ORM-фреймворков проводился с целью выяснить, какие модели и сценарии работы может предполагать этот подход и какая из них лучше всего подойдёт для OCaml библиотеки.

2.1.1. Entity Framework

Entity Framework (EF) [10] — это фреймворк для .NET, который предоставляет возможность создавать запросы к базе данных на языке C#, позволяет выполнять различные операции с данными, такие как: выборка, обновление, вставка и удаление с использованием синтаксиса LINQ. А также содержит инструменты для отображения классов и объектов исходного кода в записи базы данных.

Сценарий работы с ним предполагает следующее: разработчик либо по схеме базы данных, либо по самоописанной модели, либо по исходному коду возможностями EF генерирует весь необходимый вспомогательный код для работы с данными и их отображения. Затем создаёт подключение к интересующей его базе данных инстанцированием специального класса, работает с данными как с объектами, производит их обновление в базе данных и в конце закрывает подключение.

Entity Framework даёт возможность создавать типобезопасный код работы с базами данных без исопользования «магических» строчек и предоставляет инструменты для генерации кода, отвечающего за отображения данных, что освобождает от монотонного процесса его написания.

2.1.2. SQLAlchemy

SQLAlchemy [15] — библиотека для Python, которая позволят работать с реляционными базами данными на Python. Она предоставляет стандартные для ORM-фреймворков возможности и сценарии работы, а также разрешает работать вне концепции ORM.

Однако библиотека предлагает разработчикам самостоятельно описывать классы и их отображения на базы данных с использованием своих внутренних классов и свойств. С одной стороны, это обязывает писать много однотипного кода, с другой стороны, даёт возможность гибче описывать связи между классами, а также упрощает устройство самой библиотеки.

2.1.3. Hibernate

Hibernate [13] — ORM-решение для Java с похожими на все ранее рассмотренные по функциональности и схеме работы, которые они предоставляют при работе с данными.

Отображение в Hibernate задаётся самим разработчиком при помощи аннотаций или специальных конфигурационных файлов. Такой подход не обязывает описывать классы специальным образом как в SQLAlchemy, но всё ещё требует дополнительной деятельности при работе с моделями.

2.1.4. Выводы

Все рассмотренные фреймворки предлагают одинаковые сценарии работы, а именно: создание подключения, формирование запроса в большинстве своём без SQL-запроса используя собственные предоставляемые функции, работа с данными средствами фреймворка или языка и обновление данных в таблице через соответствующий запрос к базе с последующим закрытием подключения.

Однако решения отличаются подходом отображения внутрикодовых моделей и моделей баз данных друг на друга, они либо более гибкие, что просто позволяют описать схему базы данных, либо менее времязатратные для разработчика по времени вплоть до того, что просто автоматические.

В любом случае без генерации кода не обойтись, а так как OCaml позволяет использовать предпроцессор и тем самым изменять исходный код в определённых местах ещё до компиляции, было принято реше-

ния выбрать подход подобный Hibernate: генерировать код отображения для тех объектов, которые разработчик в исходном коде отметит специальной директивой предпроцессора. То бишь на конфигурацию отображения будет тратиться минимальное время, но оно будет предоставлять меньше возможностей.

2.2. Существующие решения для OCaml

Для OCaml уже существуют решения для работы с базами данных, поэтому не было никакого смысла в том, чтобы тратить время на собственную реализацию библиотеки именно для вызова SQL-запросов из кода, а стоило уже выбрать готовую и реализовать над ней ORM-надстройку.

Однако стоит отметить, что для OCaml уже предпринималась попытка создания ORM-библиотеки — orm [9]. Она была типобезопасной в том смысле, что созданное для чтения подключение не позволяло на уровне типов изменять базу данных. При этом orm генерировала дополнительный необходимый код для отображения и работы, основываясь на схемах баз данных и работала с SQLite.

Однако её развитие и поддержка остановились семь лет назад, в личной переписке с автором выяснилось, что это произошло из-за устаревания camlp4 [16] — одной из технологий, от которой сильно зависила библиотека. Поэтому возобновление её поддержки невозможно.

2.2.1. PostgreSQL-OCaml

PostgreSQL-OCaml [6] — библиотека, предоставляющая интерфейс для работы с PostgreSQL. Из себя представляет обёртку над C-библиотекой libpq [14] для OCaml. Позволяет создавать многопоточные приложения, но работает только с PostgreSQL.

2.2.2. PG'OCaml

PG'OCaml [7] — билиотека-клиент для работы с PostgreSQL, которая написана на чистом OCaml без использования C-кода. Для неё

существует расширение PGX [8], позволяющая проверять на типовую корректность во время компиляции написанные разработчиком SQL-запросы, а также отображать SQL-типы в типы OCaml.

2.2.3. Caqti

Caqti [1] — монадическая типобезопасная библиотека, которая позволяет работать с PostgreSQL, SQLite, MariaDB. Её интерфейс даёт возможность описывать типы пользовательских запросов, чтобы затем проверять их на корректность в коде, и работать в одном или нескольких потоках.

2.2.4. ezpostgresql

ezpostgresql [12] — библиотека для работы с PostgreSQL, в которой намеренно отказались от какой-либо типобезопасности и сосредоточились на удобстве работы в несколько потоков.

2.2.5. Выводы

В большинстве своём описанные выше библиотеки поддерживают работу только с PostgreSQL и только Caqti даёт возможность взаимодействовать с другими базами данных, например, с SQLite, что может облегчить тестирования как самой библиотеки, так и использующего её кода, при тэтом она поддерживает многопоточность и никак не ограничивает при создании запросов, поэтому было принято решение основать ORM-библиотеку на её базе.

3. Реализация

Работа была разделена на несколько этапов.

3.1. First-class модули

Объектами, в которые будут производиться отображения, были выбраны first-class модули. Это значения, которые могут быть получены из и конвертированы обратно в модули языка.

```
module type Point = sig
  include Uuid

  val x : int
  val y : int

  val show_x : int -> string
  val show_y : int -> string
end
```

Они могут содержать как сами данные, так и функции их использующие, в связи с чем удобны для работы с данными.

3.2. mrm библиотека

Основа реализованной библиотеки представляет собой несколько модулей, которые:

1. содержат типы, оборачивающие подключение к базе данных и типы, не влияющие на работу, но параметризующие его текущее состояние: ‘RW — доступно для чтения и записи, ‘$DROP$ — для конкретного подключения была вызвана функция *drop* и так далее. Это позволяет при правильном описании сигнатур функций работы с базами данных ещё до компиляции выявлять ошибки с помощью системы типов, например:

```
let _ = ... Point_DB.drop >>= Point.add ...
```

Функция *drop* должна вернуть подключение, которое параметризовано типом $`DROP$, а *add* ожидает подключение с типом $`RW$, соответственно в этом участке кода не сойдутся типы и компиляция не состоится;

2. функции непосредственно для создания базового подключения к различным базам данных, таким как: PostgreSQL, SQLite, MariaDB, а также подключения для тестов, реализованного как доступ к $SQLite :: memory$:;

3. отдельный модуль $UUID$, отвечающий за идентификацию записей в базе данных.

3.3. Генерация кода

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 следующего кода, директива *show* сгенерирует функцию, которая будет по переданному значению типа *point* строить его строковое представление.

```
type point =
    | TwoDimension of { x : int; y : int }
    | ThreeDimension of { x : int; y : int; z : int }
[@@deriving show { with_path = false }]
```

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

Это происходит следующим образом: собирается информация об именах и их типах, содержащихся в модуле, проверяется включение

модуля, отвечающего за идентификацию, далее по этим данным собирается генерируемый код в строчку, которая с помощью внутренней функции OCaml-парсера, преобразуется в часть дерева абстрактного синтаксиса и она вставляется в исходный код программы.

Помимо внутренних и необходимых для работы с Caqti генерируются следующие функции и типы:

1. conn — тип, описывающий состояние конкретной таблицы в базе данных, а также всех связанных с ней связью один-к-одному. Его необходимо передавать во все функции работы с данными, которые он параметризует типом исходного first-class модуля;

2. connect — функция, предоставляющяя начальное состояние конкретного подключения к таблице;

3. connect_inited — аналогичная connect функция, которая предполагает, что таблица для обращения уже была инициализирована, это необходимо для поддержания типобезопасности;

4. get_new_uuid — создание уникального ключа для записи и соответствующей ей модуля;

5. new_*modulename* — базовая функция инстанцирования для модуля;

6. init — инициализация таблицы в базе данных;

7. drop — удаление таблицы. Полученное после функции подключение своим типом не позволит вызвать никакие другие функции кроме init, что должно предотвратить лишние ошибки;

8. add — добавление записи в таблицу;

9. add_deep — аналогичная функция, однако, которая добавляет записи в зависимые таблицы, если исходный модуль содержит значения типов других модулей;

10. delete — удаление записи из таблицы;

11. select_by_uuid — получение объекта по уникальному ключу;
12. select_all — выборка всех объектов таблицы для последуюшей работы;
13. commit — обновление записей в таблице по переданному списку модулей;
14. migrate — перенос всех записей из таблицы одного базового подключения в таблицу другого.

3.4. Сценарий работы с библиотекой

При работе с библиотекой mrm предполагается следующий сценарий работы: разработчик описывает модули, которые по его мнению должны быть представлены в базе, создаёт подключение к интересующей его базе данных, инициирует таблицы, добавляет в них записи или делает выборки, работает с полученными данными средствами OCaml, обновляет записи в таблице.

```
module type Point = sig
  include Uuid

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

let _ = connect_sqlite ~database:'':memory:''
    >>= Point_DB.connect_not_inited
    >>= Point_DB.init
    >>= ...
    >>= Point_DB.commit conn
```

При этом, как отмечалось выше, типовые ограничения запретят ему компилировать заведомо ошибочный код, например такой, как:

```
let _ = ... >>= Point.connect_not_inited >>= Point.add ...
```

Так как *connect_not_inited* возвращает подключение с типом $`NOTINITED$, а *add* ожидает тип $`RW$. При этом следующий код скомпилируется, потому что функция *init* примет подключение с $`NOTINITED$ и вернёт с $`RW$:

```
let _ = ...
    >>= Point.connect_not_inited
    >>= Point.init
    >>= Point.add
    ...
```

3.5. 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 66 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[2] превращающий в бейджики. .

[2]https://coveralls.io/

Заключение

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

Были выполнены следующие задачи:

1. проведён обзор ORM-фреймворков и ORM-библиотек для иных языков программирования;
2. проведён обзор существующих решений для работы с базами данных на OCaml;
3. реализована ORM-библиотека, позволяющая прозрачно отображать first-class модули на записи в базе данных, используя типобезопасный код, выполнять простые запросы и работать с несколькими типами баз данных.

Однако в данный момент библиотека:

1. блокирует базу данных для других потоков при обращении;
2. реализует ограниченный набор типов для отображения в базу данных, в который не входят, например, списки, записи, пары;
3. реализирует только связь один-к-одному между записями, что ограничивает в построении модели базы данных;
4. предлагает фильтровать данные внутри кода, а не средствами базы, замедляя работу, так как не умеет генерировать подходящие SQL-запросы.

Код проекта доступен в GitHub-репозитории, имя пользователя Tozarin. https://github.com/Tozarin/mrm

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] Grove Sean, Mottl Markus. PostgreSQL-OCaml's documentation. — URL: https://mmottl.github.io/postgresql-ocaml/ (дата обращения: 2023-12-22).

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

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

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

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

[11] 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).

[12] Priambodo Bobby. ezpostgresql's documentation. — URL: https://github.com/bobbypriambodo/ezpostgresql (дата обращения: 2023-12-22).

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

[14] the PostgreSQL Global Development Group. PostgreSQL libpq's documentation. — URL: https://www.postgresql.org/docs/current/libpq.html (дата обращения: 2023-12-22).

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

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