Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 22.Б11-мм

Доработка линтера для OCaml

Ржанков Артём Евгеньевич

Отчёт по учебной практике
в форме «Производственное задание»

Научный руководитель:
ассистент кафедры системного программирования Д. С. Косарев

Санкт-Петербург
2023

Оглавление

Введение

Все программисты стремятся к написанию качественного кода, но часто даже опытные разработчики допускают ошибки. Некоторые из них являются стилистическими, другие же могут стоить большого количества времени отладки разработчиком и потерянных средств компании в случае отказа системы. К счастью, за время развития индустрии было создано множество инструментов, предназначенных для обнаружения ряда подобных ошибок, одним из которых является линтер. Линтер — это статический анализатор кода. Он позволяет привести код к единому стилю, принятому в сообществе, что улучшает его читаемость и упрощает совместную разработку и дальнейшее сопровождение созданного программного обеспечения. Многие линтеры также способны выявлять более серьезные ошибки, приводящие к проблемам с безопасностью и снижению производительности. Также линтеры упрощают проведение код-ревью, поскольку вместо траты время на обсуждение стилистических ошибок, можно сконцентрировать все внимание на более концептуальных проблемах.

Из-за своих преимуществ линтеры пользуются широкой популярностью и доступны практически для всех языков программирования. Zanuda — это линтер для языка программирования OCaml, созданный преподавателем кафедры системного программирования Д.С. Косаревым с целью проверки домашних заданий обучающихся. На данный момент проект активно развивается и уже имеет широкий спектр возможностей, но все еще требует доработки в нескольких областях.

В рамках данной работы предлагается улучшить линтер Zanuda посредством добавления нового линта и создания инструмента для автоисправления некоторых уже существующих линтов.

1. Постановка задачи

Целью работы является доработка линтера Zanuda. Для её выполнения были поставлены следующие задачи:

1. Реализовать линт для обнаружения вложенных if операторов.
2. Реализовать инструмент для автоисправления линтов, что включает в себя следующие промежуточные задачи:
   - Создание патча на основе данных, полученных линтером в ходе анализа исходного кода.
   - Реализация парсера комментариев.
   - Применение исправлений к кодовой базе проекта.
3. Провести тестирование новой функциональности.

2. Обзор

2.1. Используемые технологии

2.1.1. OCaml [3]

Функциональный язык программирования со статической типизацией, поддерживающий также императивную и объектно-ориентированную парадигмы.

2.1.2. Dune [5]

Система сборки для проектов на OCaml.

2.1.3. Angstrom [1]

Библиотека парсер-комбинаторов.

2.2. Принцип работы Zanuda

Взаимодействие с приложением происходит при помощи командной строки. На вход линтеру подается путь к корневой директории проекта. Исследуемый проект должен быть предварительно собран при помощи Dune, поскольку с её помощью линтер получает доступ к различным артефактам компиляции, таким как типизированное дерево, на основе анализа которого работает большинство линтов. Все линты разделяются на несколько уровней: Warn, Deny, None, Allow. Пользователь по своему желанию может отключить для обнаружения все линты уровня Allow передачей соответствующего флага при запуске программы. Отчет предоставляется в `stdout` с описанием каждой найденной ошибки, её локацией в исходном проекте и предложением по исправлению. Также можно настроить линтер на предоставление результатов в формате json для их более детального изучения. Есть интеграция с GitHub Actions, позволяющая автоматически создавать code review.

2.3. Существующие решения

2.3.1. clippy [4]

Линтер написан для языка программирования Rust и на данный момент имеет в своей базе более 650 линтов. Все линты разделяются на категории, каждая категория по умолчанию имеет один из пяти возможных уровней, отвечающих за действия линтера после нахождения линта для данного уровня. Таким образом, можно настраивать поведение линтера, изменяя уровень для выбранной категории. Помимо этого, поддерживается автоматическое применение некоторых предложений линтера к исходному коду проекта. Внедрение именно этой функциональности в Zanuda является одной из задач данной практики.

Также стоит отметить, что при создании Zanuda был вдохновлен именно этим линтером из-за его большого количества преимуществ перед уже существовавшими на тот момент линтерами для языка OCaml.

2.3.2. camelot[1]

Настраиваемый линтер OCaml с поддержкой различных форматов отчета — поддерживается как отчет с числом найденных линтов в проекте, так и наиболее подробный отчет для каждого линта, состоящий из описания линта и предложения по исправлению. Линтер совершает анализ на дереве абстрактного синтаксиса, которое не предоставляет информацию о типах, в связи с чем большинство линтов camelot нацелены на обнаружение именно стилистических ошибок. Поддерживается расширение для VS Code, которое имеет следующий эффект: при обнаружении изменения в проекте языковой сервер запускает camelot и отображает все предупреждения для найденного файла. В последнее время инструмент не поддерживается.

Zanuda также содержит в своей базе группу линтов, вдохновленных данным линтером.

[1]Код проекта на GitHub: https://github.com/upenn-cis1xx/camelot

2.3.3. ocp-lint [6]

Линтер для OCaml, имеющий несколько параметров конфигурации. Из особенностей можно выделить то, что анализ совершается как на абстрактном синтаксическом дереве, так и на типизированном дереве, что расширило возможности для добавления линтов. Очевидным минусом является фактическое отсутствие документации к линтам и в целом маленькая база проверок. Перестал поддерживаться в 2018 году.

3. Реализация

3.1. Линт о вложенных if операторах

Программисты иногда злоупотребляют вложенностью if выражений, что ухудшает читаемость кода, хотя в большинстве случаев с помощью небольшого рефакторинга можно сделать код более понятным.

Данный линт находит операторы if с уровнем вложенности свыше двух.

Листинг 1: Пример конструкции, которую находит линт

```
if cond1 then ... else if cond2 then if cond3 then ...
```

Листинг 2: Предложение по исправлению

```
Use let statements or helper methods, or rethink logic
```

3.2. Инструмент для автогенерации исправлений в коде

Основной задачей данной практики являлось создание инструмента для применения автоисправлений к исходному коду. Это позволит сэкономить время программиста, избавив его от рутинной работы по исправлению замечаний линтера. Также это может быть особенно полезным при интеграции Zanuda в уже существующие проекты, поскольку ручное исправление большого количества кода может стоить высоких трудозатрат.

3.2.1. Создание патча

Для подмножества линтов, описанного ниже, были добавлены соответствующие модули, создающие патч по исходному выражению, полученному в ходе анализа типизированного дерева. Взаимодействие с каждым модулем происходит посредством вызова функции `apply_fix`, принимающей на вход выражение AST. Патч для проекта представлен

в виде набора, содержащего пары ключ/значение, где ключом является имя файла, а значением — набор текстовых замен для данного файла. Каждая замена характеризуется двумя параметрами: местоположением в исходном коде и полезной нагрузкой. Такое представление является довольно универсальным, поскольку позволяет легко как вставлять новый код, так и удалять уже существующий.

На данный момент реализована генерация замены для следующих линтов:

- `if_bool` (Документация)

Листинг 3: Пример конструкции, которую находит линтер

```
if x then false else true
```

Листинг 4: После применения исправлений

```
not x
```

- `propose_function` (Документация)

Листинг 5: Пример конструкции, которую находит линтер

```
let f arg  = match arg with
| [] -> ...
| (x::xs) -> ...
```

Листинг 6: После применения исправлений

```
let f = function
| [] -> ...
| (x::xs) as arg -> ...
```

- `record1` (Документация)

Листинг 7: Пример конструкции, которую находит линтер

```
{ x = r.x; y = r.y; z = 15}
```

Листинг 8: После применения исправлений

```
{ r with z = 15 }
```

3.2.2. Парсер комментариев

Поскольку в типизированном дереве отсутствует информация о комментариях в файле, было необходимо реализовать для них собственный парсер. Необходимость в нем обуславливается тем, что при удалении кода нельзя было получить комментарии из указанного диапазона и учесть их при рефакторинге. Перед применением замен к отдельному файлу *(пункт 3.2.3)* запускается синтаксический анализатор. В качестве входных данных он получает текстовое представление исходного кода, а выходными данными является список кортежей из местоположения комментария в файле и его содержимого. Каждый удаляемый участок кода исследуется на наличие комментариев, которые, в случае обнаружения, добавляются в общий пул замен.

Ниже можно увидеть соответствующий эффект:

Листинг 9: Пример конструкции

```
{ x = p.x + 1; (* 1 *) y = p.y; (* 2 *) z = p.z (* 3 *) }
```

Листинг 10: После применения исправлений

```
{ p with x = (p.x + 1) }  (* 1 *) (* 2 *) (* 3 *)
```

3.2.3. Применение изменений к проекту

Стоит отметить важность предоставления пользователю возможности просмотра сгенерированных изменений перед их применением к файлам проекта. Это мотивировано в первую очередь тем, что модуль автоисправления встроен в базовую реализацию линтера и в противном случае пользователь увидит список ошибок уже после применения исправлений к проекту. Поэтому для каждого исходного файла создается файл с примененным патчем, а также общий журнал с изменениями

для всего проекта. Позже пользователь может применить данные исправления непосредственно к кодовой базе проекта. Ниже приведено подробное описание данного процесса.

Для каждого файла на предыдущем этапе создается патч, который применяется к текстовому представлению файла следующим образом: создается буфер, в который совместно записываются патч и код исходного файла, который остался без изменений. Для представления диапазона местоположений используется тип `Location.t`, являющийся частью внутреннего API компилятора OCaml. Для каждой новой замены определяется полезная нагрузка в исходном коде по ее локации и локации предыдущей замены — таким образом переносится контент исходного файла, который не подвергся изменениям. При вставке нового кода используется диапазон локаций нулевой длины, а при удалении участка кода — пустая полезная нагрузка. Далее контент из буфера записывает в файл, название которого представлено как `<source_file>.corrected`. При помощи утилиты `diff` содержимое каждого сгенерированного файла сравнивается с исходным и разница записывается в общий журнал, который затем пользователь может использовать для быстрого ознакомления с предложенными изменениями.

Изначально планировалось применение изменений к кодовой базе проекта посредством интеграции с системой сборки Dune. Она имеет встроенную цель сборки `@lint`, которая может запускать внутри себя препроцессорные расширения, генерирующие код на OCaml, и затем применять их к исходному коду в проекте. Но, как выяснилось в ходе работы, на данный момент Dune не позволяет определить собственное действие по генерации кода и последующему применению к исходным файлам. В связи с этим было решено по ходу работы программы генерировать собственный shell скрипт, который перезаписывает код исходного файла, для которого были созданы исправления, на код в соответствующем `.corrected` файле. После этого пользователь может ознакомиться с исправлениями и применить их к проекту, запустив линтер с соответствующим флагом `--fix` или запустить скрипт самостоятельно, отключив при этом нежелательные изменения.

4. Тестирование

Тестирование в ходе данной работы было проведено с использованием cram [2] тестов, поддерживаемых Dune, которые особенно удобны при тестировании консольных приложений. Они позволяют запускать внутри себя команды и их ожидаемый вывод, который затем сравнивается с выводом, полученным в процессе работы данных команд.

Тестовое покрытие для нового линта и модуля автоисправления составили соответственно 93% и 87%.

Заключение

В ходе данной работы были выполнены следующие задачи:

- Реализован новый линт.
- Создан инструмент для автоисправления линтов.
- Проведено тестирование новой функциональности.

Дальнейшие планы:

- Реализовать поддержку пользовательских действий по генерации кода в Dune.
- Расширить базу линтов для австоисправления.

Линтер — https://github.com/Kakadu/zanuda

Линт о вложенных if-операторах — https://github.com/Kakadu/zanuda/pull/30

Инструмент для автоисправления линтов — https://github.com/Kakadu/zanuda/pull/37

Список литературы

[1] Angstrom. — URL: https://ocaml.org/p/angstrom/0.14.0/doc/Angstrom/index.html (дата обращения: 19 декабря 2023 г.).

[2] Cram tests. — URL: https://dune.readthedocs.io/en/stable/tests.html#cram-tests (дата обращения: 19 декабря 2023 г.).

[3] The OCaml Manual / Xavier Leroy, Damien Doligez, Alain Frisch et al. — 2022. — URL: https://v2.ocaml.org/releases/4.14/htmlman/ (дата обращения: 19 декабря 2023 г.).

[4] clippy. — URL: https://doc.rust-lang.org/nightly/clippy/ (дата обращения: 19 декабря 2023 г.).

[5] dune. — URL: https://dune.readthedocs.io/en/stable/ (дата обращения: 19 декабря 2023 г.).

[6] ocp-lint, A Plugin-based Style-Checker with Semantic Patches / Çagdas Bozman, Théophane Huffschmitt, Michael Laporte, Fabrice Le Fessant // OCaml Users and Developers Workshop 2016. — Nara, Japan, 2016. — URL: https://inria.hal.science/hal-01352013 (дата обращения: 19 декабря 2023 г.).