Содержание

1 Введение **2**
1.1 Актуальность и значимость . . . 2
1.2 Краткое описание работы . . . 3
1.3 Постановка цели . . . 4
1.4 Поставленные задачи . . . 4
1.5 Структура работы . . . 5

2 Обзор **5**
2.1 Что такое биндинги . . . 5
2.2 Сравнение с аналогами . . . 6
2.2.1 CythonWrapper [2] . . . 6
2.2.2 Binder [4] . . . 6

3 Архитектура программы **7**
3.1 Получение информации о необходимых классах и функциях из AST . . . 7
3.2 Кодогенерация биндингов . . . 9
3.3 Сборка сгенерированного исполняемого файла . . . 12

4 Программная часть **12**
4.1 Описание компонентов проекта . . . 12
4.2 Тестирование . . . 15
4.3 Поддержка других систем сборки . . . 15

5 Заключение **16**
5.1 Дальнейшее развитие . . . 16

Аннотация

В данном проекте описывается и реализуется фреймворк для автоматической генерации обёртки для классов и функций, написанных на языке C++, позволяющей работать с ними на языке Python. Иными словами, новый продукт позволит с минимальными затратами перенести интерфейс взаимодействия с библиотекой, а так же позволит сохранить эффективность работы исполняемого кода.

Ключевые слова

Генерация кода, биндинги, LibTooling, C++, Python

1 Введение

В современном мире существует ряд библиотек, для которых необходимо предоставлять один и тот же пользовательский интерфейс для разных языков программирования. Примером таких библиотек могут служить API для преобразования больших данных или интерфейс взаимодействия с базами данных.

Кроме того, в некоторых случаях необходимо сохранить высокую скорость работы программы вне зависимости от того, на каком языке разработчик использует интерфейс библиотеки. Ярким примером может служить библиотека NumPy. Несмотря на то, что она используется в программах на языке Python, для которого характерна низкая скорость работы за счёт особенностей архитектуры, эта библиотека быстро обрабатывает большие массивы данных так, как если бы программа была написана на C++. Секрет кроется в том, что на самом деле внутри себя NumPy представляет собой код на более низкоуровневом языке – C. За счёт этого и достигается скорость работы программ.

Можно заметить, что для решения обеих задач было бы удобно и эффективно использовать некоторую специальную программу, которая автоматически транслирует код с одного языка программирования на другой. В данной работе реализовывается утилита, позволяющая с минимальными затратами для разработчика перенести интерфейс взаимодействия с библиотекой с C++ на Python.

1.1 Актуальность и значимость

Чтобы предоставить пользователю один и тот же интерфейс взаимодействия с библиотекой на нескольких языках программирования можно полностью продублировать код на каждом из них. Но такое решение, во-первых, будет крайне неэффективным для больших проектов – придётся заново писать сотни тысяч строк кода, а, во-вторых, усложнит поддержку этого нового кода – при малейшем измнении какой-либо логики придётся менять код сразу в нескольких местах.

Другим решением данной проблемы могут быть **биндинги** – некоторая обёртка, позволяющая пользоваться объектами, написанными на одном языке программирования, с других языков (в следующих пунктах будет подробнее описан принцип работы). Хотя это решение и менее затратно, чем предыдущее, у него всё равно есть некоторые нюансы. Проблема разработчиков, желающих под-

держать возможность пользоваться интерфейсом библиотеки на других языках, заключается в необходимости дополнительно изучать новые инструменты, настраивать и описывать этот интерфейс вручную. В больших проектах это также может быть существенной пробемой, замедляющей разработку и требующей дальнейшей поддержки дополнительного кода.

Более того, почти все биндинги поверх существующего кода представляют собой шаблонное описание, которое можно автоматизировать на основе статического анализа кода.

Именно поэтому инструмент, автоматизирующий трансляцию интерфейса C++-библиотек в Python, мог бы существенно ускорить разработку библиотек и упростить их поддержку.

1.2 Краткое описание работы

В данной работе реализовывается фреймворк, позволяющий автоматизировать создание биндингов, что позволит за несколько тривиальных действий перенести интерфейс взаимодействия какой-либо библиотеки, написанной на C++, в Python. Эти языки были выбраны как наиболее распространённые для определённых требований: C++ – для скорости обработки и Python – как наиболее простой и удобный для большинства разработчиков. Программа поделена на 2 логические части:

1. Генерация нового кода на основе классов и функций на C++
2. Сборка сгенерированного кода и создание Python-модуля.

Генерация нового кода также поделена на 2 отдельные компоненты:

1. Анализ абстрактного синтаксического дерева библиотеки на C++, фильтрация и получение необходимых данных для дальнейшей генерации
2. Генерация биндингов на основе извлечённых раннее данных

1.3 Постановка цели

Пусть у пользователя есть некоторая библиотека, написанная на C++. Он хочет предоставить возможность пользоваться этой библиотекой на Python, при этом он не хочет тратить много времени на перенос интерфейса вручную, и/или ему необходимо сохранить ту же эффективность исполнения программы, которая есть в C++. **Цель** — реализация фреймворка для автоматической генерации биндингов в соответствии со следующими критериями:

- **Входные данные пользователя:** список папок, в которых располагаются .h-файлы (*хэдеры* – по своей сути файлы, описывающие интерфейс классов и функций), из которых необходимо извлечь требуемый интерфейс, название входного модуля.
- **Выходные данные:** сгенерированный .so-файл, позволяющий импортировать новую библиотеку в Python.

Взаимодействие пользователя с приложением должно выглядеть так:

1. Пользователь импортирует код фреймворка в свой проект
2. Указывает в CMake-файле программы свою библиотеку на C++ как зависимость (реализуется как указание определённой переменной окружения)
3. Запускает сборку программы и получает на выход .so-файл с библиотекой
4. Импортирует готовый модуль в свой код на python

1.4 Поставленные задачи

1. Изучить варианты для анализа синтаксического дерева кода на C++
2. Изучить библиотеки для создания биндингов
3. Изучить и сравнить различные средства для генерации кода
4. Реализовать все логические части части фреймворка, описанные раннее
5. Протестировать написанное приложение с разными сценариями использования и разными объектами в C++ коде

1.5 Структура работы

В главе 2 проводится обзор предметной области - приводится определение биндингов, подробно описиываются их характеристики, а также проводится обзор существующих решений задачи автоматической генерации биндингов.

В главе 3 описаны архитектурные решения для различных частей фреймворка и их обоснование.

В главе 4 содержатся детали реализации программной составляющей.

В главе 5 подводятся итоги и описываются возможные направления дальнейшего развития проекта.

2 Обзор

2.1 Что такое биндинги

В программировании термин **"биндинги"** (или **"обвязки"**) относится к механизмам, которые позволяют использовать функции, классы или другие компоненты, написанные на одном языке программирования, в коде, написанном на другом языке. В частности для python-биндингов, это по сути переводчики между Python и C++. Они позволяют Python вызывать функции из скомпилированных C++-библиотек. Это достигается с помощью специальных оберток, которые "обвязывают" код и делают его доступным для Python.

Рассмотрим некоторые важные характеристики биндингов.

1. **Объекты Python:** Python имеет свою собственную систему объектов и типов данных. Python-биндинги преобразуют объекты из других языков в объекты Python, чтобы пользователь мог работать с ними так же, как с любыми другими объектами Python.

2. **Преобразование типов данных:** поскольку Python и другие языки могут использовать разные типы данных в одном и том же сценарии, биндинги выполняют преобразование между ними. Например, если функция на C++ возвращает std::vector, то биндинги автоматически преобразуют тип данных так, чтобы функция смогла правильно работать, в данном случае это будет преобразование из std::vector в list из Python.

3. **Управление памятью:** другие языки, такие как C или C++, обычно управляют памятью вручную, в то время как Python использует автоматическое управление памятью. Python-биндинги должны учитывать это различие и обеспечивать корректное управление памятью для объектов, созданных на стороне другого языка.

4. **Интеграция с интерпретатором:** Python-биндинги работают с интерпретатором Python, чтобы он мог исполнять код из C++. В частности, это включает в себя создание интерфейсов для вызова интерпретатором функций и методов из C++, а также передачу данных между Python и объектами в C++.

5. **Компиляция исходного кода:** В большинстве случаев, библиотеки на C или C++ нужно скомпилировать в бинарный код перед использованием. Аналогично биндинги поверх C++-библиотек необходимо скомпилировать в .so-библиотеку для дальнейшего импорта в python-программу. Python-биндинги обычно включают в себя инструменты для компиляции исходного кода в библиотеку, которая может быть загружена и использована в программах.

2.2 Сравнение с аналогами

2.2.1 CythonWrapper [2]

Это фреймворк, который генерирует Cython-биндинги для C++ кода. Во-первых, анализ AST-дерева в данном решении реализуется через Python API, что может потенциально быть медленнее, чем если делать это на C++. Во-вторых, в качестве библиотеки для биндингов в данном решении отсутствует поддержка некоторых объектов из C++, например умных указателей, что может привести к невозможности автоматически создать модуль без дополнительных действий, в отличие от нового решения, где это автоматически поддержано.

2.2.2 Binder [4]

Это более схожий с данным проектом аналог. Главный недостаток данного проекта – долгая и сложная генерация python-модуля на основе C++-библиотеки. Согласно официальной документации, пользователю нужно сделать целых 5

шагов, чтобы получить искомую .so-библиотеку. Хотя это позволяет гибко настраивать то, какую именно в итоге пользователь хочет видеть библиотеку, это сильно переусложняет генерацию, и для некоторых случаев, возможно, быстрее и проще будет написать биндинги вручную.

3 Архитектура программы

Пайплайн автоматической генерации модуля на python предполагается линейным. Сначала программа получает информацию об объектах на C++, строит абстрактное синтаксическое дерево, извлекает из него только необходимую информацию. Затем из полученных данных программа генерирует новый исполняемый файл с описанными биндингами. После этого она исполняет этот файл и создаёт динамическую .so-библиотеку, которую далее пользователь прилинковывает к своему проекту.

На схеме 1 логические части отделены пунктиром. Также в левой части указаны решения, с помощью которых будет реализовываться тот или иной функционал.

Теперь рассмотрим отдельные части подробнее.

3.1 Получение информации о необходимых классах и функциях из AST

Рассмотрим существующие решения для данной проблемы:

1. **LLVM LibTooling** [3]. Это библиотека, позволяющая писать произвольные инструменты для кода, связанные с компилятором Clang. В частности, она позволяет исследовать абстрактное синтаксическое дерево (AST). В контексте данного приложения также важно получать некоторый контекст из анализируемых объектов в коде. К примеру, приватный ли метод или публичный, какая сигнатура у конструктора и так далее. Все эти потребности LibTooling покрывает.

   Кроме того, это проект от создателей одного из самых известных и популярных компиляторов для C++ — Clang. Это позволяет убедиться в надёжности данного решения.

Рис. 1: Графическая иллюстрация пайплайна обработки данных

2. **CppAst** [5]. Это единственное решение, которое удалось найти в открытом доступе помимо LibTooling. Автор утверждает, что эта библиотека имеет расширенные возможности по сравнению с аналогом и позволяет более детально изучать AST. Однако у CppAst нет документации и этот проект не так известен, как его аналог.

Для анализа AST был выбран первый вариант. Это обусловлено тем, что у данного решения есть большое сообщество и множество примеров использования, а также наличие подробной документации.

Как происходит обработка AST?

Высокоуровнево рассмотрим, как происходит обработка на данном этапе. LLVM LibTooling предоставляет интерфейс обработки необходимых объектов в абстрактном синтаксическом дереве в виде абстрактного класса $MatchFinder :: MatchCallback$, который нужно реализовать пользователю в зависимости от его целей. В этом классе есть определённый метод, который вызывается при запуске обхода каждый раз, когда в AST встречается подходящий узел (это тоже настраивается пользователем). Для данного проекта необходимо обрабатывать только 3 сущности:

1. функции
2. классы и структуры
3. поля и методы определённых классов (в том числе конструкторы)

Принимая каждый из этих объектов нужно сначала понять, действительно ли он нужен в публичном интерфейсе (или это приватный объект, который нужно просто проигнорировать). Затем, если этот объект действительно должен появиться в публичной интерфейсе новой библиотеки, его нужно передать в следующий этап - генерация кода.

Программная часть данного этапа будет подробнее описана в следующей главе.

3.2 Кодогенерация биндингов

После анализа абстрактного синтаксического дерева и получения необходимых для пользовательского интерфейса объектов необходимо сгенерировать описание биндингов для них.

Расмотрим некоторые существующие в открытом доступе решения для биндингов:

1. **Pybind11** представляет собой легковесную header-only (это означает, что её не нужно предварительно собирать) библиотеку для биндингов. Из её особенностей важно выделить простоту описания биндингов в декларативном подходе, а также поддержку множества нововведений из относительно свежих версий стандарта C++, например, нативную поддержку умных указателей.

2. **Swig** (*simplified wrapper and interface generator*) также в частности представляет собой инструмент для связывания библиотек на C++ с Python. Главное отличие здесь в подходе к описанию биндингов. В отличие от pybind11, где все интерфейсы для нового модуля описываются в обычном .cpp-файле, данный инстумент требует описания интерфейсов в .i-файле, который затем нужно специальным образом собрать. Такой подход может затруднить чтение и описание интерфейсов, так как разработчику придётся разбираться с ещё одним форматом описания данных.

3. **Boost.Python** – это библиотека для создания биндингов для Python от Boost. Описание биндингов в данном проекте похоже на pybind11, но у него есть ряд недостатков по сравнению с pybind11. Во-первых, в pybind11 есть нативная поддержка STL-контейнеров, например *std::vector*, а в boost.Python такой возможности нет. Более того, boost.Python нужно предварительно компилировать (так как это не header-only библиотека), что может быть затратным по времени.

Важно отметить, что за счёт декларативного подхода к описанию биндингов во всех рассмотренных выше решениях, биндинги для одних и тех же объектов у этих библиотек отличаются с точностью до названий функций и методов, типа выходного файла, а также возможностей для описания специфичных объектов. Поэтому можно считать, что задача выбора библиотеки для биндингов и генерация кода не зависят друг от друга.

В итоге была выбрана именно pybind11. В контексте данного проекта, эта библиотека позволяет намного болеее гибко и удобно встраивать и описывать публичный интерфейс. Кроме того, у этого решения есть обширная документация, что позволяет быстро внедрять новые возможности. Поэтому эта библиотека показалась наиболее разумным выбором.

Также можно подчеркнуть, что в данном проекте код реализован так, что при необходимости можно дописать плагин и использовать иную систему построения биндингов.

Пример

На примере 1 с официального сайта можно видеть пример создания простого модуля с единственной функцией. Классы описываются так же легко - необхо-

димо просто через .def указать все поля и методы, что можно увидеть в примере 2. В итоге можно заметить, что для генерации биндингов достаточно лишь по шаблону правильно описать каждый класс и функцию.

```cpp
#include <pybind11/pybind11.h>

int add(int i, int j) {
    return i + j;
}

PYBIND11_MODULE(example, m) {
    m.doc() = "pybind11 example plugin"; // optional module docstring

    m.def("add", &add, "A function that adds two numbers");
}
```

Листинг 1: Пример создания простого модуля с одной функцией

```cpp
// example.hpp
struct Pet {
    Pet(const std::string &name) : name(name) { }
    void setName(const std::string &name_) { name = name_; }
    const std::string &getName() const { return name; }

    std::string name;
};

// bindings file
#include <pybind11/pybind11.h>

namespace py = pybind11;

PYBIND11_MODULE(example, m) {
    py::class_<Pet>(m, "Pet")
        .def(py::init<const std::string &>())
        .def("setName", &Pet::setName)
        .def("getName", &Pet::getName);
}
```

Листинг 2: Пример описания класса

Генерация кода в выходной файл

В проекте реализована простая текстовая генерация. То есть генератор на основе очередного объекта из AST сначала строит строку в зависимости от его типа и имени, затем записывает эту строку в выходной файл.

Такое решение достаточно для описания большинства объектов из C++, а также существенно упрощает разработку, так как не требует никаких сторонних зависимостей.

В дальнейшем возможна интеграция библиотеки inja [1] для генерации кода по шаблону, чтобы генерировать код для всех возможных объектов из C++ и сохранить при этом понятность и стройность кода.

3.3 Сборка сгенерированного исполняемого файла

После генерации необходимо собрать и исполнить сгенерированный файл. В качестве системы сборки был выбран CMake. Это обусловлено тем, что у pybind11 есть поддержка данной системы и удобный макрос для данной задачи – pybind11_add_module для CMake файла. В этот макрос нужно всего лишь указать название модуля на Python и название исполняемого файла, pybind11 далее сделает всю работу сам.

4 Программная часть

Теперь рассмотрим используемые объекты и функции в приложении, а также их взаимодействие между собой. На рисунке 2 указана примерная схема.

4.1 Описание компонентов проекта

RunGen

Формально, главная функция в программе. Принимает на вход контекст исполнения, полученный из параметров пользователя, связывает отдельные объекты программы между собой, конфигурирует и запускает обход синтаксического дерева.

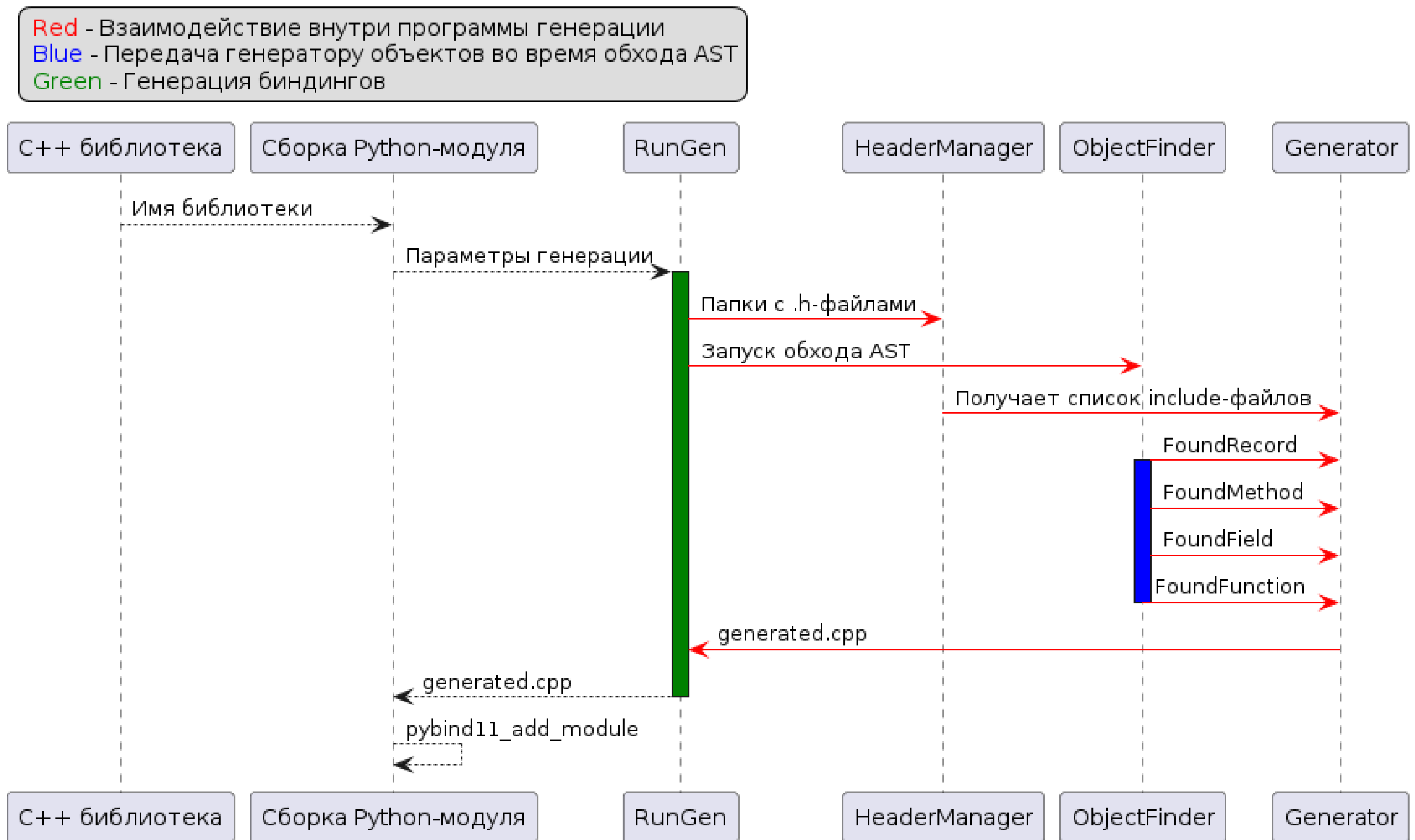

Рис. 2: Взаимодействие объектов во фреймворке

HeaderManager

Вспомогательный класс, необходимый для обхода папок с .h-файлами и получения путей до этих файлов и списка include-файлов. Имеет в публичном интерфейсе 2 метода – $GetHeaders()$ и $GetIncludes()$ для получения информации на основе обхода входных папок. Используется в компоненте *Generator* для корректной работы с хэдерами в сгенерированном файле.

ObjectFinder

Класс, реализующий интерфейс $MatchFinder :: MatchCallback$, который был описан раннее. Хранит в себе множество обработанных объектов, а также указатель на генератор кода. В обработчике проводит фильтрацию, классификацию очередного объекта из AST и передаёт его компоненте *Generator*.

Generator

Класс, реализующий генерацию описания биндингов и запись в выходной файл. Имеет 5 публичных методов, которые генерируют новое описание в зависимости от сущности входного объекта:

1. FoundRecord(const CXXRecordDecl* record) - генерация для класса
2. FoundConstructor(const CXXConstructorDecl* ctor) - генерация для конструктора
3. FoundField(const FieldDecl* field) - генерация для публичного поля класса
4. FoundMethod(const CXXMethodDecl* method) - генерация для публичного метода класса
5. FoundFunction(const FunctionDecl* function) - генерация для функции

Хранит в себе:

(a) Отображение (*class name* $\rightarrow$ *class context*) для найденных классов, где class context – некоторая сущность, необходимая для обработки разных сценариев, например перегрузки функций

(a) Массив из описаний биндингов для найденных функций

(a) std::ofstream для записи в выходной файл *generated.cpp*

Получает от компоненты $HeaderManager$ список include-файлов при инициализации, затем получает объекты для описания биндингов от $ObjectFinder$.

Процесс обработки

Рассмотрим подробнее, как происходит запуск программы и обработка AST.

Программа генерации запускается при сборке CMake-проекта с параметрами, которые зависят от библиотеки пользователя. В частности, $INCLUDE_DIRECTORIES$ C++ проекта пользователя передаются в программу в качестве папок, в которых необходимо рассмотреть .h-файлы.

При запуске задаются 2 объекта типа $DeclarationMatcher$ - они указывают на тип элемента в AST-дереве. Для нас при обходе нужны лишь 2 типа - классы и функции соответственно. Далее создаётся объект типа $ObjectFinder$,

который был описан раннее. Затем создаётся объект класса *MatchFinder* из LibTooling. Он отвечает за связывание *DeclarationMatcher*'ов и *ObjectFinder*, чтобы функция-обработчик вызывалась каждый раз, когда в AST встречается подходящий элемент (класс или функция).

В конце с помощью класса ClangTool запускается обход.

Во время этого обхода подходящие элементы из AST передаются в функцию-обработчик (переопределённый метод run(const MatchFinder::MatchResult& result) у класса *ObjectFinder*). В этом обработчике выполняются 2 действия:

1. Проверяется ряд условий для последующей обработки:
   (a) Была ли эта запись обработана раннее
   (b) Действительно ли эта запись объявлена в одном из входных .h-файлов, или же она просто попала в одном из внешних *#include*
2. Если ни одно из условий выше не выполнено, то в зависимосте от типа мы начинаем обрабатывать эту запись. Если это структура или класс, то мы передаём в генератор саму эту запись, а также все публичные поля, методы и конструкторы. Если это функция, то просто передаём запись в генератор.

При завершении программы генератор записывает в выходной файл *generated.cpp* все данные и CMake подхватывает этот файл для сборки python-модуля.

4.2 Тестирование

Для тестирования написана библиотека на C++, в которой есть различные классы и функции под разные сценарии использования. Во время сборки в Debug режиме генерируется именно эта библиотека, а также во время генерации выводится информация о найденных в AST объектах. На выходе получается .so-файл с библиотекой. Для сгенерированной библиотеки написаны Unit-тесты под каждый сценарий использования биндингов с помощью библиотеки pytest.

4.3 Поддержка других систем сборки

Помимо сборки с CMake, была добавлена возможность собирать библиотеку с системой сборки ya.make. Почти вся сборка осталась такой же, за исключением финальной генерации .so-файла. В этом случае лишь указывается макрос

$PY23_LIBRARY$. При использовании этой библиотеки в другом проекте с такой же системой сборки нужно будет только указать $PEERDIR$ до директории, в которой конфигурировалась сборка python-библиотеки.

5 Заключение

В результате удалось решить все поставленные задачи — изучены возможные решения для анализа AST, создания биндингов и генерации кода, реализованы все логические части фреймворка, описанные во введении, фреймворк протестирован.

Главная цель проекта достигнута — фреймворк для автоматической генерации биндингов из C++ в Python реализован в соответствии с первоначальным планом и критериями.

5.1 Дальнейшее развитие

В дальнейшем необходимо расширить функционал генерации до всех возможностей pybind11. А именно: пользовательские преобразования типов, переопределение методов при наследовании, перегрузка операторов, абстрактные и виртуальные методы и так далее. Кроме того, необходимо переопределить некоторые стандартные преобразования из pybind11 для корректной генерации модуля под сборкой ya.make.

Список литературы

[1] A template engine for modern c++. URL: https://github.com/pantor/inja.

[2] Alexander Fabisch. Cython wrapper. URL: https://github.com/AlexanderFabisch/cythonwrapper/tree/master.

[3] Chris Lattner. The llvm compiler infrastructure. URL: https://llvm.org.

[4] Sergey Lyskov. Binder. URL: https://cppbinder.readthedocs.io/en/latest/index.html.

[5] Jonathan Müller. Cpp ast. URL: https://github.com/standardese/cppast.

Исходный код

Ссылка на репозиторий проекта