Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

“САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ”

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

РАЗРАБОТКА БИБЛИОТЕКИ НА ЯЗЫКЕ RUST ДЛЯ РАБОТЫ С ФОРМАТОМ ДАННЫХ TRANSIT

Автор Фомин Евгений Михайлович _____
(Фамилия, Имя, Отчество) (Подпись)

Направление подготовки (специальность)_____
(код, наименование)
09.09.04 Программная инженерия

Квалификация магистр
(бакалавр, магистр)*

Руководитель ВКР Радченко И.А., к.т.н., доцент _____
(Фамилия, И., О., ученое звание, степень) (Подпись)

К защите допустить

Руководитель ОП Муромцев Д.И., к.т.н., доцент _____
(Фамилия, И.О., ученое звание, степень) (Подпись)

“_____”_____ 20 ____г.

Санкт-Петербург, 2019 г.

Студент Фомин Е.М. Группа Р4217 Факультет ПИиКТ
(Фамилия, И. О.)

Направленность (профиль), специализация _____
09.09.04 Технологии промышленного программирования

Консультант (ы):
а) _____ _____
(Фамилия, И., О., ученое звание, степень) (Подпись)

б) _____ _____
(Фамилия, И., О., ученое звание, степень) (Подпись)

ВКР принята “____”_____20 ____г.

Оригинальность ВКР _____%

ВКР выполнена с оценкой _____

Дата защиты “____”_____20 ____г.

Секретарь ГЭК _____ _____
(ФИО) (подпись)

Листов хранения _____

Демонстрационных материалов/Чертежей хранения _____

Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

“САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ”

УТВЕРЖДАЮ

Руководитель ОП

Муромцев Д.И. _____

(Фамилия, И.О.) (подпись)

«____» «_____» 20____ г.

З А Д А Н И Е
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ

Студенту Фомин Е.М. **Группа** P4217 **Факультет** ПИиКТ

Руководитель ВКР Радченко Ирина Алексеевна, к.т.н., доцент, ПИиКТ

(ФИО, ученое звание, степень, место работы, должность)

1 Наименование темы: Разработка библиотеки на языке Rust для работы с форматом данных Transit

Направление подготовки (специальность) 09.04.04. Программная инженерия

Направленность (профиль) 09.04.04. Технологии промышленного программирования

Квалификация _____

2 Срок сдачи студентом законченной работы «_____» «_____» 20_____г.

3 Техническое задание и исходные данные к работе

Требуется спроектировать и разработать библиотеку на языке Rust, позволяющую разработчикам приложений на этом языке выполнять сериализацию и десериализацию данных в формате Transit. Библиотека должна быть ориентирована на пользователя и предоставлять обоснованные механизмы по-умолчанию для упрощенного внедрения в существующие проекты.

4 Содержание выпускной квалификационной работы (перечень подлежащих разработке вопросов)

1. Провести классификацию схожих по назначению форматов сериализации и выделить особенности формата Transit
2. Изучить существующие решения для языка Rust в вопросе сериализации данных
3. С учетом особенностей формата Transit спроектировать интерфейс разрабатываемой библиотеки

4. Реализовать основные механизмы формата данных
5. Поддержать стандартные типы данных
6. Реализовать поддержку пользовательских структур по-умолчанию

5 Перечень графического материала(с указанием обязательного материала) презентация

6 Исходные материалы и пособия

Klabnik S., Nichols C.: The Rust Programming Language. No Starch press, 2018. 488p.
Hickey. R. “Transit”, Cognitect, 22 July. 2014, blog.cognitect.com/blog/2014/7/22/transit
Cognitect, Transit Format, (2019), GitHub repository: github.com/cognitect/transit-format

7 Дата выдачи задания «____» «_____» 20____г.

Руководитель ВКР_____
(подпись)

Задание принял к исполнению_____ «____» «_____» 20____г.
(подпись)

Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

“САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ”

АННОТАЦИЯ

ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

Студент Фомин Евгений Михайлович
(ФИО)

Наименование темы ВКР:
Разработка библиотеки на языке Rust для работы с форматом данных Transit

Наименование организации, где выполнена ВКР Университет ИТМО

ХАРАКТЕРИСТИКА ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

1 Цель исследования разработка программного решения для работы с данными в формате Transit в приложениях, написанных на языке Rust

2 Задачи, решаемые в ВКР проектирование и реализация библиотеки, улучшение пользовательского опыта, проведение тестирования

3 Число источников, использованных при составлении обзора

4 Полное число источников, использованных в работе 14

5 В том числе источников по годам

Отечественных			**Иностранных**		
Последние 5 лет	От 5 до 10 лет	Более 10 лет	Последние 5 лет	От 5 до 10 лет	Более 10 лет
			11	1	2

6 Использование информационных ресурсов Internet 8
(Да, нет, число ссылок в списке литературы)

7 Использование современных пакетов компьютерных программ и технологий (Указать, какие именно, и в каком разделе работы)

Пакеты компьютерных программ и технологий	**Параграф работы**
Kakoune	
Texlive	
Gnuplot	3.2
PlantUML	3.2
Cargo	

8 Краткая характеристика полученных результатов была разработана библиотека на языке Rust, позволяющая сериализовать и десериализовать данные в формате Transit как стандартных для формата типов, так и пользовательских

9 Полученные гранты, при выполнении работы
(Название гранта)

10 Наличие публикаций и выступлений на конференциях по теме выпускной работы нет
(Да, нет)

а) 1 _____

(Библиографическое описание публикаций)

2_____

3_____

_____

_____

_____

б) 1_____

(Библиографическое описание выступлений на конференциях)

2_____

3_____

_____

_____

_____

_____

Студент Фомин Е.М. _____

(ФИО) (подпись)

Руководитель ВКР Радченко И.А. _____

(ФИО) (подпись)

“_____”_____20___г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В проектировании и разработке программного обеспечения наблюдается тенденция перехода от монолитной архитектуры к использованию небольших, слабо связанных модулей — микросервисов. С распространением оркестрации контейнеров высокая гранулярность модулей позволяет рационально распределить вычислительные ресурсы с помощью масштабирования. Данный подход применим также и за пределами веб-сервисов, где модули могут быть запущены на одной физической машине без средств виртуализации и прибегать к межпроцессному взаимодействию.

В обоих случаях такой подход имеет общие преимущества:

- Кодовая база отдельно взятого модуля сравнительно небольшая и гарантированно не имеет зависимостей от других модулей
- Независимый процесс сборки и тестирования
- Аварийное завершение работы модуля не ведет к завершению приложения
- Модули могут быть написаны на разных языках программирования[1]

Наиболее примечателен последний пункт. Разработка любой технологии преследует решение определенного спектра задач, поэтому каждый из модулей может быть написан на том языке программирования, который наиболее подходит.

Для обмена информацией между модулями необходим общий формат данных. Если модули написаны на разных языках программирования, для каждого из них требуется поддержка этого формата.

В данной работе будет рассмотрена реализация поддержки одного из таких форматов — Transit, для системного языка программирования Rust. Вновь ссылаясь на преимущества модульной архитектуры, возможность использовать разные языки программирования не требует для перехода на новую техноло-

гию вести разработку с нуля, поэтому создание нового модуля может вестись на ранее не задействованном в проекте языке программирования без серьезных затрат. Таким образом, рассмотреть язык Rust можно при наличии конкретных требований к производительности и времени отклика либо как новую, но хорошо себя зарекомендовавшую технологию.[2]

1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1 Форматы сериализации и их поддержка

Сериализация — процесс перевода структур данных в битовую последовательность. Данный процесс обратим, такая процедура называется десериализацией.

Сериализация может применяться для дампа какого-либо состояния приложения, например, модели, полученные при использовании методов машинного обучения, которые можно загружать в память по необходимости, так же сериализация применяется в сервис-ориентированных архитектурах или в межпроцессном взаимодействии.

Ввиду распространения клиент-серверной архитектуры и микросервисов обмен данными по сети становится интенсивнее и необходимо поддержать общий, по крайней мере в рамках пары связанных сервисов, формат обмена данными.

Далеко не все форматы предназначены для обмена данными между приложениями, разработанными с применением различных технологий, в частности, языков программирования. Например, сериализованный с помощью Pickle датафрейм рассчитан на десериализацию средствами языка Python. [3]

Большая часть форматов менее подвержена данной проблеме, так как для них существуют библиотеки сериализации и десериализации для наболее популярных языков программирования, однако набор типов данных и возможных значений сильнее ограничен ввиду намеренной простоты или невозможности покрыть неограниченное множество типов данных различных языков программирования. Решением данной проблемы является использование схемы, которой соответствуют все взаимодействующие модули, предполагая некоторую семантику для значения по известному пути.

1.2 Самоописательность форматов сериализации

Формат, предоставляя некоторый ограниченный набор типов, позволяет пользователю построить достаточно сложную структуру, как-либо отражающую доменную область. Пользователи, интерпретируя полученные данные в конкретном формате, не всегда в состоянии однозначно привести структуру к какому-либо соответствующему для данного языка программирования или доменной области типу. Например, для JSON, представленном на рис. 1, можно привести несколько примеров недосказанности:

```
{
  "name": "Van",
  "related": [
    "Billy",
    "Mark",
    "Steve"
  ],
  "registered": "10-11-1995",
  "skills_by_rates": {
    "3": [
      "Linux",
      "Git"
    ],
    "2": [
      "Performance artist"
    ],
    "1": [
      "Rust"
    ]
  }
}
```

Рисунок 1 — Пример JSON-объекта

– Поле `related` содержит в себе множество, но выражено массивом

– Поле `registered` скорее дата, нежели строка, более того, формат даты не определен

– Поле `skills_by_rates` выражает группировку по значению, однако

числовые значения выражены строками, поскольку являются ключами, а могут и по смыслу являться строками

Для каждого из примеров для однозначной десериализации таких данных необходима дополнительная информация, какой-либо контекст или соглашение, которому все должны следовать. В противоположность подходу, когда приложение придерживается схемы, существуют самоописывающиеся форматы. [4]

Здесь следует отметить, что на уровне базовых типов данных, предоставляемых JSON, формат самоописывается: при десериализации не требуется знать точную структуру объекта наперед, чтобы получить корректное представление в терминах базовых типов. Однако на уровне бизнес логики это не соответствует желаемому представлению: без надлежащего контекста даты останутся строками как и числовые ключи в объекте. Некоторые форматы данных используют файл схемы, переиспользуемый во всех модулях, тем самым избегая несоответствий в реализации, поэтому это может быть оправданной альтернативой самоописательным форматам.

1.3 Сравнение форматов сериализации данных

Для анализа были использованы следующие критерии:

Бинарность Формат данных не обладает соответсвующим текстовым представлением

Является надмножеством Наращивает возможности существующего формата данных, переиспользуя его базовые типы

Расширяемость Множество типов данных не ограничено

Самоописательность Наличие контекста/схемы не требуется для десериализации; в данном случае рассматриваются выражаемые данным форматом типы, а не надстройки, управляемые соглашениями

	Бинарность	Надмножество	Расширяемость	Самоописательность
JSON	—	—	—	+
MsgPack	+	—	—	+
ProtoBuf	+	—	+	—
ION	+/—	+	—	+
Transit	+/—	+	+	+

Рисунок 2 — Сравнение некоторых форматов сериализации

Сравнительная таблица представлена на рис. 2.

Использование бинарных форматов сокращает объем передаваемых данных, следовательно, снижая нагрузку на сеть, в ущерб человекочитаемости, поэтому в условиях отсутствия требований к последнему их использование предпочтительнее. Важным исключением являются сервисы, выполняемые в браузере: парсер выбранного формата должен быть написан на JavaScript, поэтому JSON и форматы поверх него выигрывают в скорости обработки из-за встроенного в браузер парсера [5].

На рис. 2 строки с ION и Transit отмечены как +/—: данные форматы имеют обе реализации, бинарная для ION[6] и MsgPack для Transit, и текстовое представление в виде JSON для обоих[7].

Самоописательные форматы удобнее для прототипирования и в дальнейшем не требуют содержать схему актуальной, однако следование неявным соглашениям, по которым модули работают с этими данными, никак не проверяется. Таким образом, явное внесение семантики в структуру данных положительно сказывается на надежности приложения и требует меньше действий от разработчика по их обработке.

Успешная модель JSON и применимость в браузере позволила ему стать распространенным форматом сериализации. В рамках данной работы предлагается использовать относительно небольшую надстройку над ним — Transit в

варианте реализации JSON, явном и сокращенным, первый из которых удобен для разработки. Одной из целей работы является также расширение экосистемы языка Rust, для которого отсутствует реализация этого формата.

1.4 Формат Transit

Transit — это формат и набор библиотек для транспортировки значений между приложениями, написанными на разных языках программирования. Формат обладает подробной и наглядной спецификацией, что позволяет упростить его реализацию. Он предоставляет набор базовых элементов и расширения для представления типизированных значений. Механизм расширений открыт и позволяет программам, использующим Transit, добавлять новые типы для своих нужд. В ином случае пользователям необходимо опираться на схемы данных, соглашения и контексты, чтобы передавать объекты, не входящие в базовые типы.

Дизайн формата предполагает использование поверх форматов, для которых уже существуют производительные процессоры, в данном случае — JSON. Все расширения строятся поверх базовых типов и бинарное представление не используется, если это не конкретно бинарный тип. Таким образом, Transit всегда может быть декодирован, даже если приложение может не знать о некоторых расширениях. Основная направленность формата — коммуникация между программами: в то время как есть возможность просматривать кодированные данные в формате JSON (JSON-Verbose), человекочитаемость не является целью.

Следует принять во внимание, что типы в Transit задают семантику и не являются частью системы типов, так же отсутствуют ссылочные типы.

Базовые типы и встроенные расширения покрывают большую часть структур данных, общих для многих языков программирования. Хорошая реализация должна задать соответствие этих типов и тех, что относятся к языку программирования, принимая во внимания их семантику.

Формат позволяет не только расширить набор типов, но и сократить объем данных к передаче и хранению — для этого используется кэширование. Например, ключи с длинами более трех символов подменяются на двух или трехсимвольный идентификатор, который далее подставляется.

Помимо расширения набора типов формат Transit снимает ограничения JSON на строковые ключи и скалярные значения на верхнем уровне.[8]

1.4.1 Алгоритм работы

Далее на рис. 3 и 4 представлены диаграммы записи и чтения соответственно.

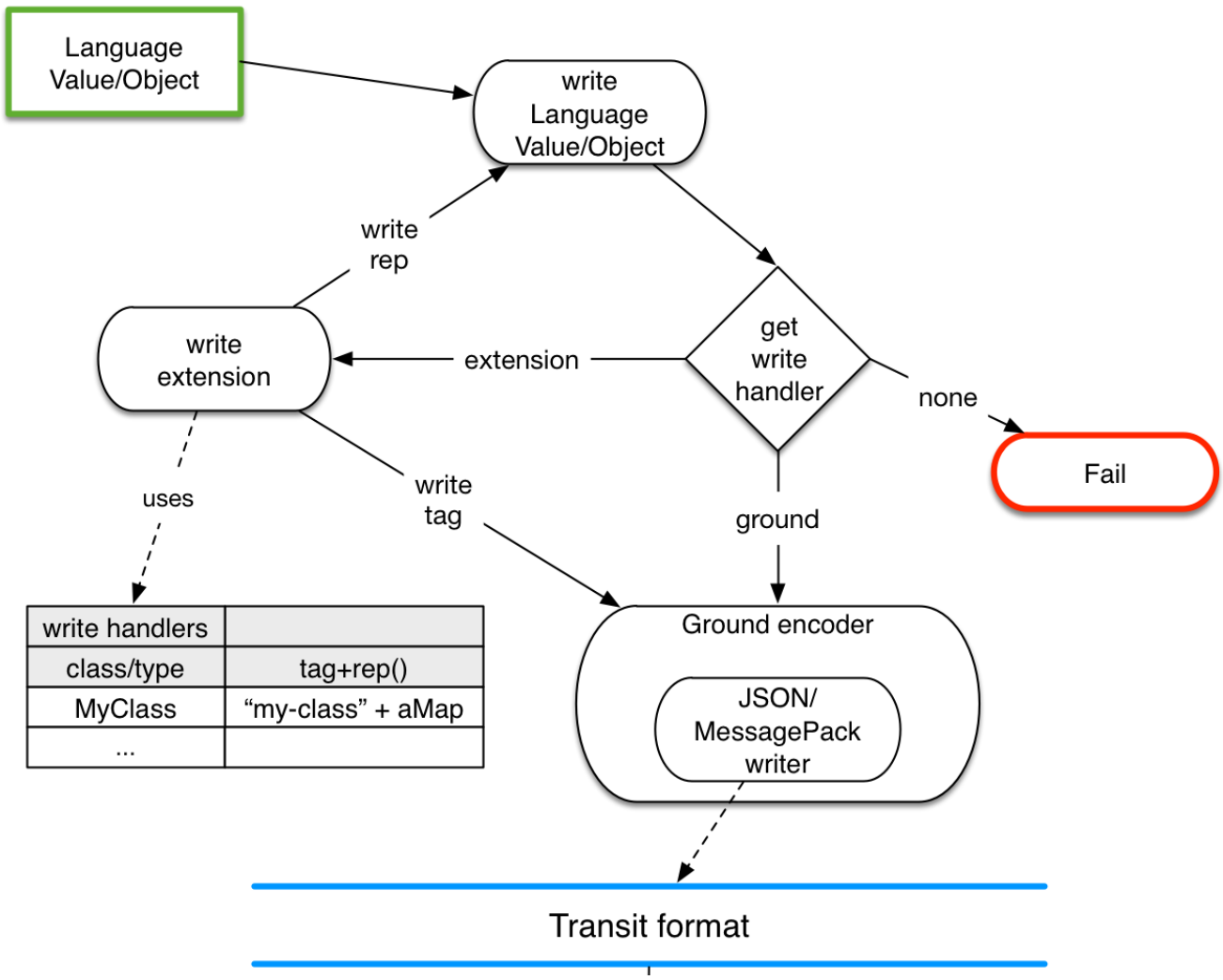

Рисунок 3 — Transit запись

Для базовых типов Transit определено однозначное представление в ви-

де соответствующих типов JSON или в виде строки с префиксом. Указанные в спецификации соответствия отправляются в JSON writer. Для типов расширений требуется выполнить рекурсивную процедуру формирования структур с требуемыми тегами нужной вложенности вплоть до базовых типов.

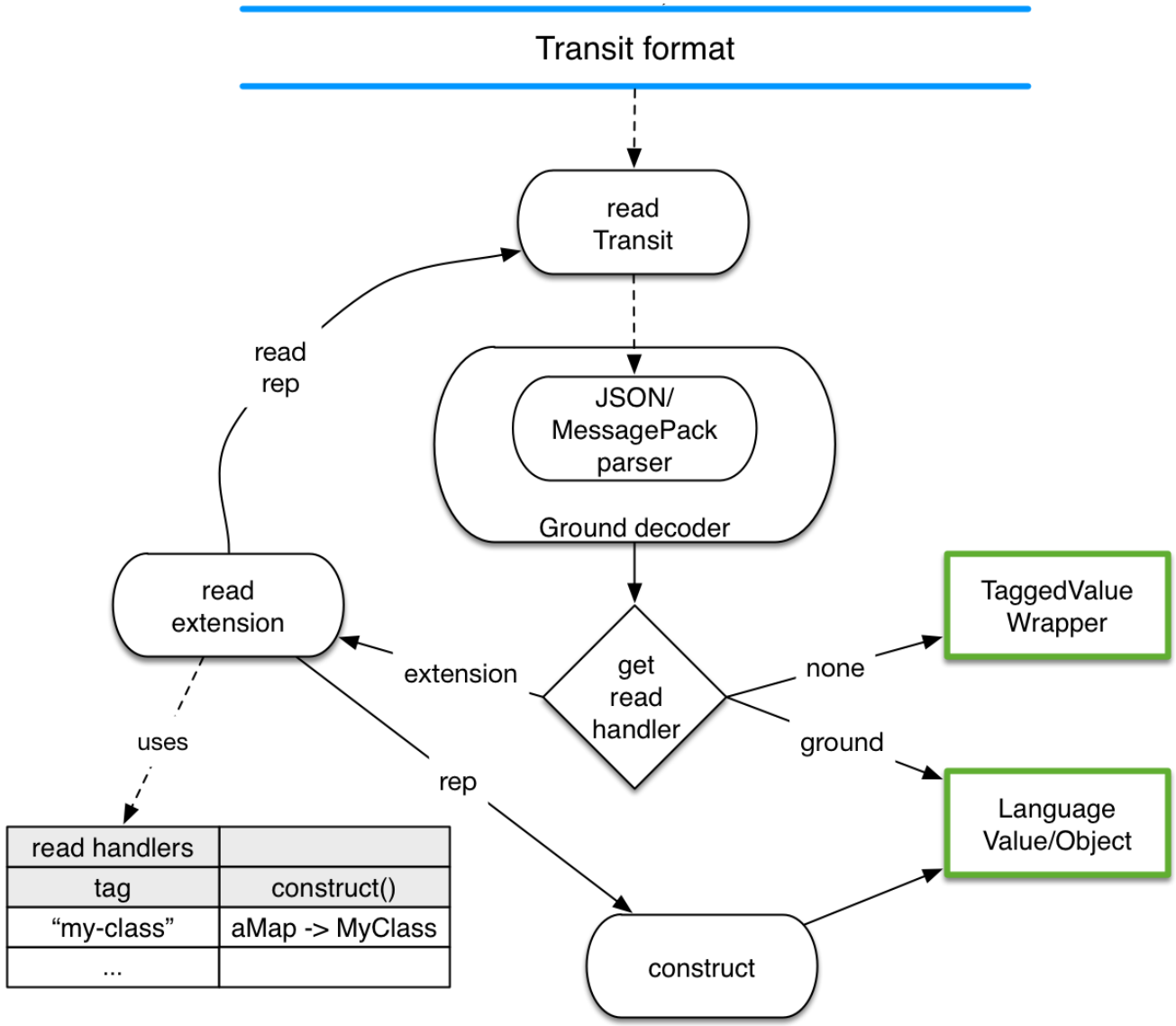

Рисунок 4 — Transit чтение

Очевидно, что данные в формате Transit являются корректным объектом JSON, это означает, что предварительно нужно получить преобразованные данные, с которыми можно работать в коде программы. По мере разбора объекта допустимы три случая:

1. для полученного тега нет ассоциированной структуры, тогда данные обо-

рачиваются в обобщенную обертку TaggedValueWrapper и далее могут использоваться, например, в отладке

2. тип относится к базовым и преобразуется к типу языка
3. тип расширения разбивается рекурсивно вплоть до базовых, далее собирая вложенную структуру в языке

   Остальные детали реализации будут рассмотрены позднее.

2 РЕАЛИЗАЦИЯ НА SERDE

Основное преимущество формата Transit — возможность работы с ним отовсюду, где можно работать с JSON, переиспользуя производительные парсеры. По сути, библиотека поддержки Transit позволяет автоматически приводить к конкретным типам данных языка автоматически из уже десериализованного JSON и наоборот. Таким образом, необходимо изучить готовые решения для работы с JSON и сериализации в целом для использования всех преимуществ реализуемого формата данных.

2.1 Обзор Serde

Serde — это фреймворк для сериализации и десериализации структур данных Rust эффективно и обобщенно. Экосистема Serde состоит из структур данных, над которыми известны функции сериализации и десериализации и форматов данных, функции приведения к которым так же известны. Serde обеспечивает слой, который позволяет любой поддерживаемой структуре данных быть сериализованной и десериализованной любым из поддерживаемых форматов данных.

Хоть и конечная цель — поддержка (де)сериализации в рамках Serde для Transit, следует учесть, что данный фреймворк не включает в себя парсер. Для работы со слабо типизированными JSON структурами в языке Rust есть библиотека Serde JSON, которая включает в себя парсер, рендерер из этих структур и удобные средства для их описания [9].

Фреймворк Serde является стандартом де-факто для сериализации и десериализации, но так как Transit имеет отличиную специфику от поддерживаемых форматов по умолчанию, таких как JSON, YAML или XML, он может быть менее применим, однако его изучение позволяет позаимствовать идеи, специфичные для Rust в вопросах сериализации, и лучшие практики.

2.2 Serde JSON

Так как Serde не включает в себя парсеры, совместимый парсер JSON распространяется в виде отдельной библиотеки. Serde JSON экспортирует тип `Value`, представленный на рис. 5, являющийся типом-суммой всех типов, которые включает в себя JSON.

```
enum Value {
    Null,
    Bool(bool),
    Number(Number),
    String(String),
    Array(Vec<Value>),
    Object(Map<String, Value>),
}
```

Рисунок 5 — Объявление JSON Value

Библиотека позволяет осуществлять разбор JSON-текста в указанный тип `Value` и наоборот. В дальнейшем с целью переиспользования конечной и отправной точками сериализации и десериализации должен являться тип `Value`, оставляя работу с текстом за библиотекой Serde JSON.

2.3 Реализация на Serde

Являясь системным языком, Rust имеет минимальную среду выполнения, осуществляя почти все проверки на этапе компиляции, поэтому информация о типах, а также значения безразмерных типов, во время выполнения отсутствует. Когда как многие языки программирования для сериализации полагаются на рефлексию, Rust использует мощную систему трейтов, напоминающую в первом приближении интерфейсы из объектно-ориентированных языков.[10]

Структура данных, для которой известно, как нужно осуществлять сериализацию и десериализацию, это та, для которой реализованы трейты `Serialize`, объявление которого представлено на рис. 6, и `Deserialize`, объявленный как на рис. 7, из фреймворка Serde.

```rust
pub trait Serialize {
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
    where
        S: Serializer;
}
```

Рисунок 6 — Объявление Serialize

```rust
pub trait Deserialize<'de>: Sized {
    fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
    where
        D: Deserializer<'de>;
}
```

Рисунок 7 — Объявление Deserialize

2.3.1 Serde Data Model

Согласно объявлению трейта Serialize в единственный метод передается объект обобщенного типа, для которого реализован трейт Serializer, а тип возвращаемого в результате сериализации значения так же указан в конкретной реализации трейта Serializer для типа S. Такое объявление явно следует из Serde Data Model — API, с помощью которого структуры данных на языке Rust сопоставляются с представлениями в форматах данных. Использование Serializer с ограничением на возвращаемый тип для Serialize не позволяет выразить структуру в конечном формате данных, поскольку он заранее не известен и не влияет на реализацию Serialize, то есть для типа есть одна реализация Serialize, которая затем параметризуется Serializer, соответствующим конкретному формату. Здесь Serde Data Model выступает промежуточной моделью, абстрагирующей конкретный формат данных; важно отметить, что во времени выполнения данных в промежуточной модели не существует и преобразования идут напрямую в конечное представление.

Задача Serializer — выразить структуру в терминах 29 типов Serde Data Model:

- 14 примитивных типов, аналогичные встроенным в язык Rust
- UTF-8 строки, причем во время десериализации указывается, требуется ли строку копировать или использовать ссылку на пришедшие данные
- массив байтов
- Option<T>
- Unit/Unit struct/unit variant — «ничего» само по себе или в исполнении структуры или варианта перечисления без вложенного значения
- Newtype/Newtype variant — структура или вариант перечисления с безымянным полем
- Seq — гетерогенная коллекция нефиксированной длины
- Tuple/Tuple struct/Tuple variant — кортеж или Newtype с несколькими полями
- Map/Struct/Struct variant — аналогично Tuple, но с именованными полями

Таким образом, Serialize ожидает от Serializer методов, позволяющих выразить каждый из перечисленных типов.[11]

2.4 Сериализация базовых типов

Поддерживаемые форматом Transit типы можно классифицировать следующим образом:

- По наличию в JSON
  - Базовый — такому типу есть соответствующий из JSON
  - Тип-расширение — значения такого типа построены из базовых типов и не имеют соответствующей семантики в JSON
- Скалярность/Агрегатность

2.4.1 Скалярные типы

Так как базовые типы в Transit соответствуют оным в JSON, а также воспроизводятся в терминах Serde Data Model, реализация Serialize для них состоит из вызова соответствующего метода Serializer вида `serializer.serialize_*` для нужного типа. Ввиду такой особенности формата как *quoting*, преобразованное значение скалярного типа может быть дополнительно обернуто для получения корректного JSON-объекта. По смыслу этот механизм схож с одноименной особенностью языков семейства Lisp. [13]

2.4.2 Композитные типы

Базовыми композитными типами являются массив и объект (словарь). В Serde Data Model помимо примитивных типов есть множество типов со вложенными значениями, для сериализации которых у Serializer есть соответствующие методы, подобно примитивным типам, однако их результатом является не конечное представление, как у примитивных типов, а специальный объект, реализующий поставленный в соответствии коллекции трейт. Например, объявление для сериализации массива будет выглядеть так, как на рис. 8.

```
pub trait Serializer: Sized {
    type Error: Error;
    type SerializeSeq: SerializeSeq<Ok = Self::Ok, Error = Self::Error>;

    fn serialize_seq(self, len: Option<usize>)
        -> Result<Self::SerializeSeq, Self::Error>;
}
```

Рисунок 8 — Объявление Serializer для последовательностей

Трейт SerializeSeq в свою очередь содержит объявления методов, специфичных для последовательностей, они представлены на рис. 9.

В рамках реализации формата Transit при сериализации массива эта про-

```rust
pub trait SerializeSeq {
    type Ok;
    type Error: Error;
    fn serialize_element<T: ?Sized>(&mut self, value: &T)
        -> Result<(), Self::Error>
    where
        T: Serialize;
    fn end(self) -> Result<Self::Ok, Self::Error>;
}
```

Рисунок 9 — Объявление SerializeSeq

цедура будет произведена и для его элементов, что допустимо ввиду наличия ограничения `T: Serialize`.

Схожим образом работает SerializeMap, объявленный на рис. 10.

```rust
pub trait SerializeMap {
    type Ok;
    type Error: Error;
    fn serialize_key<T: ?Sized>(&mut self, key: &T) -> Result<(), Self::Error>
    where
        T: Serialize;
    fn serialize_value<T: ?Sized>(&mut self, value: &T) -> Result<(), Self::Error>
    where
        T: Serialize;
    fn end(self) -> Result<Self::Ok, Self::Error>;

    fn serialize_entry<K: ?Sized, V: ?Sized>(
        &mut self,
        key: &K,
        value: &V,
    ) -> Result<(), Self::Error>
    where
        K: Serialize,
        V: Serialize;
}
```

Рисунок 10 — Объявление SerializeMap

Так как сериализуемая пара ключ-значение известна, применяется метод `serialize_entry`. Для формата Transit знание о том, что значение стоит в позиции ключа объекта, влияет на применяемый для него Serializer: `bool`, являясь базовым типом, будет представлен соответствующим логическим типом из

JSON, однако в позиции ключа он недопустим, поэтому будет применен реализованный KeySerializer, для которого возвращаемый тип `type Ok = String`, подходящий под ограничение JSON для ключей, и его метод `serialize_bool`, возвращающий `"~?t"` или `"~?f"`. Это специфическое представление для `bool` и при обратной операции будет интерпретировано соответствующе.

2.5 Сериализация типов-расширений

В отличие от базовых типов, множество типов-расширений не ограничено, однако их определение подчиняется нескольким правилам:

- Использование строк с префиксом-тегом для скалярных типов, например, `"~t2019-04-20T16:20:00Z"`
- Использование объекта с ключом-тегом для композитных типов, например, `{"~#set": [4, 2, 0]}`
- Сериализация/Десериализация происходит рекурсивно

Для композитных типов была введена оборачивающая структура Tagged, описанная на рис. 11. Так как `T: Serialize`, то содержимое будет сериализовано и обернуто в нужный тег.

2.5.1 Предподготовка типов-расширений

Автоматическая генерация реализаций Serialize для структур очень удобна и может дать реализацию, подходящую для большинства форматов данных, но в случае с Transit все типы, что не соответствуют базовым, требуется помечать. Для указания тега и преобразования к виду, удовлетворяющему Serialize, вводится трейт TransitToSerialize, описанный на рис. 12.

Приняв во внимание ограничения трейтов, визуально процесс можно представить так, как на рис. 13.

```rust
pub struct Tagged<T: Serialize> {
    pub tag: &'static str,
    pub value: T,
}

impl<T: Serialize> Serialize for Tagged<T> {
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
    where
        S: Serializer,
    {
        let mut map = serializer.serialize_map(Some(1))?;
        map.serialize_entry(self.tag, &self.value)?;
        map.end()
    }
}
```

Рисунок 11 — Объявление Tagged

```rust
pub trait TransitToSerialize {
    type Output: Serialize;
    fn to_serialize(&self) -> Self::Output;
}
```

Рисунок 12 — Объявление TransitToSerialize

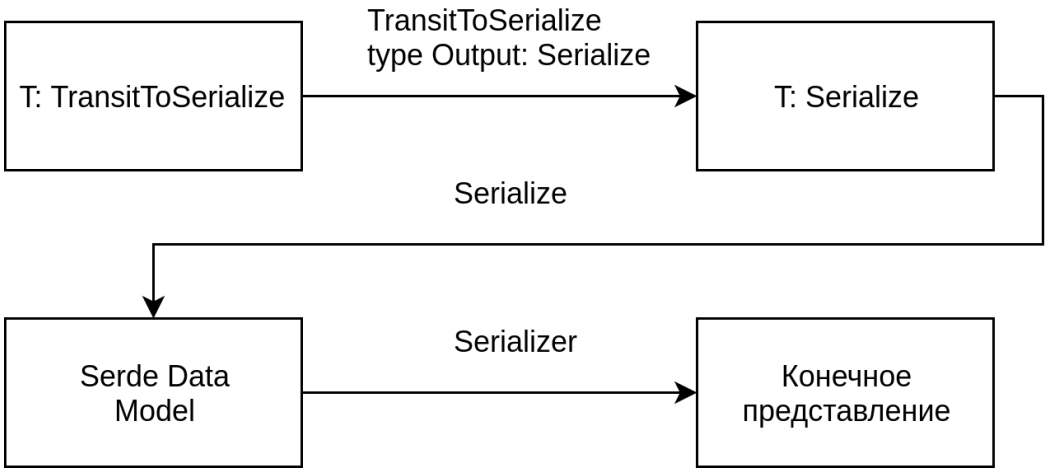

Рисунок 13 — Связь трейтов для сериализации типов-расширений

3 САМОСТОЯТЕЛЬНАЯ РЕАЛИЗАЦИЯ

Как следует из сказанного ранее, Serializer отвечает за конечное представление, в данном случае это JSON, а Serialize за представление типа в терминах Serde Data Model. Так как Serialize может быть реализован для типа только один раз, и эта реализация может быть желательной для всех форматов данных, с которыми работает приложение, кроме Transit, был введен трейт TransitToSerialize, над которым уже имеется полный контроль.

Однако для некоторых типов требуется больше информации, чем сериализованный вид вложенных значений, например, для стандартного для Transit типа CMap — Map с композитными ключами. Следовательно, для всех типов, которые могут являться ключами, требуется информация о том, являются ли они скалярными типами или композитными, чем трейт Serialize не располагает. Таким образом, для всех типов данных, которые должны быть сериализованы в формат Transit, должнен быть реализован отличный от Serialize трейт. Ввиду такой детали, реализация без использования фреймворка Serde, а только заимствование основных идей и парсера Serde JSON, окажется проще и удобнее, так как есть нужда обходить стандартные механизмы фреймворка.

3.1 JSON Verbose

Реализация формата Transit, работающая поверх JSON, позволяет работать в двух режимах: JSON и JSON Verbose. Последний удобен для разработки и отладки, так как более человекочитаем и не прибегает к механизмам, позволяющим уменьшить объем выходных данных. Аналогично Serde, разрабатываемая библиотека должна абстрагировать пользователя над «транспортными» деталями, поэтому следующие далее объявления применимы для всех реализаций. Однако, в первую очередь будет реализован сериализатор JSON Verbose и в дальнейшем адаптирован к JSON с кешированием.

3.1.1 Основные объявления для сериализации

Для всех типов, которые должны быть сериализованы в формат Transit, необходимо реализовать трейт TransitSerialize, объявленный на рис. 14.

```
pub trait TransitSerialize {
    const TF_TYPE: TransitType;
    fn transit_serialize<S: TransitSerializer>(&self, serializer: S)
        -> S::Output;
    fn transit_key<S: TransitSerializer>(&self, serializer: S)
        -> Option<S::Output>;
}
```

Рисунок 14 — Объявление TransitSerialize

TF_TYPE Ассоциированная константа, указывающая, к какой группе типов принадлежит тип, для которого реализуется трейт. Объявление представлено на рис. 15.

transit_serialize Аналогичен по смыслу методу `serialize` из фреймворка Serde: точка входа в процесс сериализации для типа. `TransitSerializer` объявлен на рис. 16 и будет рассмотрен позднее.

transit_key Так как на позиции ключа представление может быть иным, с помощью данного метода тип предоставляет свой сериализованный вид для этого случая. Возвращаемое значение опционально, поскольку не каждый тип может быть представлен ключом объекта: объект с композитными ключами приводится к типу CMap, упомянутому ранее.

```
#[derive(PartialEq)]
pub enum TransitType {
    Scalar,
    Composite,
}
```

Рисунок 15 — Объявление TransitType

```rust
pub trait TransitSerializer {
    type Output;
    type SerializeArray: SerializeArray<Output = Self::Output>;
    type SerializeMap: SerializeMap<Output = Self::Output>;

    fn serialize_null(self) -> Self::Output;
    fn serialize_string(self, v: &str) -> Self::Output;
    fn serialize_bool(self, v: bool) -> Self::Output;
    fn serialize_int(self, v: i64) -> Self::Output;
    fn serialize_float(self, v: f64) -> Self::Output;
    fn serialize_array(self, len: Option<usize>) -> Self::SerializeArray;
    fn serialize_map(self, len: Option<usize>) -> Self::SerializeMap;
}
```

Рисунок 16 — Объявление TransitSerializer

Поскольку Transit может работать поверх JSON, причем с модифицкацией Verbose, а также поверх MessagePack, требуется абстрагироваться от конечного представления, подобно Serde, поэтому был введен TransitSerializer. Так как нет необходимости соответсвовать Serde Data Model, методы TransitSerializer объявлены только для базовых типов Transit, возвращающие для скаляров `Output`. Для получения композитных базовых типов вводятся трейты SerializeArray и SerializeMap, объявленные на рис. 17. Типы, их реализующие, будут соответствовать выходному «транспортному» формату.

```rust
pub trait SerializeArray {
    type Output;
    fn serialize_item<T: TransitSerialize>(&mut self, v: T);
    fn end(self) -> Self::Output;
}
pub trait SerializeMap {
    type Output;
    fn serialize_pair<K, V>(&mut self, k: K, v: V)
    where
        K: TransitSerialize,
        V: TransitSerialize;
    fn end(self) -> Self::Output;
}
```

Рисунок 17 — Объявления SerializeMap и SerializeArray

Как и при реализации на Serde, на параметры типа накладывается ограничение `T: TransitSerialize`, поскольку сериализация рекурсивна.

3.1.2 Примеры реализаций

Для скалярного базового типа `bool` реализация трейта TransitSerialize описана на рис. 18, с различиями в обработке на разных позициях. Пример для композитного типа представлен на рис. 19.

```
impl TransitSerialize for bool {
    const TF_TYPE: TransitType = TransitType::Scalar;
    fn transit_serialize<S>(&self, serializer: S) -> S::Output
    where
        S: TransitSerializer,
    {
        serializer.serialize_bool(*self)
    }
    fn transit_key<S>(&self, serializer: S) -> Option<S::Output>
    where
        S: TransitSerializer,
    {
        let s = if *self {
            "~?t".to_owned()
        } else {
            "~?f".to_owned()
        };
        Some(serializer.serialize_string(&s))
    }
}
```

Рисунок 18 — Реализация TransitSerialize для bool

```rust
impl<K, V> TransitSerialize for BTreeMap<K, V>
where
    K: TransitSerialize,
    V: TransitSerialize,
{
    const TF_TYPE: TransitType = TransitType::Composite;

    fn transit_serialize<S>(&self, serializer: S) -> S::Output
    where
        S: TransitSerializer,
    {
        let mut ser_map = serializer.serialize_map(Some(self.len()));
        for (k, v) in self.iter() {
            ser_map.serialize_pair((*k).clone(), (*v).clone());
        }
        ser_map.end()
    }

    fn transit_key<S>(&self, _serializer: S) -> Option<S::Output>
    where
        S: TransitSerializer,
    {
        None
    }
}
```

Рисунок 19 — Реализация TransitSerialize для btree

Информация, указанная в константе `TF_TYPE`, используется при реализации SerializeMap. На данный момент это требуется только для одного стандартного типа — CMap, однако какой-либо пользовательский тип так же может опираться на эту информацию.

3.1.3 Основные объявления для десериализации

Как и для сериализации, текущая реализация десериализации опирается на статически известный ожидаемый тип и управляет ходом разбора полученных данных. Объявления трейтов представлены на рис. 20.

```rust
pub trait TransitDeserialize: Sized {
    const TF_TYPE: TransitType;

    fn transit_deserialize<D: TransitDeserializer>(
        deserializer: D,
        input: D::Input,
    ) -> TResult<Self>;

    fn transit_deserialize_key<D: TransitDeserializer>(
        deserializer: D,
        input: D::Input,
    ) -> TResult<Self>;
}

pub trait TransitDeserializer {
    type Input;
    type DeserializeArray: IntoIterator<Item = Self::Input>;
    type DeserializeMap: IntoIterator<Item = (Self::Input, Self::Input)>;

    fn deserialize_string(self, v: Self::Input) -> TResult<String>;
    fn deserialize_bool(self, v: Self::Input) -> TResult<bool>;
    fn deserialize_int(self, v: Self::Input) -> TResult<i64>;
    fn deserialize_float(self, v: Self::Input) -> TResult<f64>;
    fn deserialize_array(self, v: Self::Input) -> TResult<Self::DeserializeArray>;
    fn deserialize_map(self, v: Self::Input) -> TResult<Self::DeserializeMap>;
}
```

Рисунок 20 — Объявление TransitDeserialize и TransitDeserializer

Методы TransitDeserializer так же соответствуют базовым типам. Так как при попытке разбора, например, `bool` может возникнуть ошибка, если это числовое значение, все методы возвращают `TResult`, объявленный как `type TResult<T> = Result<T, Error>`, где `Error` — тип-сумма возможных ошибок при десериализации.

Продолжая аналогию с TransitSerializer, данный трейт реализуют типы, соответствующие «транспортным» форматам данных. Для типов внутри Serde JSON есть нужные операции по извлечению данных из `serde_json::Value`, например, `.as_i64(&self) -> Option<i64>`, реализующий весь функционал `deserialize_int(self, v: Self::Input) -> TResult<i64>` для Json Deserializer для целого числа вне позиции ключа.

Как методы TransitSerializer должны возвращать `Self::Output`, так каждый метод TransitDeserializer должен принимать `Self::Input`, для JSON Verbose в обоих случаях это будет `serde_json::Value`. Для десериализации последовательностей и словарей указано ограничение `IntoIterator`, где элементами выступает `Self::Input`, либо пара из них для словаря. Таким образом, любые входные данные, для которых реализован TransitDeserializer, можно пытаться десериализовать в структуру, для которой реализован TransitDeserialize и для ее вложенных значений, если это требование не удовлетворено, программа не будет скомпилирована.

Дизайн, опирающийся на единый входной тип, понятен интуитивно, будь это строка, другие бинарные данные или, как в данном случае, тип-сумма. Однако по-умолчанию итерация по JSON объекту средствами Serde JSON возвращает пару `(String, serde_json::Value)`, где String входным типом не является. Это было решено итератором-оберткой, объявленным на рис. 21.

```
struct JsonObjectIntoIter {
    js_iter: JsMapIntoIter,
}

impl Iterator for JsonObjectIntoIter {
    type Item = (JsVal, JsVal);

    fn next(&mut self) -> Option<Self::Item> {
        self.js_iter.next().map(|(k, v)| (JsVal::String(k), v))
    }
}
```

Рисунок 21 — Объявление JsonObjectIntoIter

3.1.4 Тест производительности

Ввиду того, что текущая реализация библиотеки для работы с форматом Transit для языка Rust является единственной, выполнить тест поизводительности на равных между реализациями условиях не выйдет. Однако с целью удостовериться, что алгоритмически текущая реализация не уступает эталонной, было

произведено сравнение с реализацией на языке Clojure, график хода которого представлен на рис. 22. Для осуществления тестов был использован фреймворк Criterion — порт одноименного фреймворка для языка Haskell. [12]

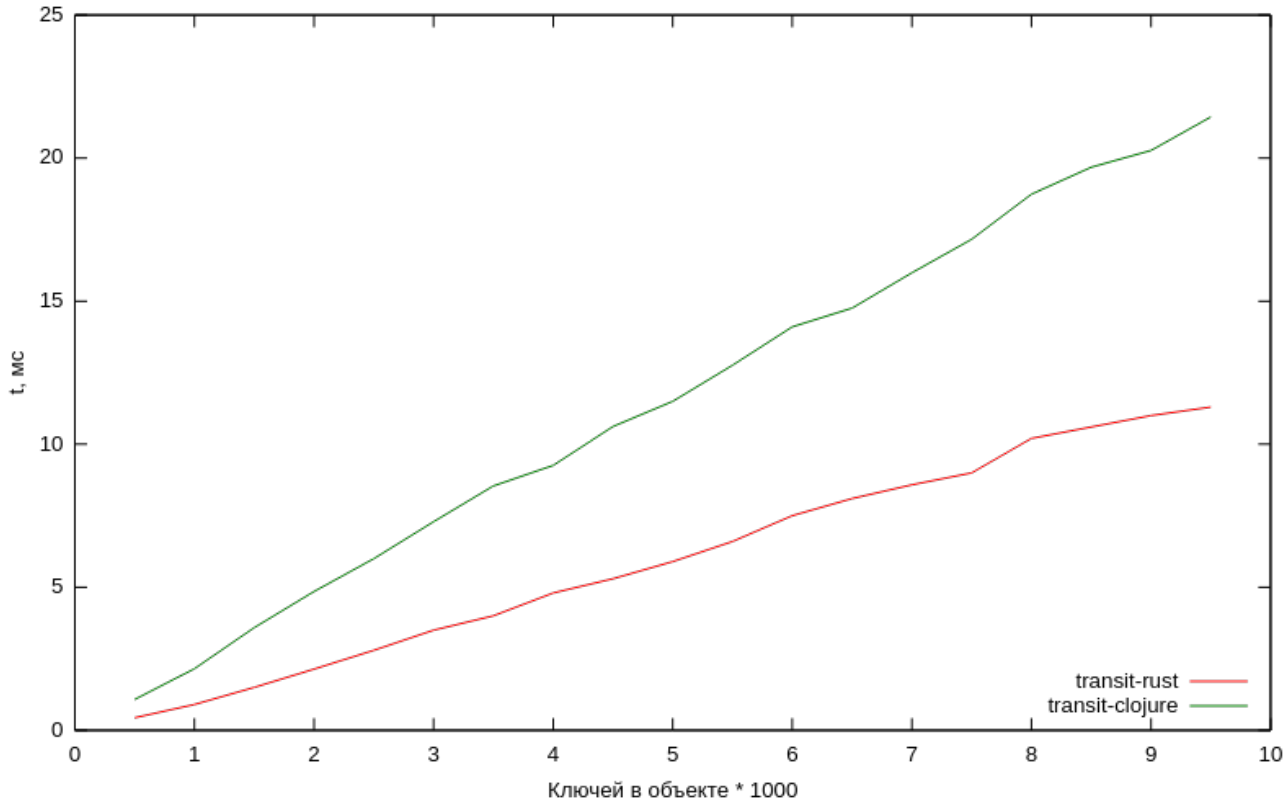

Рисунок 22 — Тест производительности сериализации Rust и Clojure

Для теста использовалась стуктура данных словарь, ключами которого выступали другие словари, а значением число. Количество ключей во внешнем словаре варьировалось согласно оси X, получаемый объект в формате Transit — CMap.

Как видно на графике, обе фукнции линейны с различным коэффициентом ввиду отличий сред выполнения. Таким образом, можно предположить, что разрабатываемая реализация не уступает референсной.

3.2 JSON non-verbose

Построенная база для JSON Verbose должна позволить добавлять варианты «транспорта», не затрагивая реализации TransitSerialize для конкретных типов данных.

3.2.1 Упразднение JSON Object

Первое отличие от JSON Verbose заключается в том, что объект JSON как базовый тип более не используется: для key-value структур применяется массив, первый элемент которого символ `^`. Схожее представление имеет тип CMap для композитных ключей, только первый элемент — `"~#cmap"`, а остальные обернуты во вложенный массив.

По результатам тестов производительности, график которых представлен на рис. 23, можно сказать, что использование массивов вместо объектов немного ускоряет процесс сериализации.

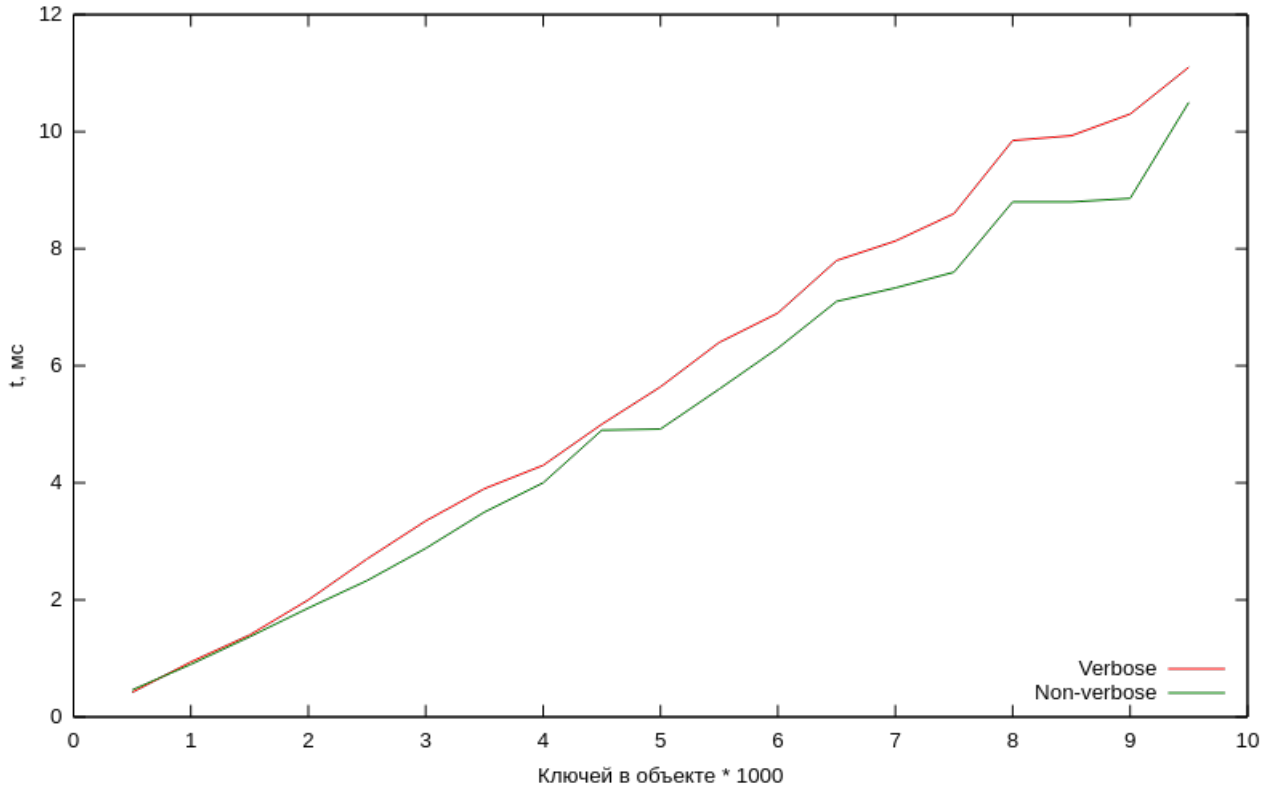

Рисунок 23 — Тест производительности сериализации Verbose и Non-verbose

Отказ от JSON Object ведет к неоднозначности: метод `serialize_map` использовался для записи key-value структур, в случае с JSON Verbose это был object, для non-verbose — массив с первым элементом `^`, однако значения с тегами, например, `{"~#set": [4, 2, 0]}`, которые ранее могли быть записаны как объект с одним ключом, теперь будут следовать за `^`, в то время как должны выглядеть как `["~#set", [4, 2, 0]]`. Таким образом, необходимо разделить запись объектов и тегированных значений, так как разных режимах работы одно не выражается через другое.

Формально тегированное значение представляет из себя следующее:

- Строковый тег, начинающийся с `~#`
- Значение композитного типа
- Базовый композитный тип (объект/массив), в который обернуты предыдущие два значения

Ввиду этого, требуется изменить интерфейс таким образом, чтобы ограничить допустимые значения под тегом только композитными типами, то есть предоставить `serialize_array` и `serialize_map`. Удовлетворяющие этим требованиям изменения трейта TransitSerializer представлены на рис. 24. Трейты SerializeTagArray и SerializeTagMap схожи с описанными ранее SerializeArray и SerializeMap соответственно, но будут использоваться для возврата тегированных значений.

```
pub trait TransitSerializer {
    /* ... */
    type SerializeTagArray: SerializeTagArray<Output = Self::Output>;
    type SerializeTagMap: SerializeTagMap<Output = Self::Output>;

    /* ... */

    fn serialize_tagged_array(self, tag: &str, len: Option<usize>)
        -> Self::SerializeTagArray;
    fn serialize_tagged_map(self, tag: &str, len: Option<usize>)
        -> Self::SerializeTagMap;
}
```

Рисунок 24 — Изменения TransitSerializer

Текущая реализация механизма сериализации в JSON представлена в виде UML диаграммы классов на рис. 25.

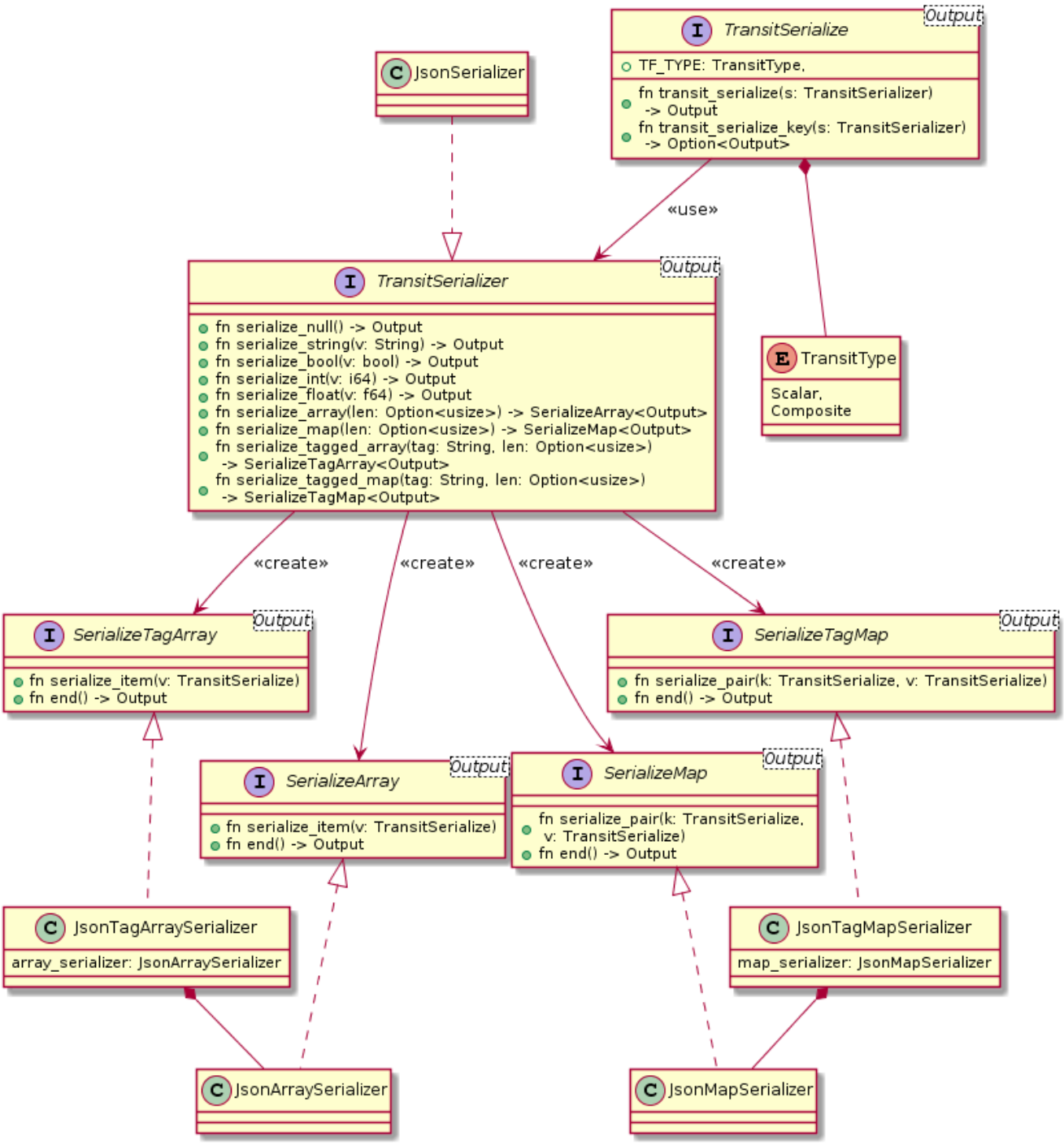

Рисунок 25 — UML диаграмма классов сериализации

3.2.2 Поддержка кеширования ключей

В режиме JSON non-verbose помимо отказа от использования объектов так же применяется кеширование ключей: то, что сериализуется как объект, то есть используется `serialize_map` и `serialize_tagged_*`, может быть сокращено в объеме за счет подстановки вместо повторяющихся ключей коротких

идентификаторов, если изначально ключ не менее четырех символов. На рис. 26 показаны примеры данных, к которым применено кеширование ключей.

```
[
  [
    "^",
    "name", "Evgeny",
    "full_ages", 24,
    "likes",
    [
      "~#set",
      ["rust", "transit"]
    ]
  ],
  [
    "^",
    "name", "Nadja",
    "full_ages", 23,
    "likes", ["~#set", ["python"]]
  ]
]
```

```
[
  [
    "^",
    "name", "Evgeny",
    "full_ages", 24,
    "likes",
    [
      "~#set",
      ["rust", "transit"]
    ]
  ],
  [
    "^",
    "^0", "Nadja",
    "^1", 23,
    "^2", ["^3", ["python"]]
  ]
]
```

Рисунок 26 — Примеры работы кеширования ключей

Кешированный ключ представляет из себя строку, состоящую из символа ^ и следующего за ним кеш-кода — одно- или двузначного числа в 44-ичной системе счисления (ASCII: 48 - 91). Для реализации данного механизма в процессе сериализации будет использоваться ассоциативный массив строк-ключей и кеш-кодов, таким образом, последующее обнаружение встреченной ранее строки будет заменено. При добавлении строки в ассоциативный массив требуется получить следующее новой значение кеш-кода — для решения этой задачи будет удобно применить итераторы и реализовать ленивую последовательность из кеш-кодов.

Так как «нумерация», полученная путем кеширования, является сквозной, то есть идет по мере обнаружения новых ключей в документе, объект, инкапсулирующий логику кеширования будет общий и передаваться по ссылке во все нисходящие `JsonSerializer` и ассоциированные с ним.

Реализация JSON non-verbose считается законченной, поэтому было вы-

полнено сравнение производительности с JSON verbose, результаты которого приведены на графике 27. Обе реализации имеют практически одинаковое время работы на массивах небольших объектов, однако разрыв в пользу кеширования увеличивается по мере роста числа и длины ключей.

Помимо скорости сериализации, кеширование влияет на размер выходных данных, что в свою очередь непосредственно сказывается на затрачиваемое на передачу время. График зависимости размера выходных данных от количества объектов представлен на рис. 28.

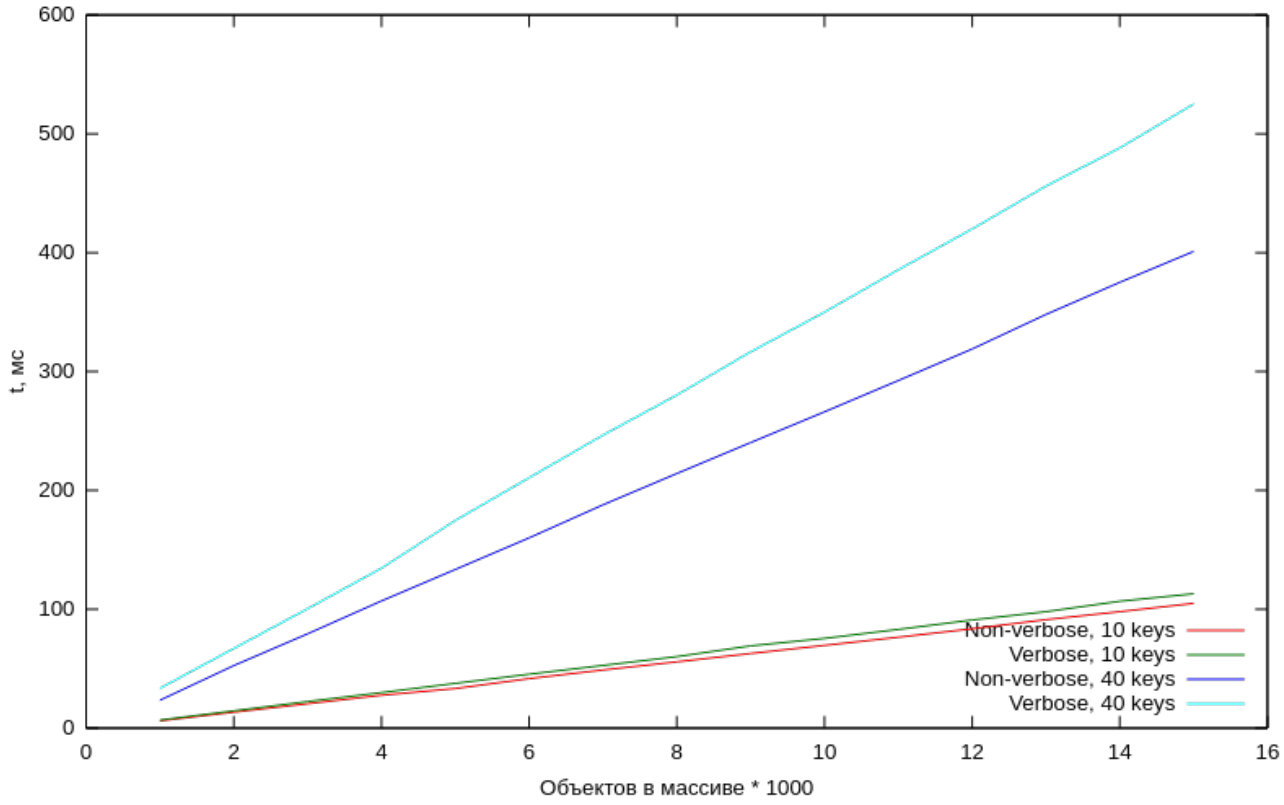

Рисунок 27 — Сравнение производительности при кешировании ключей

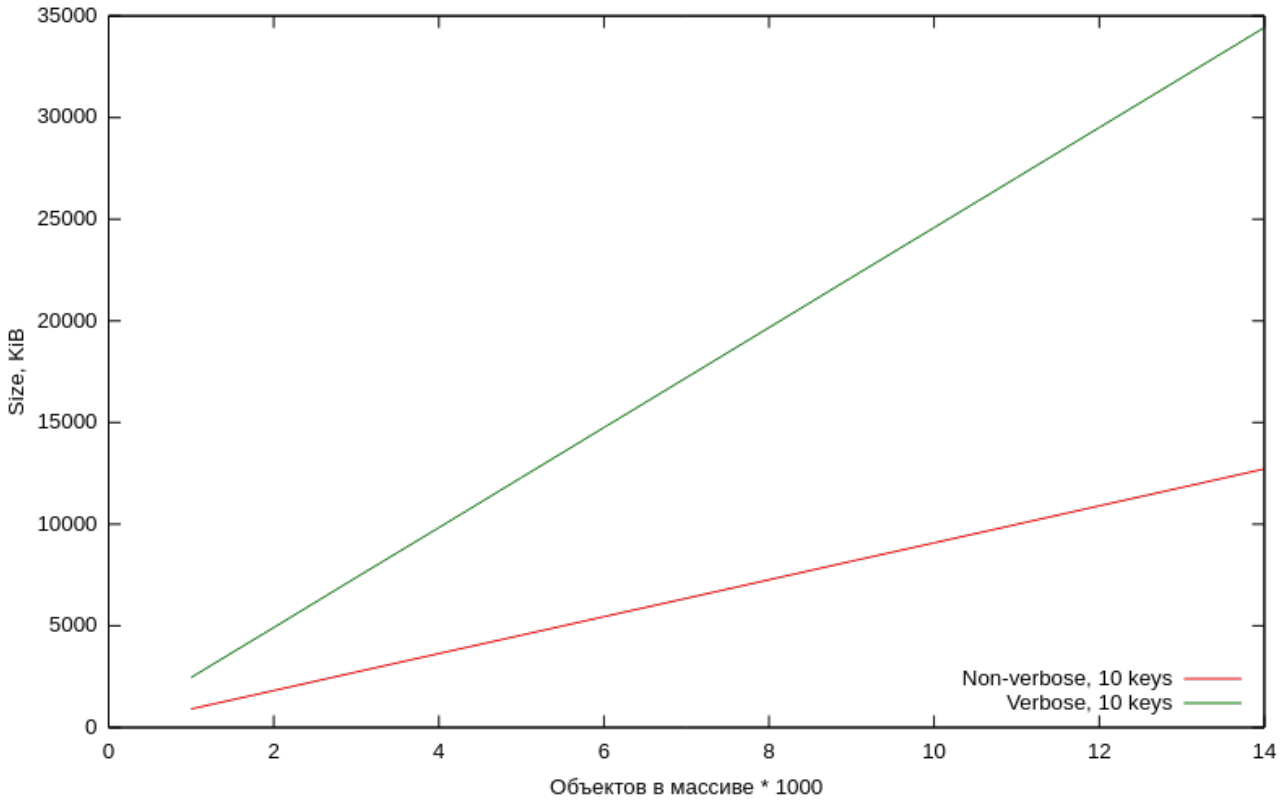

Рисунок 28 — Сравнение размера выходных данных при кешировании ключей

3.3 Поддержка стандартных типов данных

Помимо механизмов определения новых типов через базовые, формат Transit предоставляет стандартный набор типов, не входящие в какой-то конкретный домен и зачастую используемые как части другого, более сложного типа. Поэтому, используя наработки, следует поддержать стандартный набор типов, как того требует формат. Для использования стандартных реализаций этих типов был применен механизм условной компиляции, так как стандартный тип для Transit может не иметь стандартного аналога для языка и выбор реализации остается на усмотрения пользователя библиотеки, однако в случае отсутствия предпочтений путем указания флага можно подключить зависимость и использовать ее для работы с одним из таких типов.

3.3.1 Целые числа

В языке Rust целые числа представлены от `u8` до `u128` для беззнаковых и от `i8` до `i128` для чисел со знаком. Для сохранения возможности обмена информацией с помощью JSON между приложениями, размер типа Number ограничивается; таким образом, диапазон целых значений, помещаемый в Number, равен $-2^{53}+1 \ldots 2^{53}+1$. Числа, входящие в этот диапазон, сериализованы с помощью базового типа Number, остальные будут сериализованы как строки с тегом `~i` [14]. Кроме того, этот тег используется в позиции ключа. Для целых чисел произвольной точности, известных как BigInt, используется тег `~n`.

Реализация включает в себя проверку на вхождение в упомянутый ранее диапазон для всех примитивных типов целых чисел и с дальнейшим использованием строки или базового типа с применением методов `serialize_int` или `serialize_string`. BigInt сериализуется только в стоку, соответствующий тип взят из крейта num-bigint[1].

Функция десериализации для ожидаемого типа должна предпринять попытку извлечь информацию из JSON Number (`deserialize_int`) или же строки (`deserialize_string`). Следует учесть, что возможен исход, когда десериализация идет в числовой тип конкретного размера, в который полученные данные уместить нельзя.

3.3.2 Null

Null как таковой представлен в Rust через Option — тип-сумма, выражающий значение, которое может отсутствовать[2]. Так как в Transit явно не указано, какое значение может быть Null, положим, что любое значение может оказаться таковым. Так как в Rust это указано явно, то реализация сериализации для Option идет по двум ветвям:

[1] GitHub: rust-num/num-bigint

[2] Rust: std::option

1. При наличии значения, делегировать реализации для вложенного типа
2. При отсутствии значения, сериализовать базовым типом `null`. На позиции ключа — как строку `~_`.

Таким образом, если существует реализация сериализации для некоторого типа `T`, то автоматически будет определена сериализация для `Option<T>`.

3.3.3 Десятичные дроби

Аналогично целым числам, в формате Transit десятичные дроби могут быть представлены как JSON Number в качестве числа с плавающей точкой, в Rust для них два типа — `f32` и `f64`. Если это число NaN или Inf, то оно кодируется с тегом `~z`.

Для произвольной точности используется отдельный крейт Bigdecimal[3]. В отличие от целых чисел, тегированное представление используется только на позиции ключа с тегом `~d`.

3.3.4 Скаляры-расширения

Описанные выше типы являются базовыми и имеют представление в JSON, но так же обладают и строковым тегированным представлением, если уже не являются строкой. Типы, представленные в этом параграфе являются скалярными типами-расширениями и в формате Transit имеют только тегированное строковое представление. Их реализация идентична с отличием по символу тега, и таблица с типами и тегами представлена на рис. 29. Следует отметить типы Keyword и Symbol, являющиеся стандартными для формата Transit, но не распространенными в целом. Keyword — строка, семантика которой подразумевает ее использование в качестве ключа в объекте и тип специфичен для языка Clojure. Symbol — строка, которая может использоваться в качестве идентификатора (привязки) в языках семейства Lisp. Оба этих типа объявлены в разраба-

[3] GitHub: akubera/bigdecimal-rs

тываемой библиотеке и выражены строками, обернутыми в структуры с одним безымянным полем, так как распространенных аналогов в экосистеме языка Rust нет.

Name	Tag	Example
Keyword	:	`~:key`
Symbol	$	`~$sym`
Timestamp	t	`~t2019-04-17T13:37:00.33Z`
UUID	u	`~u73bc8129-f9e4-461e-b9de-c68d0028f8ec`
URI	r	`~rhttps://github.com/fominok`
Char	c	`~ci`
Arb. precision decimal	f	`~f13.37`
Arb. precision number	n	`~n420`
Special num	z	`~z-INF`

Рисунок 29 — Скалярные типы-расширения

3.3.5 Базовые типы-коллекции

Массив В зависимости от используемой реализации, JSON или JSON Verbose, данный базовый тип может соответсвовать разным коллекциям: в JSON он используется для сериализации любых коллекций, когда как в JSON Verbose — в качестве упорядоченной коллекции или вспомогательной, например, для Set, где порядок значения не имеет.

Если рассматривать массив JSON как самостоятельный тип, то в Rust ему будет соответствовать тип `&[T]` — slice типа `T`. Slice — ссылка на продолжительную часть какой-либо коллекции — это позволяет абстрагироваться над любыми последовательностями, например, тип `std::Vec<T>` или

массив `[T]` будут сериализованы в JSON Array независимо от конкретного типа.

Ассоциативный массив Для JSON Verbose в противоположность массиву JSON Object используется наиболее часто: для тегированных коллекций и для непосредственно ассоциативных массивов, когда в сокращенной реализации используются только массивы.

Семантически ассоциативный массив можно выразить через JSON Object или же JSON Array с первым элементом «^». В языке Rust в стандартной библиотеке реализованы 2 типа: `BTreeMap` и `HashMap`, каждая из которых, используя `serialize_map`, будет сериализована в зависящее от выбранной реализации представление ассоциативного массива.

3.3.6 Множество

Композитный тип-расширение, сериализуемый как тегированный массив с тегом `~#set`. В языке Rust стандартная неупорядоченная коллекция с уникальными элементами это `HashSet`.

3.3.7 Ассоциативный массив с композитными ключами

Так как в формате Transit нет ограничения на ключи, ими могут являться в том числе и композитные типы данных. Для такого случая существует стандартный тип, на который ранее в работе ссылались как на CMap. Для его работы каждый тип, который можно сериализовать, то есть все те, для которых реализован трейт `TransitSerialize`, должен декларировать, является ли он композитным или нет. При вызове `serialize_map` происходит проверка, являются ли все ключи скалярными типами и приводятся к строкам, иначе такой ассоциативный массив сериализуется как тегированный массив, где ключи и значения чередуются.

3.3.8 Ссылка

Тегированный ассоциативный массив, с полями «href», «rel», «name», «render», «prompt». Все поля за исключением первого являются опциональными и строковыми, поле «href» содержит URI и будет сериализовано с тегом `~r`, как было упомянуто на рис. 29.

3.4 Автоматическая реализация TransitSerialize для пользовательских структур

Так как расширяемость является одной из главных особенностей формата Transit, для улучшения пользовательского опыта библиотеки требуется сократить количество требуемого шаблонного кода для добавления новых типов данных.

3.4.1 Структура с именованными полями

Для примера можно обратиться к рассмотренным ранее данным, представленным на рисунке 1. Структура, которая соответствует этому объекту в языке Rust с учетом устраненных неточностей описана на рисунке 30.

Известно, что:

1. Сериализация происходит рекурсивно
2. Существуют стандартные реализации для сериализации строк, чисел, множеств и словарей
3. Структура с именованными полями может быть сконструирована словарем

Из перечисленного следует, что для рассматриваемой структуры возможно реализовать TransitSerialize. Более того, пункт 3 можно и нужно обобщить на любые подобные структуры, так как по принципу реализации идентичны за исключением набора строковых ключей.

Объявление структуры является частью исходного кода, и для работы с ним требуется прибегать к средствам метапрограммирования, а именно — мак-

росам. В языке Rust поддерживаются 2 типа макросов: декларативные и процедурные. Макросы для работы с синтаксическим деревом относят к последним, к ним же и относят Derive-макросы, ориентированные на автоматическую реализацию трейтов, что есть данный случай. Реализация макроса представлена в приложении А. На этапе компиляции известен набор полей структуры и имя типа; генерируется реализация, которая вызывает `serialize_tagged_map` с именем типа в качестве тега и в ходе разворачивания макроса подставляет число вызовов `serialize_pair` равное количеству полей структуры.

3.4.2 Структура с позиционными полями

Альтернативой использованию полей с именами являются структуры с безымянными полями, идентифицируемые их позицией, иначе говоря — именованный кортеж. Например, точка на плоскости может быть выражена структурой `struct Point(i32, i32)`. Реализация TransitSerialize для такой структуры может быть выполнена схожим образом, но через `serialize_tagged_array`, так как это упорядоченная коллекция, нежели словарь. Аналогично, для каждого из полей будет вызван метод `serialize_item`, осуществляющий сериализацию вглубь.

3.4.3 Перечисление

Перечисления позволяют определить тип и его возможные значения и являются вторым способом для определения пользовальских типов. Наиболее часто используемые перечисления — `Option<T>` и `Result<T, E>`. В то время как для формата Transit null-семантика определена через Option, для пользовательских перечислений назначение будет отличаться и требовать ручной реализации. Однако, можно предоставить автоматическую реализацию, предположив, что каждый вариант перечисления будет наделен собственным тегом.

Например, в приложении по сети между компонентами происходит обмен

сообщениями, выражающих какие-то произошедшие в системе события. Так как множество событий ограничено и сопровождающая их информация может обладать разной структурой, это удобно смоделировать типом-перечислением с отличными наборами полей для каждого из вариантов. В слабоструктурированном представлении это могло быть в виде поля, обозначающего тип события, и нефиксированного набора всех остальных полей, зависящего от типа сообщения, и без проверок на этапе компиляции на обработку всех случаев. Пример объявления такого перечисления и возможного представления варианта в формате Transit представлен на рисунке 31.

```
enum Event {
    TemperatureChanged {
        room_name: String,
        temperature: i32,
    },
    MotionDetected {
        room_name: String
    },
    GoneOnline(Uuid),
    GoneOffline(Uuid),
}
```

```
{
    "~#temperaturechanged": {
        "room_name": "kitchen",
        "temperature": 32
    }
}

/* ... */
{
    "~#goneoffline": ["92d112b0..."]
}
```

Рисунок 31 — Пример результата сериализации варианта перечисления

Как видно на примере, вариант перечисления может обладать как именованными полями, так и позиционными, по аналогии со структурами. Автоматическая реализация должна поддержать оба варианта.

Общее для перечислений В зависимости от значения под типом-перечислением при сериализации должен получиться разный тег и коллекция под ним. Для ветвления по вариантам перечисления используется выражение `match` с количеством ветвей эквивалентным количеству вариантов. Для каждого из возможных значений генерируется код различный для именованных и позиционных полей варианта. Сгенерированные ветви помещаются в тело `match` через разделитель.

Именованные поля Реализация для именованных полей выполнена аналогично структуре с этим типом полей с тем отличием, что требуется получить ветвь для выражения `match`. Таким образом, слева от `=>` требуется выполнить привязки по именам полей, а в правой части получить по ним значения для вложенной сериализации.

Позиционные поля Для данного типа полей отличий больше, так как имена идентификаторов отсутствуют. В случае со структурой доступ к значениям можно было осуществить по численному литералу, когда как в ветви `match` требуется выполнить привязку для каждого поля — необходимо предварительно сгенерировать идентификаторы.

```
use chrono::{DateTime, TimeZone, Utc};
#[derive(Clone, TransitSerialize)]
struct User {
    name: String,
    related: BTreeSet<String>,
    registered: DateTime<Utc>,
    skills_by_rates: BTreeMap<i32, BTreeSet<String>>,
}

let mut rel = BTreeSet::new();
rel.insert("Billy".to_owned());
rel.insert("Mark".to_owned());
rel.insert("Steve".to_owned());

let mut skills = BTreeMap::new();
let mut hs1 = BTreeSet::new();
hs1.insert("Linux".to_owned());
hs1.insert("Git".to_owned());
skills.insert(3, hs1);
let mut hs2 = BTreeSet::new();
hs2.insert("Performance artist".to_owned());
skills.insert(2, hs2);
let mut hs3 = BTreeSet::new();
hs3.insert("Rust".to_owned());
skills.insert(1, hs3);

let u = User {
    name: "Van".to_owned(),
    related: rel,
    registered: Utc.ymd(1995, 10, 11).and_hms(0, 0, 0),
    skills_by_rates: skills,
};
```

Рисунок 30 — Объявление Person с примером заполнения

ЗАКЛЮЧЕНИЕ

Была разработана библиотека на языке Rust, позволяющая приложениям, написанным на этом языке, проще интегрироваться в системы, где для обмена данными используется формат Transit. Представленное программное решение разделено на модули, и пользователь может выбрать нужную реализацию, импортируя соответствующий модуль.

Одной из особенностей формата Transit является то, что работа с ним не требует обязательного наличия библиотек кроме тех, что используются для работы с JSON. Однако ввиду основных рассмотренных в работе причин использования этого формата, а так же мощной системы типов языка Rust, использование специализированной библиотеки помимо удобств дает возможность предотвратить множество ошибок на этапе компиляции, что было идеей, лежащей в основе принимаемых архитектурных решений.

Предполагается, что созданное программное решение упростит разработчикам систем, где используется формат Transit, создание и интеграцию производительных сервисов на языке Rust, снижая временные затраты на удовлетворение внутренних соглашений и отладку.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Kolodin D. Hands-On Microservices with Rust. Packt Publishing, 2019. 520p.
2. Stack Overflow's annual Developer Survey. Available at: https://insights.stackoverflow.com/survey/2019#most-loved-dreaded-and-wanted (accessed 10 May 2019).
3. Python object serialization: pickle. Available at: https://docs.python.org/3/library/pickle.html (accessed 10 May 2019).
4. Buneman P. Semistructured Data. University of Pennsylvania, 1997.
5. Chromium source code. Available at: https://chromium.googlesource.com/chromium/src/base/+/master/json/json_reader.h (accessed 10 May 2019).
6. Amazon Ion. Available at: http://amzn.github.io/ion-docs/ (accessed 10 May 2019).
7. Rich Hickey: Transit. Available at: http://blog.cognitect.com/blog/2014/7/22/transit (accessed 10 May 2019).
8. Transit format specification. Available at: https://github.com/cognitect/transit-format (accessed 10 May 2019).
9. Serde docs. Available at: https://serde.rs (accessed 10 May 2019).
10. Klabnik S., Nichols C.: The Rust Programming Language. No Starch press, 2018. 488p.
11. Serde docs: SDM. Available at: https://serde.rs/data-model.html (accessed 10 May 2019).

12. O’Sullivan, Bryan. (2009). Criterion, A New Benchmarking Library for Haskell.

13. McCarthy, John. (1960). Recursive Functions of Symbolic Expressions and Their Computation by Machine.

14. RFC 8259 The JavaScript Object Notation (JSON) Data Interchange Format, ISSN: 2070-1721, December 2017

ПРИЛОЖЕНИЕ А

```rust
extern crate proc_macro;

use proc_macro2::{Span, Ident, TokenStream};
use quote::quote;
use syn;

#[proc_macro_derive(TransitSerialize)]
pub fn transit_macro_derive(
    input: proc_macro::TokenStream
) -> proc_macro::TokenStream {
    let ast = syn::parse(input).unwrap();
    impl_transit_macro(&ast)
}

fn put_serialize_struct_body(name: &Ident, body: TokenStream) -> TokenStream {
    quote! {
        impl TransitSerialize for #name {
            const TF_TYPE: TransitType = TransitType::Composite;
            fn transit_serialize<S: TransitSerializer>(&self, serializer: S)
                -> S::Output {
                #body
            }
            fn transit_serialize_key<S: TransitSerializer>(&self, serializer: S)
                -> Option<S::Output> {
                None
            }
        }
    }
}

fn named_body(tag: String, fields: &syn::FieldsNamed) -> TokenStream {
    let fields_named = fields
        .named
        .iter()
        .map(|x| (&x.ident).clone().expect("Requires identifier"));
    let fields_str = fields_named.clone().map(|x| x.to_string());
    let len = fields_named.len();

    quote! {
        let mut ser_map = serializer
            .clone()
            .serialize_tagged_map(#tag, Some(#len));
        #(ser_map.serialize_pair(#fields_str, self.#fields_named.clone());)*
        ser_map.end()
    }
}
```

```rust

fn unnamed_body(tag: String, fields: &syn::FieldsUnnamed) -> TokenStream {
    let len = fields.unnamed.len();
    let accessors = 0..len;
    quote! {
        let mut ser_arr = serializer
            .clone()
            .serialize_tagged_array(#tag, Some(#len));
        #(ser_arr.serialize_item(self.#accessors);)*
        ser_arr.end()
    }
}

fn process_struct_named(
    name: &Ident,
    tag: String,
    fields: &syn::FieldsNamed
) -> TokenStream {
    let body = named_body(tag, fields);
    put_serialize_struct_body(name, body)
}

fn process_struct_unnamed(
    name: &Ident,
    tag: String,
    fields: &syn::FieldsUnnamed
) -> TokenStream {
    let body = unnamed_body(tag, fields);
    put_serialize_struct_body(name, body)
}

fn process_enum(name: &Ident, variants: &[TokenStream]) -> TokenStream {
    quote! {
        impl TransitSerialize for #name {
            const TF_TYPE: TransitType = TransitType::Composite;

            fn transit_serialize<S: TransitSerializer>(
                &self,
                serializer: S
            ) -> S::Output {
                match self {
                    #(#variants),*
                }
            }

            fn transit_serialize_key<S: TransitSerializer>(
                &self,
                serializer: S,
            ) -> Option<S::Output> {
                None
```

```
            }
        }
    }
}

fn process_enum_variants(
    name: &Ident,
    variants: &syn::punctuated::Punctuated<syn::Variant, syn::token::Comma>,
) -> Vec<TokenStream> {
    let mut quoted_variants: Vec<TokenStream> = vec![];
    for v in variants.iter() {
        let tag = format!("~#{}", v.ident).to_lowercase();
        match &v.fields {
            syn::Fields::Named(fields) => {
                let fields_named = fields
                    .named
                    .iter()
                    .map(|x| (&x.ident).clone().expect("Requires identifier"));
                let fields_named2 = fields_named.clone();

                let fields_str = fields_named.clone().map(|x| x.to_string());
                let len = fields_named.len();

                let body = quote! {
                    let mut ser_map = serializer
                        .clone()
                        .serialize_tagged_map(#tag, Some(#len));
                    #(ser_map.serialize_pair(#fields_str, #fields_named.clone());)*
                    ser_map.end()
                };
                let vident = v.clone().ident;
                let arm = quote! {
                    #name::#vident {#(#fields_named2),*} => {#body}
                };
                quoted_variants.push(arm);
            }
            syn::Fields::Unnamed(fields) => {
                let len = fields.unnamed.len();
                let accessors = (0..len).map(|x| {
                    let ident = format!("syn{}", x);
                    syn::Ident::new(&ident, Span::call_site())
                });
                let accessors2 = accessors.clone();

                let body = quote! {
                    let mut ser_arr = serializer
                        .clone()
                        .serialize_tagged_array(#tag, Some(#len));
                    #(ser_arr.serialize_item(#accessors.clone());)*
                    ser_arr.end()
```

```
                };
                let vident = v.clone().ident;
                let arm = quote! {
                    #name::#vident (#(#accessors2),*) => {#body}
                };
                quoted_variants.push(arm);
            }
            _ => unimplemented!(),
        }
    }
    quoted_variants
}

fn impl_transit_macro(ast: &syn::DeriveInput) -> proc_macro::TokenStream {
    let name: &Ident = &ast.ident;
    let tag = format!("~#{}", name).to_lowercase();
    let gen = match &ast.data {
        syn::Data::Struct(ds) => match &ds.fields {
            syn::Fields::Named(fields) => process_struct_named(name, tag, fields),
            syn::Fields::Unnamed(fields) => process_struct_unnamed(name, tag, fields),
            _ => unimplemented!(),
        },
        syn::Data::Enum(de) => {
            let processed_variants = process_enum_variants(name, &de.variants);
            process_enum(name, &processed_variants)
        }
        _ => unimplemented!(),
    };
    gen.into()
}
```

Рисунок 32 — Исходный код макроса для реализации TransitSerialize