Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 23.Б10-мм

Улучшение PyFormLang

Бугаев Александр Сергеевич

Отчёт по учебной практике
в форме «Решение»

Научный руководитель:
доцент кафедры системного программирования, к. ф.-м. н., Григорьев С. В.

Санкт-Петербург
2024

Оглавление

Введение

Теория формальных языков играет ключевую роль в информатике и программировании. Эта область математики тесно связана с теорией автоматов [16, 19]; понятие языка также активно используется в таких разделах, как теория вычислимости [20, 21] и теория сложности вычислений [6, 5]. С практической точки зрения, методы разбора формальных языков повсеместно применяются в компиляторах и статических анализаторах кода [10, 3]. Таким образом, развитие данной теории внесло значительный вклад в создание известных нам сегодня языков программирования высокого уровня. Помимо этого, формальные языки нашли своё применение в текстовых редакторах и утилитах [22], графовых базах данных [8], а также в машинном обучении [9] и биоинформатике [26].

Теория формальных языков преподаётся и на Математико-Механическом факультете Санкт-Петербургского государственного университета. В своём курсе[1] С. В. Григорьев использует библиотеку *pyformlang* [25], специализирующуюся на обучении в данной области. Проект реализован на языке программирования *Python*, в нём представлен обширный набор представлений грамматик и автоматов, а также используемых для них алгоритмов. Так как *Python* является динамически типизированным языком, для повышения надёжности кода и удобства его использования применяют типовые аннотации[2]: статические указания типов переменных, параметров и возвращаемых значений функций.

На данный момент у *pyformlang* можно выделить явный недостаток: частичное отсутствие типовых аннотаций[3] в коде, что затрудняет использование библиотеки. Анализ кода дал понять, что за этим стоит

[1]Репозиторий курса по формальным языкам:
https://github.com/FormalLanguageConstrainedPathQuerying/formal-lang-course
[дата обращения: 3 ноября 2024 г.]

[2]Документация по типовым аннотациям *Python*:
https://docs.python.org/3/library/typing.html
[дата обращения: 5 ноября 2024 г.]

[3]Issue на *GitHub* про отсутствие аннотаций:
https://github.com/FormalLanguageConstrainedPathQuerying/formal-lang-course/issues/161
[дата обращения: 3 ноября 2024 г.]

более фундаментальная проблема: запутанная архитектура с множеством циклических зависимостей, препятствующая развитию проекта. В частности, это мешало и добавлению типовых сигнатур. К тому же, в библиотеке можно встретить неоднородную функциональность: так, например, не для всех представлений языков реализованы генераторы слов[4]. В особенности не хватает генератора по конечному автомату[5] — абстрактной машине, способной принять или отвергнуть данную строку.

Таким образом, целью данной работы стало совершенствование проекта *pyformlang*. Для достижения поставленной цели было предложено улучшить интерфейс библиотеки с помощью добавления типовых аннотаций и устранения обнаруженных несоответствий. По мере добавления аннотаций, предлагалось разработать улучшенную архитектуру библиотеки и внедрить её в кодовую базу *pyformlang* для устранения циклических зависимостей. Также в ходе работы планировалось восполнить недостающую функциональность библиотеки добавлением новых алгоритмов. И так мы подходим к задачам, поставленным в рамках данной работы.

[4]Реализация генератора для одного из видов грамматик:
https://github.com/Aunsiels/pyformlang/blob/master/pyformlang/cfg/cfg.py#L925
[дата обращения: 5 ноября 2024 г.]

[5]Issue на *GitHub*, связанная с отсутствием генератора по конечному автомату:
https://github.com/FormalLanguageConstrainedPathQuerying/formal-lang-course/issues/142
[дата обращения: 5 ноября 2024 г.]

1 Постановка задачи

Целью работы является улучшение библиотеки *pyformlang* как с точки зрения пользователя, так и с точки зрения разработчика. Для достижения данной цели были поставлены следующие задачи.

1. Реализовать генератор, выдающий строки, принимаемые конечным автоматом, и внедрить его в основной репозиторий. Он должен быть реализован для наиболее общего представления конечного автомата, не выдавать дубликатов, иметь возможность указать максимальную длину генерируемых строк.

2. Добавить типовые аннотации с помощью одного из статических анализаторов кода на языке *Python*, по возможности переработать код в более строгом виде. Типы должны быть указаны для каждой публичной функции.

3. Разработать улучшенную архитектуру и внедрить её в кодовую базу вместе с типовыми аннотациями. Архитектура должна не иметь циклических зависимостей.

2 Сведения из теории формальных языков

Далее собраны сведения, которые помогут понять принцип работы генератора строк по конечному автомату, роль автоматов в теории формальных языков, а также общую архитектуру *pyformlang*.

2.1 Языки и грамматики

Пусть Σ — некоторое конечное множество символов, которые мы назовём *алфавитом*. Тогда *формальный язык* на Σ можно определить как множество *слов* (*строк*) — конечных последовательностей символов данного алфавита [18]. Далее нам будет удобно обозначить за Σ^* язык, состоящий из всех возможных слов, составленных по алфавиту Σ, включая *пустую* строку ε — строку из нуля символов.

Формальный язык, в свою очередь, можно задать конструкцией, называемой *формальной грамматикой* [17]. Одним из наиболее часто используемых видов грамматик являются *контекстно-свободные* грамматики. Они являются обобщением более простого типа грамматик — *регулярных* грамматик — которые задают языки, называемые *регулярными*.

Кроме грамматик, регулярные языки также часто задаются *конечными автоматами*: в таком случае язык — это множество строк, принимаемых автоматом [16]. Далее нам понадобятся некоторые определения.

2.2 Детерминированные конечные автоматы

Определение 1. Детерминированным *конечным автоматом называется пятёрка* $M = (Q, \Sigma, \delta, q_0, F)$, *где:*

1. Q — *конечное множество, называемое множеством* состояний*;*
2. Σ — *конечное множество, называемое* входным алфавитом*;*

3. δ — *отображение* $Q \times \Sigma \to Q$, *называемое* функцией перехода;

4. $q_0 \in Q$ — начальное *состояние;*

5. $F \subset Q$ — *множество* конечных *состояний.*

Неформально автомат мы можем представить как машину, запоминающую состояние из Q. В начале работы машина находится в состоянии q_0. На вход автомату поступает последовательность символов из Σ — строка — которая читается слева направо. Каждый символ переводит машину из текущего состояния в новое в соответствие с функцией перехода δ.

Далее нам пригодится расширить функцию перехода δ для применения к строкам. Новое отображение $\hat{\delta} : Q \times \Sigma^* \to Q$ мы определим следующим образом:

- $\hat{\delta}(q, \varepsilon) = q$
- $\hat{\delta}(q, xa) = \delta(\hat{\delta}(q, x), a),\ x \in \Sigma^*, a \in \Sigma$

Это поможет нам определить принятие строки конечным автоматом.

Определение 2. *Пусть* M *— детерминированный конечный автомат,* $x \in \Sigma^*$. *Мы говорим, что* M принимает x, *если* $\hat{\delta}(q_0, x) \in F$, *иначе мы говорим, что* M отвергает x.

2.3 Недетерминированные конечные автоматы

Помимо детерминированных автоматов, мы должны рассмотреть их обобщение — *недетерминированные* конечные автоматы [16]. Они отличаются тем, что вместо одного состояния функция перехода δ возвращает подмножества Q. Ещё одно существенное отличие: вместо одного начального состояния q_0 имеется множество начальных состояний $Q_0 \subset Q$. Аналогично случаю детерминированного автомата, определим расширенную функцию перехода $\hat{\delta} : Q \times \Sigma^* \to 2^Q$:

$$\hat{\delta}(q, \varepsilon) = \{q\}, \quad \hat{\delta}(q, xa) = \bigcup_{p \in \hat{\delta}(q,x)} \delta(p, a),\ x \in \Sigma^*, a \in \Sigma$$

И затем будет удобно расширить $\hat{\delta}$ на домен $2^Q \times \Sigma^*$:

$$\tilde{\delta}(\{q_1, \ldots, q_n\}, x) \mapsto \bigcup_{i=1}^{n} \hat{\delta}(q_i, x)$$

Определение 3. *Пусть M — недетерминированный конечный автомат, $x \in \Sigma^*$. Тогда мы говорим, что M* принимает *x, если:*

$$\tilde{\delta}(Q_0, x) \cap F \neq \varnothing$$

Иначе мы говорим, что M отвергает *x.*

Данное определение будет проще воспринимать, если представить конечный автомат как *ориентированный граф*, возможно с *петлями* и *кратными* рёбрами [28]. *Вершинами* в данном случае являются состояния, а *рёбра* задаются функцией перехода δ, причём каждому ребру задаётся значение в виде символа. Тогда работу автомата можно представить как *обход* графа *в ширину* [11], где *фронт* обхода изначально равен Q_0 и на каждом ходе обновляется функцией $\tilde{\delta}$. Можем сказать, что слово принимается автоматом, если после чтения данной строки во фронте лежит хотя бы одно из конечных состояний. Данное представление нам позже пригодится при реализации генератора.

Нам также понадобится ещё одно обобщение: недетерминированный автомат с *ε-переходами* [13]. Здесь единственное отличие в том, что считается возможным переход между состояниями без принятия символа. Функция перехода δ в таком случае имеет домен $Q \times (\Sigma \cup \{\varepsilon\})$, где под ε понимается специальный символ, означающий данный переход.

2.4 Регулярные выражения

Помимо конечных автоматов существует ещё одно представление регулярных языков — регулярные *выражения*. Эти два представления тесно связаны друг с другом: конечные автоматы часто строят по регулярным выражениям, существуют алгоритмы преобразования выражения в автомат и наоборот. Подробнее об этом рассказано в разделе [14], отсюда же нам потребуются некоторые определения.

Определение 4. *Пусть Σ — некоторый алфавит. Считая R* регулярным выражением *и обозначив за $L(R)$ задаваемый им язык, мы можем определить R как выражение над следующими литералами:*

- ε, $L(\varepsilon) = \{\varepsilon\}$
- $a \in \Sigma$, $L(a) = \{a\}$
- $A \subset \Sigma^*$, $L(A) = A$

И затем нам нужно задать для регулярных выражений некоторые операции.

Определение 5. *Пусть R и S — регулярные выражения. Тогда определены следующие операции:*

- объединение $R|S$, $L(R|S) = L(R) \cup L(S)$
- конкатенация RS, $L(RS) = \{\alpha\beta | \alpha \in L(R)\ \beta \in L(S)\}$
- возведение в степень $R^n = \begin{cases} \underbrace{R \dots R}_{n} & n \in \mathbb{N} \\ \varepsilon & n = 0 \end{cases}$
- замыкание Клини $R^* = R^0|R^1|R^2|\dots$

2.5 Существующие методы перечисления строк

Для реализации генератора строк по конечному автомату мы должны рассмотреть некоторые существующие методы перечисления строк по данному представлению языка.

Одной из наиболее цитируемых работ на эту тему является статья [24], рассматривающая перечисление строк в лексикографическом порядке по контекстно-свободным и регулярным грамматикам. Идея перечисления заключается в получении слова, следующего по порядку за данным, с помощью конкатенации общего для данных строк префикса, символа на первой найденной позиции, в которой строки различны, и

минимального возможного суффикса нужной длины. Для этого в статье дан алгоритм получения минимальных строк каждой из допустимых длин по каждому *нетерминалу* данной грамматики. Несмотря на то, что данный метод генерации описан именно для грамматик, для конечного автомата он будет выглядеть аналогичным образом, если *правила* регулярной грамматики представить в виде переходов автомата. Однако, для данного подхода не учтены переходы в несколько состояний по символу, а также ε-переходы, возможные в более общих представлениях автоматов. В связи с этим, данный метод будет актуален лишь для детерминированного конечного автомата. В рамках нашей задачи следует искать более общие подходы и алгоритмы.

Более полезными для нашей работы будут методы генерации строк по недетерминированному автомату без ε-переходов, приведённые в статье [1]. Здесь перечисление языка описано с помощью понятия *кросс-секции* — множества слов фиксированной длины, содержащихся в языке. В таком случае, для перечисления языка в лексикографическом порядке нам нужно получить кросс-секции всех допустимых порядков. Первым делом из данной статьи можно выделить алгоритм вычисления кросс-секции по недетерминированному автомату, обобщающий подход из предыдущей рассмотренной нами работы. Главное различие заключается в методе поиска следующего по порядку слова: минимальный возможный суффикс ищется из всех подходящих состояний автомата. Затем, стоит упомянуть альтернативный метод поиска минимальных слов для каждого состояния, заключающийся в последовательном построении слова с помощью *матриц смежности* по автомату. Наконец, мы должны упомянуть описанный в статье метод вычисления кросс-секции с применением обхода в ширину. Здесь мы строим следующую кросс-секцию по данной, используя представление автомата в виде графа. Получившуюся кросс-секцию мы должны отсортировать, так как порядок на ней не гарантирован. После рассмотрения некоторых подходов к генерации строк будет также полезно сравнить их эффективность.

Описанные в статье методы вычисления минимальных слов и кросс-

секций по-разному комбинируются, производя в итоге несколько алгоритмов перечисления языка. Производительность получившихся алгоритмов в работе [1] была протестирована на случайных автоматах, из результатов данного эксперимента были сделаны некоторые выводы. Для нас будет важен тот факт, что перечисление через обход в ширину оказалось эффективнее на малых автоматах с малой требуемой длиной слов, но проиграло остальным рассмотренным методам в большинстве других случаев.

Приведённые сведения помогут в реализации генератора строк по конечному автомату, описании архитектуры *pyformlang*, а также при переработке некоторой существующей функциональности.

3 Архитектура PyFormLang

До выполнения данной работы в *pyformlang* [25] были реализованы следующие модули:

- **finite_automaton** — конечные автоматы [16];
- **regular_expression** — регулярные выражения [14];
- **rsa** — рекурсивные автоматы [4];
- **cfg** — контекстно-свободные грамматики [15];
- **pda** — автоматы с магазинной памятью [19];
- **fcfg** — грамматики с признаками [27];
- **fst** — конечные автоматы с выходом [23];
- **indexed_grammar** — индексированные грамматики [2].

Существующие зависимости между модулями показаны на рисунке 1.

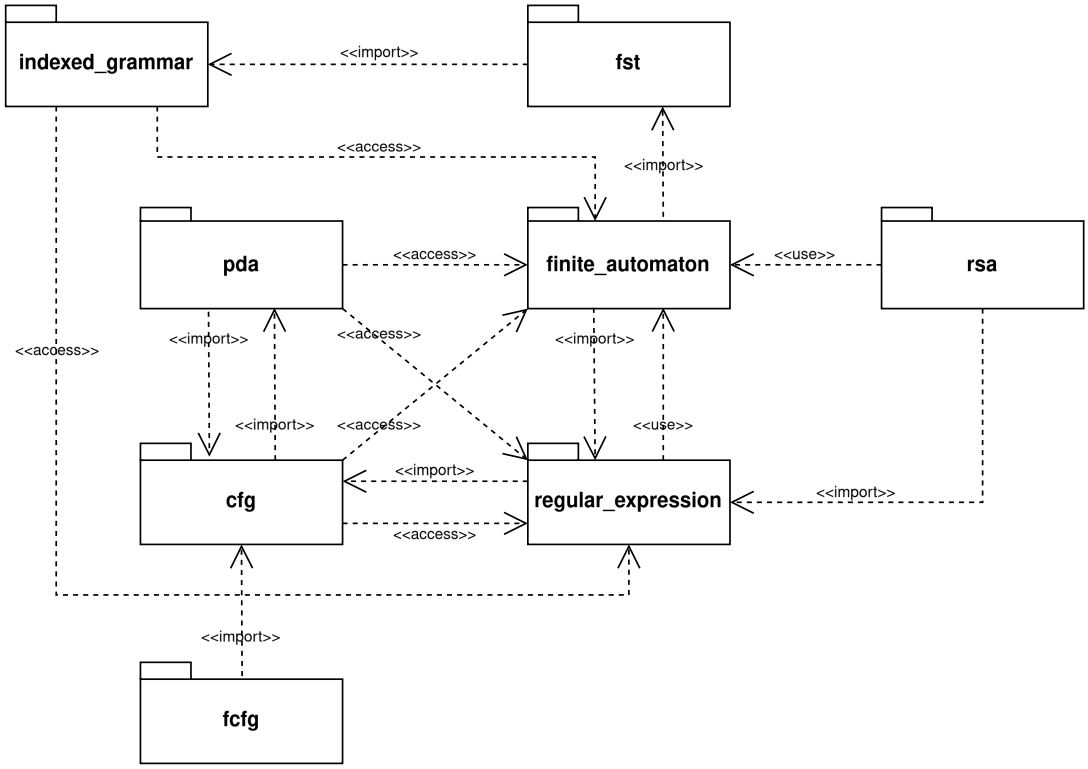

Рис. 1: Зависимости модулей *pyformlang* до выполнения работы

На данном рисунке можно увидеть недостатки архитектуры, заключающиеся в запутанных зависимостях с множеством циклов. И так мы подходим к более подробному описанию данных проблем.

3.1 Проблемы архитектуры

Большинство проблем возникает при наличии двусторонних связей между модулями: между finite_automaton и regular_expression, между cfg и pda. Другие циклы связаны с реализацией алгоритмов пересечения. Так, например, у классов CFG и PDA пересечение должно быть определено строго для детерминированного автомата, но вместо этого конвертация входных данных в автомат происходит на месте. Аналогичная ситуация в классе IndexedGrammar при пересечении с FST.

Отдельного внимания достоин finite_automaton, где проблемы возникают уже на уровне модуля. Циклические зависимости можно увидеть на рисунке 2: здесь каждое представление автомата реализует аб-

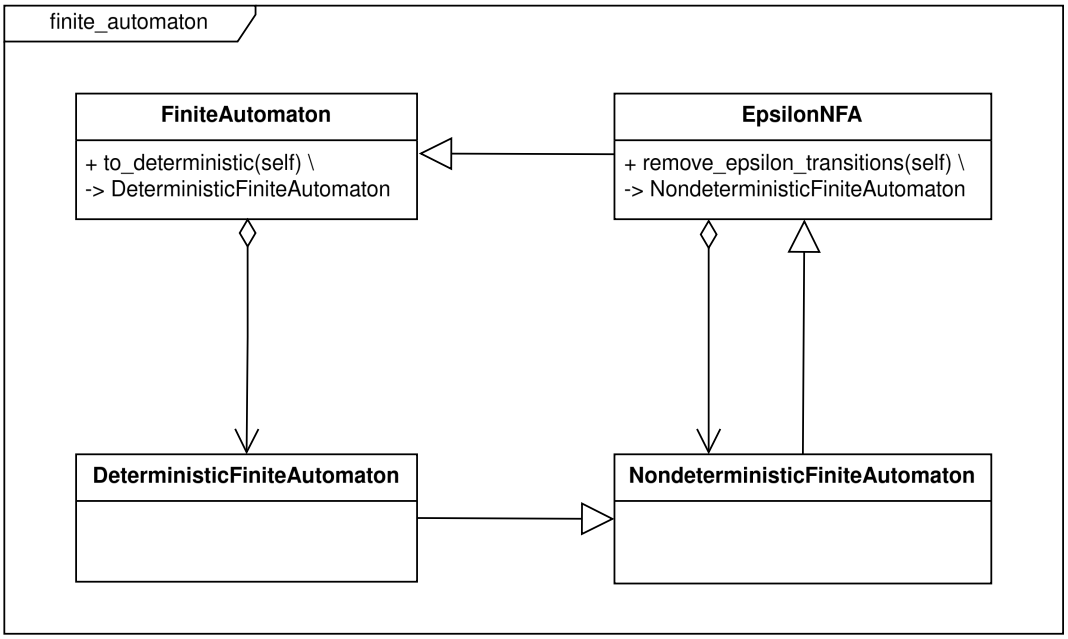

Рис. 2: Зависимости основных классов модуля finite_automaton

страктный метод преобразования в детерминированный конечный автомат. Учитывая то, что каждая реализация автомата наследуется от более общего класса, при добавлении строгой типизации циклические зависимости оказываются неизбежны.

Стоит также более детально рассмотреть связи между модулями regular_expression и finite_automaton. До выполнения работы класс регулярных выражений хранил по указателю эквивалентный ему конечный автомат: это было нужно для реализации определения принятия слова выражением. В то же время, у автоматов был реализован метод преобразования в регулярное выражение, и это создавало циклическую зависимость. К тому же, автоматы реализовывали интерфейс Regexable, представляющий возможность данного преобразования, и с помощью данного интерфейса были реализованы некоторые операции над автоматами. Конкретно, для объединения, конкатенации и построения замыкания Клини данные автоматы были сперва преобразованы в регулярное выражение, и лишь затем эти операции были применены.

Нам также понадобится рассмотреть причины циклов между cfg и pda. Здесь проблема кроется в том, что у классов CFG и PDA были реализованы методы преобразования друг в друга. Данные зависимости стоило разрешить в одну из сторон.

Рассмотрение существующей архитектуры и её недостатков далее позволит нам разработать улучшенный дизайн библиотеки.

4 Используемые технологии

В данной работе будет использован язык программирования *Python*, так как это основной язык библиотеки *pyformlang*. Для добавления типовых аннотаций мы будем использовать один из статических типовых анализаторов языка *Python*. Выбранный анализатор должен быть распространён среди разработчиков, быть активно поддерживаемым и широко конфигурируемым. По возможности, должны быть поддержаны более строгие выводы типов. К тому же, желательно чтобы анализатор был более производительным.

Нами были рассмотрены анализаторы *Mypy*[6] и *Pyright*[7], являющиеся наиболее распространёнными. Далее приведена таблица 1 со сравнением[8] данных вариантов по указанным критериям.

анализатор	число звёзд на *GitHub*, тыс.	число fork'ов на *GitHub*, тыс.	широко конфигурируем	более строгий вывод типов	относительная производительность
Mypy	18.5	2.8	+		1
Pyright	13.4	1.5	+	+	3–5

Таблица 1: Сравнение рассмотренных нами статических анализаторов

Несмотря на то, что *Mypy* является более популярным выбором в среде *Python* разработчиков, мы всё же остановимся на анализаторе *Pyright* из-за его строгости и производительности. Данные характеристики будут особенно важны для нас при аннотации и переработке больших объёмов кода. К тому же, вполне возможен переход с одного анализатора на другой при необходимости.

[6]Репозиторий анализатора *Mypy*: https://github.com/python/mypy
[дата обращения: 17 ноября 2024 г.]

[7]Репозиторий анализатора *Pyright*: https://github.com/microsoft/pyright
[дата обращения: 17 ноября 2024 г.]

[8]Более подробное сравнение приведённых анализаторов:
https://github.com/microsoft/pyright/blob/main/docs/mypy-comparison.md
[дата обращения: 17 ноября 2024 г.]

5 Реализация генератора строк по конечному автомату

Перед тем, как мы приступим к деталям реализации генератора, стоит пояснить наши рассуждения при выборе алгоритма генерации строк.

5.1 Выбранный подход к генерации строк

Приведённые ранее методы перечисления строк по языку могут быть полезны в рамках нашей задачи, однако для их применения потребуется пойти на некоторые компромиссы. Так, например, будет необходимо удалить в данном автомате ε-переходы, а также задать или потребовать порядок на произвольном алфавите. Из достоинств данного подхода стоит выделить выдачу строк в лексикографическом порядке. Несмотря на это, было решено реализовать генератор для недетерминированного автомата с ε-переходами, используя более простой алгоритм, отвечающий минимальным заданным требованиям.

Нами был выбран алгоритм, основанный на обходе в ширину, не учитывающий порядок выданных строк. Несмотря на то, что данный подход оказался неэффективен на больших данных по результатам эксперимента из статьи [1], вполне возможно, что это обусловлено сортировкой вычисленной кросс-секции. К тому же, библиотека *pyformlang*, направлена на обучение, а не на производительность, поэтому эффективность на малых данных можно посчитать более важным фактором при выборе метода генерации.

И так, мы подходим к более подробному описанию реализации генератора.

5.2 Детали реализации

Идея генератора основана на обходе конечного автомата как графа в ширину, как было описано ранее. По мере обхода мы будем склады-

вать строку из символов — значений рёбер — добавляя её во фронт вместе с состоянием. Исключением здесь станет символ ε, который к строке добавлять не нужно. При обходе мы должны учесть длину собранной строки и не продолжать работу в данном состоянии, если длина оказалась больше максимально допустимой. Так же, получившиеся конфигурации из состояния и строки мы будем запоминать, чтобы не обрабатывать их несколько раз. При попадании в конечное состояние мы будем запоминать собранную строку и выдавать её только в том случае, если она не была выдана ранее, чтобы избежать дубликатов. Стоит заметить, что обход на этом завершаться не должен, так как мы хотим рассмотреть все возможные пути в любое из конечных состояний. Визуализация работы генератора представлена на рисунке 3.

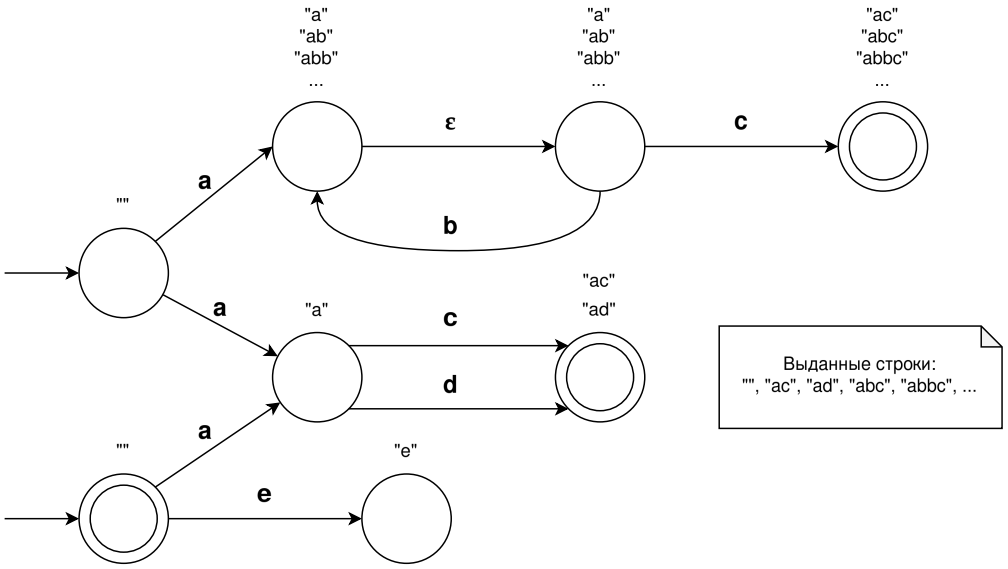

Рис. 3: Визуализация работы генератора строк по конечному автомату. Здесь над каждым состоянием указаны строки, сгенерированные на них

Корректность этой идеи можно обосновать схожестью с определением принятия слова: вместо переходов по данной строке мы собираем строку по существующим переходам.

Однако с данным подходом возникает следующая проблема: работа может не завершиться даже когда все возможные слова уже выданы. Это может произойти в том случае, когда в автомате есть цикл, не ведущий к одному из конечных состояний. Простой пример такого автомата приведён на рисунке 4. Стоит также заметить, что тривиальной провер-

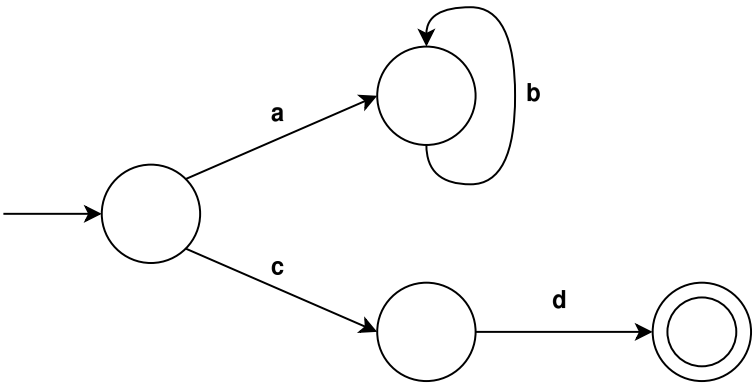

Рис. 4: Пример автомата, для которого генератор будет работать бесконечно после выдачи всех строк

ки того, что состояние уже было посещено, будет недостаточно, потому что цикл может находиться и на пути в конечное состояние. Для решения данной проблемы нам нужно для каждого состояния проверять, что из данного состояния действительно можно перейти в конечное. Далее нам будет достаточно получить все состояния, находящиеся на пути из начального в одно из конечных. Для удобства, такие состояния мы назовём *разрешающими*. И так мы подходим к описанию алгоритма получения всех разрешающих состояний по данному конечному автомату.

Алгоритм получения разрешающих состояний будет основан на обходе графа *в глубину* [12] с применением *динамического программирования* [7]. Мы воспользуемся простой идеей: если из данного состояния можно попасть в конечное, то и из всех её состояний-*предков* также существует путь в конечное состояние. Искомое множество состояний мы проинициализируем конечными состояниями автомата и начнём обход из начальных. Чтобы реализовать алгоритм в итеративном стиле, мы будем использовать стек, где вместе с текущим состоянием будем хранить предка, из которого оно было получено. Уже посещённые состояния мы будем запоминать, чтобы сохранить результаты обхода. Тогда, если состояние из вершины стека уже находится во множестве разрешающих состояний, то и его предка мы добавим в искомое множество. Иначе, если мы попали в уже посещённое состояние, то возможны два случая: либо оно не является разрешающим, либо мы нашли цикл, и ответ для данного состояния ещё не получен. Чтобы не потерять нуж-

ные нам состояния, такие переходы мы вынесем в отдельную очередь и рассмотрим после завершения обхода, аналогично проверяя целевые состояния на их принадлежность искомому множеству. При этом важно перечислять вынесенные переходы именно в порядке добавления, чтобы при необходимости мы могли добавить во множество следующие друг за другом состояния. Если же состояние не было посещено ранее, мы помечаем его как посещённое и продолжаем обход.

Сперва мы опишем шаг обхода в глубину алгоритмом 1, считая что нужные нам коллекции уже проинициализированы.

Algorithm 1 GetResolvingStep($Resolving, Visited, ToVisit, Delayed$)

Require: $Resolving \subset Q$, $Visited \subset Q$,
$ToVisit$ — не пустой стек, $Delayed$ — очередь
Ensure: $ToVisit$ обновлён
  $Previous, Current \leftarrow \text{pop}(ToVisit)$
  if $Previous \neq \text{null}$ **and** $Current \in Resolving$ **then**
    $Resolving \leftarrow Resolving \cup \{Previous\}$
    continue
  end if
  if $Current \in Visited$ **then**
    enqueue$(Delayed, (Previous, Current))$
    continue
  end if
  $Visited \leftarrow Visited \cup \{Current\}$
  $NextStates \leftarrow \bigcup_{a \in (\Sigma \cup \{\varepsilon\})} \delta(Current, a)$
  if $NextStates \neq \varnothing$ **then**
    push$(ToVisit, (Previous, Current))$
    for $Next \in NextStates$ **do**
      push$(ToVisit, (Current, Next))$
    end for
  end if

И тогда получение разрешающих состояний по данному автомату мы опишем алгоритмом 2. После рассмотрения данного алгоритма, мы можем приступить к более формальному описанию работы генератора.

На вход генератору будет поступать произвольный конечный автомат и целое число — максимальная длина генерируемых строк. Перед началом обхода в ширину нам нужно получить все разрешающие со-

Algorithm 2 GetResolving(M)

Require: $M = (Q, \Sigma, \delta, Q_0, F)$ — конечный автомат
Ensure: $Resolving \subset Q$ — множество разрешающих состояний M
  $Resolving \leftarrow F$
  $Visited \leftarrow \varnothing$
  $ToVisit$ = empty stack $\leftarrow \{(\text{null}, q_0) \mid q_0 \in Q_0\}$
  $Delayed$ = empty queue
  while $ToVisit$ is not empty **do**
    GetResolvingStep($Resolving, Visited, ToVisit, Delayed$)
  end while
  for $Previous, Current \in Delayed$ **do**
    if $Previous \neq$ null **and** $Current \in Resolving$ **then**
      $Resolving \leftarrow Resolving \cup \{Previous\}$
    end if
  end for
  return $Resolving$

стояния автомата с помощью упомянутого алгоритма. Чтобы хранить сгенерированные слова по каждому состоянию мы будем использовать словарь. Для обхода в ширину нам понадобится очередь, которую мы проинициализируем начальными состояниями автомата в парах с пустой строкой. В ходе работы, сперва мы будем проверять длину собранной строки на превышение максимальной, а также наличие данной строки в словаре по текущему состоянию. После этого мы добавим слово в словарь и обновим очередь в соответствие с алгоритмом 3.

Algorithm 3 VisitNextStates($State, Word, Resolving, ToVisit$)

Require: $State \in Q$, $Word \in \Sigma^*$ $Resolving \subset Q$, $ToVisit$ — очередь
Ensure: посещены состояния, смежные со $State$ и лежащие в $Resolving$
  for each edge adjacent to $State$ **do**
    if target(edge) $\in Resolving$ **then**
      $NewWord \leftarrow Word$
      if value(edge) $\neq \varepsilon$ **then**
        $NewWord \leftarrow NewWord +$ value(edge)
      end if
      enqueue($ToVisit$, (target(edge), $NewWord$))
    end if
  end for

И тогда работу генератора целиком мы можем описать с помощью алгоритма 4.

Algorithm 4 GetWords(M, N)

Require: $M = (Q, \Sigma, \delta, Q_0, F)$ — конечный автомат, $N \in \mathbb{Z} \cup \{+\infty\}$
Ensure: выданы строки, принимаемые M, длины не больше N

```
Resolving ← GetResolving(M)
YieldedWords ← ∅
WordsByState = empty dictionary ← {q : ∅ for q ∈ Q}
ToVisit = empty queue ← {(q_0, ε) | q_0 ∈ Q_0}
while ToVisit is not empty do
    State, Word ← dequeue(ToVisit)
    if len(Word) > N or Word ∈ WordsByState[State] then
        continue
    end if
    WordsByState[State] ← WordsByState[State] ∪ {Word}
    VisitNextStates(State, Word, Resolving, ToVisit)
    if State ∈ F and Word ∉ YieldedWords then
        yield Word
        YieldedWords ← YieldedWords ∪ {Word}
    end if
end while
```

В ходе работы мы также проверяем, было ли слово выдано ранее, чтобы не выдавать его дважды. Максимальной длиной по умолчанию мы можем считать $+\infty$: если на каждом пути в конечное состояние нет циклов, то алгоритм всегда завершит работу.

После рассмотрения реализации генератора, мы можем подойти к задаче добавления типовых аннотаций в код библиотеки *pyformlang*.

6 Добавление типовых аннотаций

В данном разделе мы опишем некоторые проблемы, обнаруженные и решённые в ходе добавления типовых аннотаций.

До выполнения данной работы в *pyformlang* существовали все необходимые представления таких объектов, как состояния и символы. Эти объекты создавались по входным данным публичных функций, хранились в соответствующим им классах, а также служили выходными значениями. Одним из важных вопросов стала проблема выбора требуемого формата входных данных. В большинстве алгоритмов, представленных в *pyformlang*, с такими объектами используются структуры данных, для которых требуется хеширование: реализации множеств и словарей. Поэтому, для входных данных было решено требовать формат Hashable, как минимальный допустимый.

Функциональности данных объектов, однако, заметно отличались, и следовало привести их к единому виду. Для этого был создан класс FormalObject, от которого будут наследоваться остальные. Так же, для устранения типовых несоответствий в одном из конвертеров PDA объектов было решено переработать связи между данными объектами. Отношения были переработаны следующим образом: стековый символ мы стали наследовать от символа входного алфавита, а ε-символ от стекового символа. Итоговые связи показаны на рисунке 5. Другой вопрос заключается в том, в каком случае объекты должны быть эквивалентны. Для большей строгости, было решено считать равными объекты одного типа с равными значениями, то есть состояние и символ не будут эквивалентны в любом случае.

Однако не все модули использовали рассмотренные нами объекты: в модулях indexed_grammar и fst данные хранились в произвольном виде. Для большей строгости, модуль indexed_grammar был модифицирован с использованием терминалов и нетерминалов КС грамматик, а в модуле fst было решено использовать состояния и символы конечных автоматов. Состояния и символы, однако, находились в модуле finite_automaton, который уже зависел от fst, и было бы странно их пе-

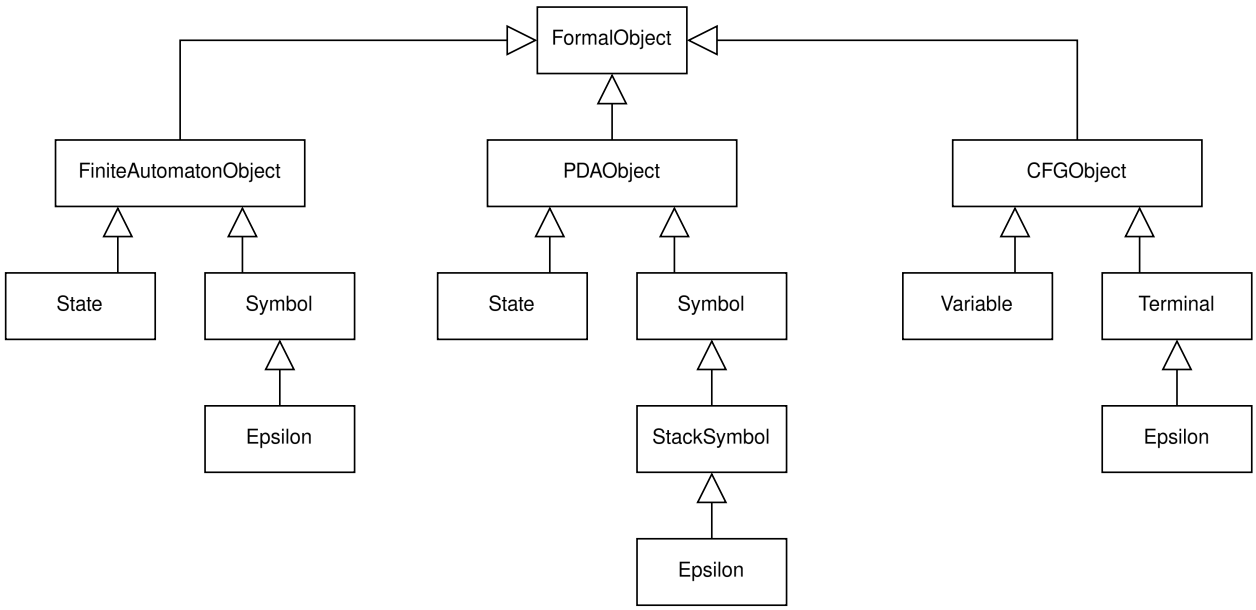

Рис. 5: Итоговые зависимости объектов автоматов и грамматик

реносить. Это добавляет ещё одну циклическую зависимость, которую нам далее предстоит устранить.

Стоит заметить, что серьёзных несоответствий при добавлении аннотаций замечено не было, а мелкие замечания были устранены. Так, например, в генераторе строк по контекстно-свободным грамматикам тип выдаваемых слов был исправлен на список терминалов. Также, чтобы привести функциональность к более однородному виду, в ходе работы были добавлены некоторые недостающие простые функции: для большинства классов были реализованы методы копирования, добавления и удаления переходов или правил, а также их итерации.

Для каждой публичной функции были добавлены аннотации, однако для использования необходимых типов требуется импортировать соответствующий им модуль библиотеки, требуя при этом инициализацию данного модуля. Учитывая существующие циклические зависимости библиотеки, чтобы модифицированный код мог быть собран без ошибок, нам потребуется данные циклы устранить. И так мы подходим к задаче переработки архитектуры библиотеки *pyformlang*.

7 Улучшение архитектуры

В данном разделе мы опишем наши рассуждения при разработке модифицированной архитектуры и некоторые детали её внедрения.

Первым делом были переработаны связи между классами модуля finite_automaton. Единственным разумным выходом для устранения указанных ранее циклов стала переработка методов трансформации автоматов следующим образом: новые автоматы мы будем строить в наследниках, передавая им более общие реализации. Для этого были добавлены статические методы from_epsilon_nfa и from_nfa, использующие существующую реализацию данных преобразований. Так же, в ходе работы были переработаны функции перехода, использующиеся в автоматах, что в некоторой степени упростило существующий код. Для этого был добавлен интерфейс TransitionFunction, а детерминированную функцию перехода мы стали наследовать от недетерминированной. Зависимости основных классов модуля finite_automaton после переработки представлены на рисунке 6.

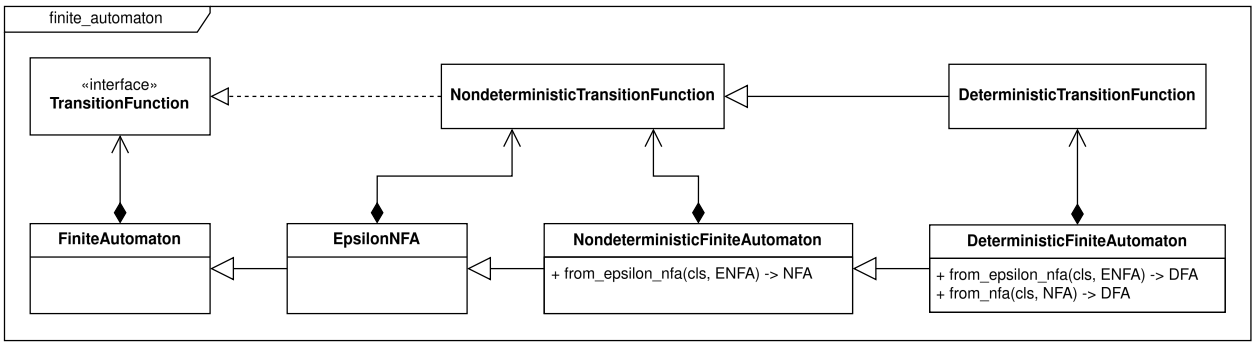

Рис. 6: Зависимости классов модуля finite_automaton после выполнения работы

Далее мы перешли к устранению циклов между finite_automaton и regular_expression. Так как класс, представляющий регулярные выражения, уже был реализован с помощью недетерминированных автоматов, было решено переработать методы преобразования автомата в выражение. Для этого в класс Regex был добавлен метод from_finite_automaton, принимающий объект абстрактного класса, представляющий конечный автомат. Переработанные зависимости по-

казаны на рисунке 7. После этого, чтобы не терять существующую

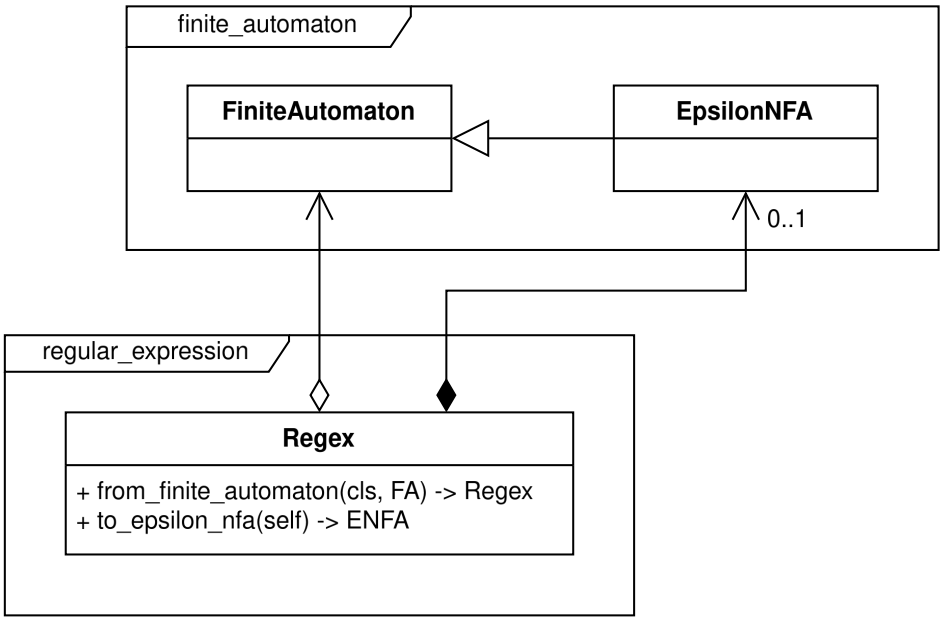

Рис. 7: Зависимости между модулями finite_automaton и regular_expression после выполнения работы

функциональность, методы объединения, конкатенации и построения замыкания Клини конечных автоматов были переписаны без использования регулярных выражений. Данные операции были реализованы с помощью построения нового автомата по исходным следующим образом.

- Объединение: создадим новое начальное состояние и добавим из него ε-переходы в начальные состояния исходных автоматов; конечными состояниями будут конечные состояния данных автоматов.

- Конкатенация: начальными состояниями будут начальные состояния первого автомата, а конечными — конечные состояния второго; добавим ε-переходы из каждого конечного состояния первого автомата в каждое начальное состояние второго.

- Замыкание Клини: создадим новое состояние, которое будет начальным и конечным, и из него добавим ε-переходы в исходные начальные состояния; для каждого конечного состояния добавим ε-переход в новое начальное.

Затем стоило разобраться с циклами между классами CFG и PDA. В итоге было решено перенести преобразование КС грамматики в автомат добавлением метода from_cfg в класс PDA. Данный выбор был сделан для более равномерного распределения функциональности. К тому же, данная зависимость больше соответствует повествованию книг по теории формальных языков, на которых основывалась библиотека [25]. Так же, чтобы разрешить циклы внутри модуля cfg, был добавлен абстрактный класс FormalGrammar, для которого была обобщена некоторая функциональность КС грамматик.

После этого, для проблемных функций пересечения требуемые типы были сужены до минимальных необходимых: до детерминированного автомата в CFG и PDA, до FST в IndexedGrammar. Итоговые связи между модулями cfg, pda и finite_automaton представлены на рисунке 8. Несмотря на кажущуюся тривиальность, данное изменение

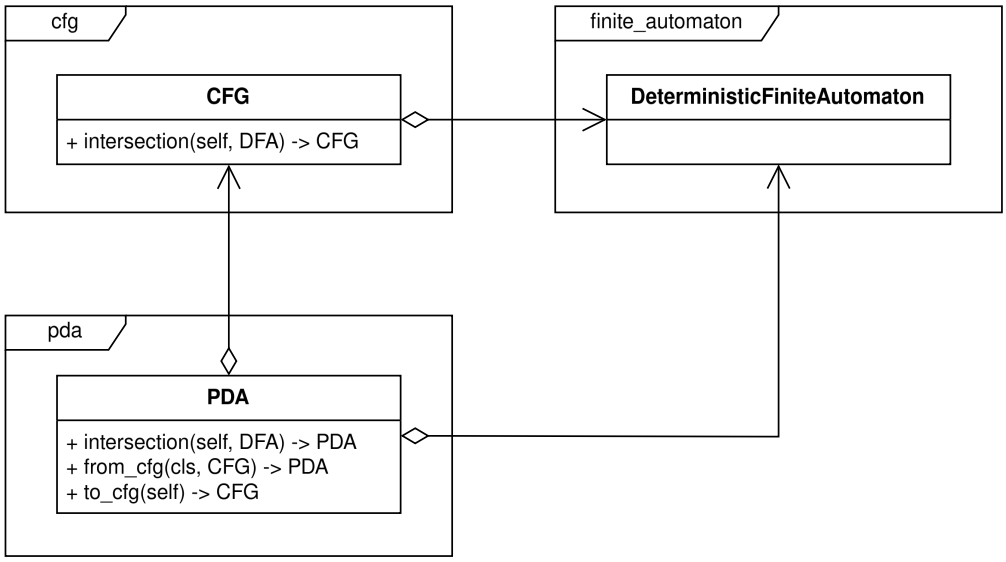

Рис. 8: Переработанные зависимости модулей cfg, pda и finite_automaton

значительно упростило итоговую архитектуру. Однако, в случае с индексированными грамматиками всё оказалось не столь просто: пересечение было реализовано именно в классе FST, и это создавало очередной цикл. Логичнее всего было перенести данную реализацию в класс IndexedGrammar, что и было сделано.

Наконец, все представления состояний и символов, а также некото-

рые связанные с ними функции, были вынесены в отдельный модуль objects. Это позволило не только устранить существующие циклы, но и лучше структурировать код библиотеки. К тому же, данный модуль не зависит от остальных, и любой другой сможет использовать его без проблем. После этого, зависимости всех модулей библиотеки мы покажем на рисунке 9.

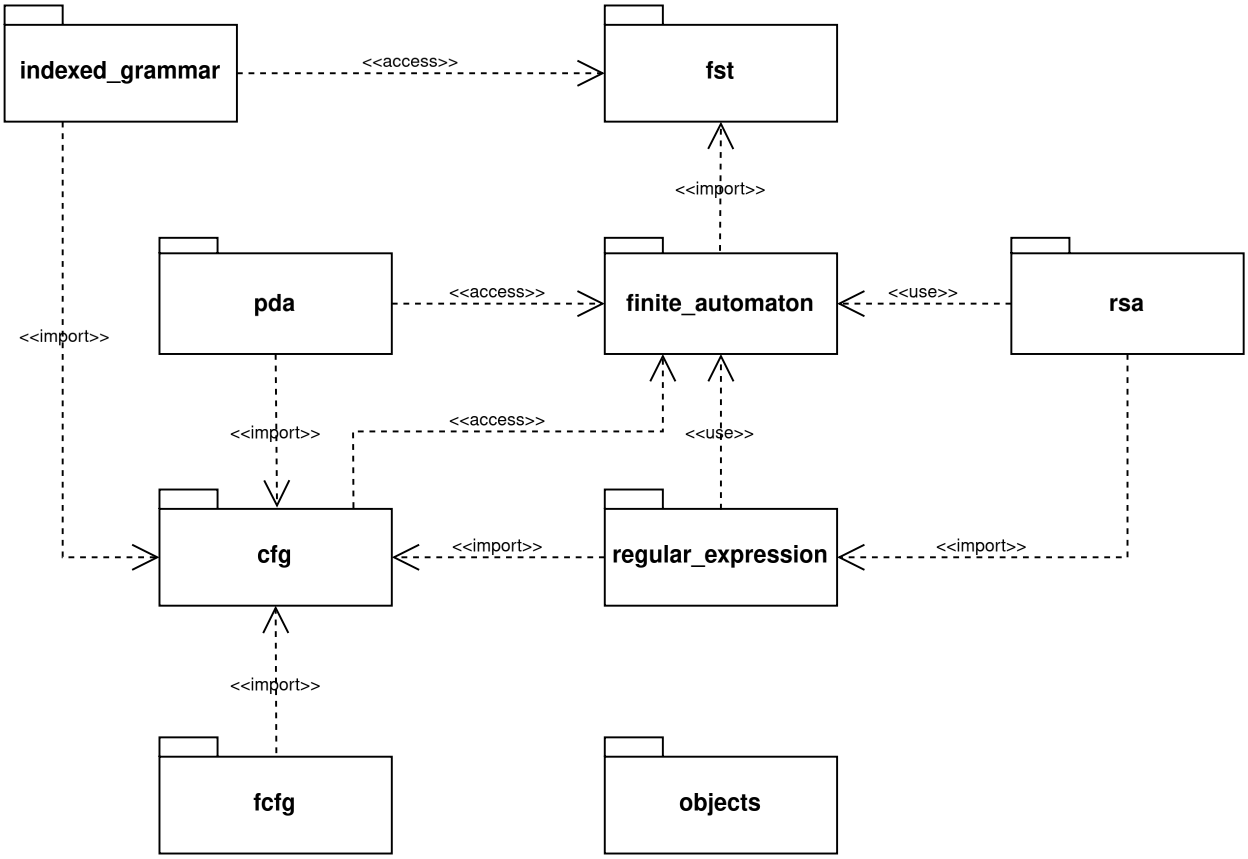

Рис. 9: Зависимости модулей *pyformlang* после выполнения работы. Здесь модуль objects может быть использован любым другим

Таким образом, циклические зависимости были устранены, и это позволит внедрить в библиотеку добавленные ранее аннотации.

8 Тестирование итоговой функциональности

После описания деталей решения поставленных задач, мы можем подойти к теме валидации нашего решения.

Вся затронутая нами функциональность была проверена с применением юнит-тестирования. Для написания и исполнения тестов был использован фреймворк *pytest*, так как он используется в библиотеке *pyformlang*. Все добавленные и существовавшие ранее тесты были исполнены для *Python* версий 3.8, 3.9 и 3.10.

Подробно стоит остановиться на методах тестирования реализованного нами генератора. Тесты на корректность были добавлены для всех существующих в библиотеке реализаций конечных автоматов: детерминированных, недетерминированных с ε-переходами и без. В ходе написания тестов были рассмотрены автоматы, различающиеся по следующим критериям:

- наличие циклов на пути в конечное состояние;
- наличие циклов, не ведущих в конечное состояние;
- наличие ε-циклов;
- число и расположение начальных и конечных состояний;
- наличие дублирующихся путей;
- принятие пустой строки;
- пустота задаваемого языка.

Полученный генератором язык с учётом дубликатов целиком сравнивался с ожидаемым. Помимо базовых сценариев, работа генератора была также проверена для нулевой максимальной длины строк, а также для максимальной длины по умолчанию.

Юнит-тесты также были добавлены для функций, реализованных нами в ходе аннотации кода и переработки архитектуры. Для переписанных операций над автоматами существующие в библиотеке тесты были дополнены.

Что касается типовых аннотаций, получившийся код был проверен с помощью анализатора *Pyright* со стандартной конфигурацией. Проверка с помощью данного анализатора была также включена в существующую CI конфигурацию проекта.

Как результат, все тесты, включая добавленные нами и существовавшие ранее, завершаются успешно. Успешно завершается также и удалённая сборка проекта, а проверка кода с помощью *Pyright* проходит без ошибок и предупреждений.

Заключение

Подводя итоги, в рамках работы над улучшением библиотеки *pyformlang* были достигнуты следующие результаты:

1. был реализован генератор строк по недетерминированному конечному автомату с ε-переходами, учитывающий дубликаты и максимальную допустимую длину строк;

2. были добавлены типовые аннотации для каждой из публичных функций, модули indexed_grammar и fst были модифицированы с использованием более строгих типов;

3. архитектура библиотеки была переработана с устранением циклических зависимостей.

Изменения архитектуры согласовывались с автором библиотеки, были открыты реквесты в основной репозиторий.

- Реквест по генератору:
  https://github.com/Aunsiels/pyformlang/pull/22.

- Реквест по аннотациям и архитектуре:
  https://github.com/Aunsiels/pyformlang/pull/27.

Реализованный нами генератор будет полезен для перечисления регулярных языков. Добавление аннотаций не только сделает более удобной работу с библиотекой, но и упростит анализ и модификацию её кода. Устранение циклических зависимостей также позволит развить и дополнить библиотеку в будущем.

Направления работы в будущем

На момент завершения работы, функциональность *pyformlang* можно было расширить некоторыми видами грамматик. Из очевидного, в библиотеке не хватает регулярных грамматик, которые имеют много

общего с конечными автоматами [16]. Так же, для обучения будут полезны более общие представления грамматик, такие как *контекстно-зависимые* грамматики [17].

К тому же, архитектуру библиотеки можно значительно прояснить структурированием существующих модулей. Возможно, стоит объединить представленные в проекте виды грамматик и автоматов.

Список литературы

[1] Ackerman Margareta. Efficient enumeration of words in regular languages // Theoretical Computer Science. — 2009. — September. — Vol. 410, no. 37. — P. 3461–3470. — URL: https://www.sciencedirect.com/science/article/pii/S0304397509002345 (дата обращения: 19 ноября 2024 г.).

[2] Aho Alfred V. Indexed Grammars — An Extension of Context-Free Grammars // Journal of the ACM. — 1968. — October. — Vol. 15, no. 4. — P. 647–671. — URL: https://dl.acm.org/doi/abs/10.1145/321479.321488 (дата обращения: 15 ноября 2024 г.).

[3] Aho Alfred. V., Johnson Stephen. C. LR Parsing // ACM Computing Surveys. — 1974. — June. — Vol. 6, no. 2. — P. 99–124. — URL: https://dl.acm.org/doi/10.1145/356628.356629 (дата обращения: 4 ноября 2024 г.).

[4] Analysis of Recursive State Machines / Rajeev Alur, Michael Benedikt, Kousha Etessami et al. // ACM Transactions on Programming Languages and Systems. — 2005. — July. — Vol. 27, no. 4. — P. 786–818. — URL: https://dl.acm.org/doi/10.1145/1075382.1075387 (дата обращения: 15 ноября 2024 г.).

[5] Arora Sanjeev, Barak Boaz. NP and NP completeness // Computational Complexity: A Modern Approach. — Cambridge University Press, 40 W. 20 St. New York, NY, United States, 2007. — January. — P. 39–64. — URL: https://theory.cs.princeton.edu/complexity/book.pdf (дата обращения: 4 ноября 2024 г.).

[6] Arora Sanjeev, Barak Boaz. The computational model — and why it doesn't matter // Computational Complexity: A Modern Approach. — Cambridge University Press, 40 W. 20 St. New York, NY, United States, 2007. — January. — P. 11–38. — URL: https://theory.cs.princeton.edu/complexity/book.pdf (дата обращения: 4 ноября 2024 г.).

[7] Bellman Richard. Dynamic Programming // Science. — 1966. — July. — Vol. 153, no. 3731. — P. 34–37. — URL: https://www.science.org/doi/abs/10.1126/science.153.3731.34 (дата обращения: 23 ноября 2024 г.).

[8] Buneman Peter, Fan Wenfei, Weinstein Scott. Path Constraints on Semistructured and Structured Data // PODS '98: Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems. — Seattle, Washington, USA, 1998. — May. — P. 129–138. — URL: https://dl.acm.org/doi/pdf/10.1145/275487.275502 (дата обращения: 4 ноября 2024 г.).

[9] Carbonell Jaime G., Michalski Ryszard S. Machine Learning: A Historical and Methodological Analysis // AI Magazine. — 1983. — September. — Vol. 4, no. 3. — P. 69–78. — URL: https://ojs.aaai.org/aimagazine/index.php/aimagazine/article/view/406 (дата обращения: 4 ноября 2024 г.).

[10] Earley Jay. An efficient context-free parsing algorithm // Communications of the ACM. — 1970. — February. — Vol. 13, no. 2. — P. 94–102. — URL: https://dl.acm.org/doi/10.1145/362007.362035 (дата обращения: 4 ноября 2024 г.).

[11] Erickson Jeff. Basic Graph Algorithms // Algorithms. — Jeff Erickson, 2019. — June. — P. 187–224. — URL: https://jeffe.cs.illinois.edu/teaching/algorithms/book/Algorithms-JeffE.pdf (дата обращения: 7 ноября 2024 г.).

[12] Erickson Jeff. Depth-First Search // Algorithms. — Jeff Erickson, 2019. — June. — P. 225–256. — URL: https://jeffe.cs.illinois.edu/teaching/algorithms/book/Algorithms-JeffE.pdf (дата обращения: 7 ноября 2024 г.).

[13] Hopcroft John E., Motwani Rajeev, Ullman Jeffrey D. Finite Automata With Epsilon Transitions // Introduction to Automata Theory, Languages, and Computation. —

Addison-Wesley, 2006. — June. — P. 72–79. — URL: https://lecture-notes.tiu.edu.iq/wp-content/uploads/2022/02/Introduction-to-Automata-Theory-Languages-and-Computations-by-J-Hopcroft-Rajeev-Motwani-Jeffrey-D.-Ullman-z-lib.org_.pdf (дата обращения: 5 ноября 2024 г.).

[14] Hopcroft John E., Motwani Rajeev, Ullman Jeffrey D. Regular Expressions and Languages // Introduction to Automata Theory, Languages, and Computation. — Addison-Wesley, 2006. — June. — P. 85–126. — URL: https://lecture-notes.tiu.edu.iq/wp-content/uploads/2022/02/Introduction-to-Automata-Theory-Languages-and-Computations-by-J-Hopcroft-Rajeev-Motwani-Jeffrey-D.-Ullman-z-lib.org_.pdf (дата обращения: 15 ноября 2024 г.).

[15] Hopcroft John E., Ullman Jeffrey D. Context-Free Grammars // Formal languages and their relation to automata. — Addison-Wesley Longman Publishing Co., Inc., United States, 1969. — January. — P. 46–67. — URL: https://dl.acm.org/doi/abs/10.5555/1096945 (дата обращения: 4 ноября 2024 г.).

[16] Hopcroft John E., Ullman Jeffrey D. Finite Automata and Regular Grammars // Formal languages and their relation to automata. — Addison-Wesley Longman Publishing Co., Inc., United States, 1969. — January. — P. 26–45. — URL: https://dl.acm.org/doi/abs/10.5555/1096945 (дата обращения: 4 ноября 2024 г.).

[17] Hopcroft John E., Ullman Jeffrey D. Grammars // Formal languages and their relation to automata. — Addison-Wesley Longman Publishing Co., Inc., United States, 1969. — January. — P. 8–25. — URL: https://dl.acm.org/doi/abs/10.5555/1096945 (дата обращения: 4 ноября 2024 г.).

[18] Hopcroft John E., Ullman Jeffrey D. Languages and Their Representations // Formal languages and their relation to automata. —

Addison-Wesley Longman Publishing Co., Inc., United States, 1969. — January. — P. 1–7. — URL: https://dl.acm.org/doi/abs/10.5555/1096945 (дата обращения: 4 ноября 2024 г.).

[19] Hopcroft John E., Ullman Jeffrey D. Pushdown Automata // Formal languages and their relation to automata. — Addison-Wesley Longman Publishing Co., Inc., United States, 1969. — January. — P. 68–79. — URL: https://dl.acm.org/doi/abs/10.5555/1096945 (дата обращения: 4 ноября 2024 г.).

[20] Hopcroft John E., Ullman Jeffrey D. Turing Machines // Formal languages and their relation to automata. — Addison-Wesley Longman Publishing Co., Inc., United States, 1969. — January. — P. 80–101. — URL: https://dl.acm.org/doi/abs/10.5555/1096945 (дата обращения: 4 ноября 2024 г.).

[21] Hopcroft John E., Ullman Jeffrey D. Turing Machines: The Halting Problem, Type 0 Languages // Formal languages and their relation to automata. — Addison-Wesley Longman Publishing Co., Inc., United States, 1969. — January. — P. 102–114. — URL: https://dl.acm.org/doi/abs/10.5555/1096945 (дата обращения: 4 ноября 2024 г.).

[22] Inference of Regular Expressions for Text Extraction from Examples / Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, Fabiano Tarlao // IEEE Transactions on Knowledge and Data Engineering. — 2016. — January. — Vol. 28, no. 5. — P. 1217–1230. — URL: https://ieeexplore.ieee.org/abstract/document/7374717 (дата обращения: 4 ноября 2024 г.).

[23] Mohri Mehryar. Finite-State Transducers in Language and Speech Processing // Computational Linguistics. — 1997. — June. — Vol. 23, no. 2. — P. 269–311. — URL: https://aclanthology.org/J97-2003/ (дата обращения: 15 ноября 2024 г.).

[24] Mäkinen Erkki. On Lexicographic Enumeration Of Regular And Context-Free Langugages // Acta Cybernetica. — 1997. — January. —

Vol. 13, no. 1. — P. 55–62. — URL: https://www.researchgate.net/publication/220123278_On_Lexicographic_Enumeration_Of_Regular_And_Context-Free_Langugages (дата обращения: 19 ноября 2024 г.).

[25] Romero Julien. Pyformlang: An Educational Library for Formal Language Manipulation // The 52nd ACM Technical Symposium on Computer Science Education. — Virtual Event, USA, 2021. — March. — P. 576–582. — URL: https://pure.mpg.de/rest/items/item_3286205_5/component/file_3493432/content (дата обращения: 3 ноября 2024 г.).

[26] Sakakibara Yasubumi. Grammatical inference in bioinformatics // IEEE Transactions on Pattern Analysis and Machine Intelligence. — 2005. — May. — Vol. 27, no. 7. — P. 1051–1062. — URL: https://ieeexplore.ieee.org/abstract/document/1432739 (дата обращения: 4 ноября 2024 г.).

[27] Schmidt Albrecht, Windhouwer Menzo, Kersten Martin. Feature Grammars // Proceedings of the International Conference on Systems Analysis and Synthesis. — 1999. — August. — URL: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=475f5e810b6193673ee81b1058cf5b7d6efe42ce (дата обращения: 15 ноября 2024 г.).

[28] Wilson Robin J. Definitions and examples // Introduction to Graph Theory. — Prentice Hall, 1996. — May. — P. 8–25. — URL: https://www.maths.ed.ac.uk/~v1ranick/papers/wilsongraph.pdf (дата обращения: 6 ноября 2024 г.).