Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 21.Б07-мм

# Реализация привязок к MLIR для OCaml

***Хечнев Семен Евгеньевич***

Отчёт по учебной практике
в форме «Решение»

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

Санкт-Петербург
2024

# Оглавление

# Введение

В современном мире существует множество инструментов, реализованных на разных языках программирования. Для того чтобы при разработке чего-либо на каком-либо языке программирования использовать надёжные готовые решения, написанные на других языках, реализуют привязки (от англ. bindings) — интерфейс, предоставляющий набор функций для взаимодействия с библиотеками, написанными на других языках программирования.

LLVM [3] — инфраструктура для разработки компиляторов. LLVM состоит из множества подпроектов. Ядро LLVM базируется на промежуточном представлении LLVM (LLVM IR). LLVM IR это довольно низкоуровневое промежуточное представление, поэтому прямолинейное моделирование на нём высокоуровневых языковых конструкций затруднительно. Чтобы решить эту проблему, а также внедрить специфичные для языка оптимизации добавляют множество уровней абстракции, определяя новые промежуточные представления.

Разработка промежуточных представлений – хорошо изученная область. Однако стоимость проектирования и внедрения промежуточных представлений всё равно остаётся высокой.

Проект MLIR [4] (Multi-Level Intermediate Representation) — это подпроект LLVM, направленный на непосредственное решение проблем проектирования и реализации промежуточных представлений за счёт удешевления определения и внедрения новых уровней абстракции и предоставления инфраструктуры для решения распространенных проблем разработки промежуточных представлений.

Проблема в том, что MLIR реализован на языке C++, но разработчикам, использующим OCaml, также хотелось бы иметь такой мощный инструмент у себя в арсенале. Таким образом, эта работа посвящена реализации привязок к MLIR для языка программирования OCaml.

# 1. Постановка задачи

Целью работы является реализация привязок к MLIR для OCaml. Для её выполнения были поставлены следующие задачи:

1. реализовать привязки к MLIR;
2. разработать компилятор для мини языка программирования с использованием разработанных привязок;
3. реализовать тесты.

# 2. Обзор

## 2.1. SSA форма

Промежуточное представление соответствует SSA [2] форме тогда и только тогда, когда каждой переменной значение присваивается ровно один раз.

SSA форма используется во многих современных компиляторах императивных языков программирования, например, в LLVM, HotSpot JVM и GCC. Использование SSA формы значительно упрощает анализ потока управления, поэтому внедрение многих оптимизаций (например, свёртка констант, удаление неиспользуемого кода и удаление общих подвыражений) становится легче.

## 2.2. MLIR

MLIR [4] (Multi-Level Intermediate Representation) — это гибкая инфраструктура для разработки промежуточных представлений компиляторов на C++. MLIR имеет встроенное (от англ. builtin) промежуточное представление, которое соответствует SSA форме и содержит базовые операции, типы и атрибуты. Из базового промежуточного представления можно создавать собственные промежуточные представления, также в SSA форме, добавляя к базовому промежуточному представлению новые типы и операции, которые могут иметь любую семантику, заданную пользователем.

Далее обзор базовых возможностей MLIR.

### 2.2.1. Базовые объекты MLIR

- **Операции**

  Операция — это основная сущность в MLIR. Всё от инструкций до модулей и функций представляется как операция. MLIR позволяет пользователям определять операции.

  Операция имеет уникальное имя, состоящее из названия диалекта,

которому она принадлежит, и идентификатора. Операция имеет нуль или более операндов и результатов. Кроме того, операция может содержать атрибуты, регионы и информацию о местоположении.

**Листинг 1: Пример операции.**

```
%res:2 = "mydialect.op"(%arg1, %arg2) { attribute1 = true,
    attribute2 = 1.5} : (tensor<*xf64>, i32) -> (f32, i8)
                        loc(callsite("foo" at "mysrc.e":10:8))
```

На листинге 1 определена абстрактная операция «op» из диалекта «mydialect», которая имеет два операнда «%arg1» и «%arg2» типа `tensor<*xf64>` (безранговый тензор с элементами типа `float64`) и `int32` соответственно; два именованных атрибута: «attribute1» со значением `true`, «attribute2» со значением `1.5`; два результата типа `float32` и `int8` соответственно. А также операция имеет информацию о местоположении в исходном тексте программы.

- **Атрибуты**

  Атрибут — это любая compile-time константа, которая прикреплена к операции. Любая операция имеет словарь именованных атрибутов, у которого ключ — имя атрибута, значение — атрибут. Пользователь может определять собственные атрибуты.

- **Типы**

  Каждое значение в MLIR имеет тип. Система типов в MLIR полностью расширяема.

  MLIR предоставляет базовые типы. Например, целочисленные типы, типы с плавающей точкой, строки, кортежи, многомерные вектора, тензоры.

- **Блоки и регионы**

  Блок — это список операций, который заканчивается операцией-терминатором. Блоки образуют граф потока управления програм-

мы (от англ. control flow graph). В MLIR для поддержания SSA формы вместо использования $\varphi$-узлов (как например в LLVM IR) используются аргументы блоков (терминатор блока непосредственно передаёт значения в аргументы следующего блока).

Регион — это список блоков. Любая операция может содержать регионы. При помощи регионов в MLIR реализована вложенная структура: регион содержит список блоков, а блоки содержат список операций, которые, в свою очередь, могут содержать регионы.

- **Диалекты**

  Диалект в MLIR — это пространство имён (от англ. namespace) для операций, типов и атрибутов, при помощи которого MLIR поддерживает расширяемость.

  Базовое промежуточное представление MLIR это не что иное, как диалект, который называется «builtin»[1].

  В репозитории MLIR также содержится множество других диалектов[2].

### 2.2.2. TableGen

MLIR стремится сделать определение новых диалектов и операций, а также правил переписывания (от англ. rewrite rules[3]) паттернов в IR (Intermediate Representation) простым. Для этого MLIR предоставляет декларативный DSL (Domain Specific Language) — TableGen[4], из которого генерируется C++.

Для того, чтобы определить свой диалект необходимо лишь определить класс, который наследуется от базового класса «Dialect». Внутри класса определить имя, краткое описание, полное описание диалекта и имя для пространства имён диалекта в C++.

[1] https://mlir.llvm.org/docs/Dialects/Builtin/ (дата доступа: 4 января 2024 г.).
[2] https://mlir.llvm.org/docs/Dialects/ (дата доступа: 4 января 2024 г.).
[3] https://mlir.llvm.org/docs/DeclarativeRewrites/ (дата доступа: 5 января 2024 г.).
[4] https://mlir.llvm.org/docs/DefiningDialects/Operations/ (дата доступа: 4 января 2024 г.).

**Листинг 2: Пример определения диалекта «MyDialect», при помощи TableGen.**

```
def MyDialect : Dialect {
    let name = "myDialect";

    let summary = "My dialect.";
    let description = [{ My cool dialect. }];

    let cppNamespace = "myDialect";
}
```

**Листинг 3: Пример определения операции сложения тензоров, при помощи TableGen.**

```
def TransposeOp : Op<MyDialect, "transpose", [Pure]> {
    let summary = "transpose operation";
    let description = [{
            The op takes tensor and returns same tensor
            with reversed shape.
    }];

    let arguments = (ins F64Tensor:$arg);
    let results = (outs F64Tensor);

    let assemblyFormat = [{
        "(" $arg ":" type($arg) ")" attr-dict "to" type(results)
    }];
}
```

Для определения операции необходимо указать какому диалекту принадлежит операция, задать уникальное имя (в рамках диалекта), список признаков (от англ. traits), которые задают поведение операции, аргументы, атрибуты и результаты операции. Также операция может содержать краткое и полное описание, из которого можно сгенерировать документацию для операции (в формате Markdown). Кроме того,

для операции можно определять красивый вывод (от англ. pretty print), указывать на наличие верификаторов операции, которые можно определять в C++.

На листинге 3 приведён пример определения операции диалекта «MyDialect», которая моделирует транспонирование тензоров с элементами типа `float64`. Операция называется «transpose» и имеет признак «pure», означающий, что операция детерминирована и не имеет побочных эффектов. Кроме этого, операция имеет красивый вывод.

**Листинг 4: Пример вывода операции, определённой на листинге 3, без переопределённого вывода.**

```
%1 = "myDialect.transpose"(%0):(tensor<2x3xf64>)->tensor<3x2xf64>
```

**Листинг 5: Пример вывода операции, определённой на листинге 3, с переопределённым выводом.**

```
%1 = myDialect.transpose(%0 : tensor<2x3xf64>) to tensor<3x2xf64>
```

**Правила переписывания**

TableGen позволяет декларативно определять правила переписывания[5] шаблонов в IR.

**Листинг 6: Базовый класс для шаблона.**

```
class Pattern<
    dag sourcePattern, list<dag> resultPatterns,
    list<dag> additionalConstraints = [],
    list<dag> supplementalPatterns = [],
    dag benefitsAdded = (addBenefit 0)>;
```

Правило переписывания содержит две главные компоненты: исходный паттерн и итоговый паттерн. При применении правила исходный паттерн в IR будет заменён на итоговый паттерн.

**Пример**

Так как применение чётного количество раз операции транспонирования к тензору это тождественное преобразование, то можно опреде-

[5] https://mlir.llvm.org/docs/DeclarativeRewrites/ (дата доступа: 5 января 2024 г.).

лить правило переписывания, которое будет заменять вызов чётного числа транспонирования некоторого тензора на данный тензор.

**Листинг 7: Пример создания правила переписывания для операции, определённой на листинге 3**

```
// transpose(transpose(x)) -> x
def RedundantTransposeOptPattern :
  Pattern<(TransposeOp(TransposeOp $arg)),
          [(replaceWithValue $arg)]>;
```

Правило переписывания на листинге 7 находит в IR двойной вызов операции «myDialect.transpose» от некоторого аргумента, захватывает аргумент первого вызова и заменяет эту последовательность на аргумент первого вызова.

## 2.3. Взаимодействие OCaml с C

Далее будет кратко рассказано как код, написанный на языке C можно вызывать из OCaml. Более подробно написано здесь[6].

### 2.3.1. Объявление

Для того чтобы объявить внешнюю функцию, в OCaml существует ключевое слово «external».

**Листинг 8: Синтаксис объявления внешней функции.**

```
external name : type = C_function_name
```

**Листинг 9: Пример объявления внешней функции «print_int», которая ссылается на функцию, определённую в C, с именем «caml_print_int».**

```
external print_int : int -> unit = "caml_print_int"
```

[6] https://v2.ocaml.org/manual/intfc.html (дата доступа: 4 января 2024 г.).

### 2.3.2. Определение

С функция должна принимать столько же аргументов, сколько и в объявлении внешней функции (если количество аргументов больше 5, то см. здесь[7]), типа `value` и возвращать значение типа `value`. Тип `value` служит для представления OCaml типа.

**Листинг 10: Содержимое файла C с определением функции «caml_print_int».**

```
#include <stdio.h>
#include <caml/mlvalues.h>

CAMLprim value caml_print_int(value val)
{
    int num = Int_val(val);
    printf("%d\n", num);
    return Val_unit;
}
```

Заголовочный файл «caml/mlvalues.h» предоставляет макросы для преобразования значений OCaml-типа в тип C и наоборот. Функция «print_int», определённая на стороне OCaml, принимает `int` и возвращает `unit`, поэтому аргумент функции «caml_print_int» типа `value` представляет из себя целое число. «Int_val» преобразует значение типа `value` в значение C-типа `int`. Ожидается, что функция вернёт `unit`, поэтому функция «caml_print_int» возвращает значение «Val_unit».

Таким образом, когда внешняя функция вызывается в OCaml, вызывается функция C с переданными аргументами. Затем значение, возвращаемое функцией, передается обратно в OCaml как результат вызова функции.

Реализация внешней функции представляет из себя две задачи:

1. декодирование аргументов для извлечения значений C из значений OCaml и кодирование возвращаемого значения как OCaml-

[7] https://v2.ocaml.org/manual/intfc.html#ss:c-prim-impl (дата доступа: 4 января 2024 г.).

значения;

2. фактическое вычисление какого-то результата из аргументов.

Предпочтительно иметь две отдельные функции C для реализации этих двух задач:

- первая функция реализует основную логику функции, принимая и возвращая значения C;
- вторая — часто называемая «заглушкой» (от англ. stub), представляет собой простую обёртку вокруг первой функции, которая преобразует свои аргументы из значений OCaml в значения C, вызывает первую функцию и преобразует возвращаемое значение C в значение OCaml.

**Листинг 11: Функция «print_int» — реализует логику, функция «caml_print_int» — заглушка**

```
void print_int(int num) {
   printf("%d\n", num);
   return;
}

CAMLprim value caml_print_int(value val)
{
    print_int(Int_val(num));
    return Val_unit;
}
```

### 2.3.3. Линковка

Существует два вида линковки OCaml с C:

1. динамическая;
2. статическая.

Особенности динамической линковки:

1. сохраняет независимость исполняемых файлов, скомпилированных в байт-код, от платформы;
2. размер исполняемого файла меньше чем при статической линковки;
3. пользователям библиотеки не нужно иметь компилятор и линковщик C, а также runtime библиотеки C;
4. итоговый исполняемый файл не является автономным (от англ. standalone): на машине должны быть библиотеки;

## 2.4. Ctypes

Ctypes[8] — это OCaml библиотека для создания привязок к C. Основная цель библиотеки — сделать создание привязок к C максимально простым. Используя эту библиотеку, больше не нужно писать «заглушки» на C.

Так, например, для того чтобы вызвать из OCaml ранее определённую на листинге 11 функцию «print_int», необходимо предоставить имя функции, описать сигнатуру функции при помощи предоставляемых «ctypes» комбинаторов и слинковать C и OCaml.

**Листинг 12: Привязка к функции «print_int», определённой на листинге 11, при помощи «ctypes»**

```
let print_int = foreign "print_int" (int @-> returning void)
```

### 2.4.1. Линковка

Существует два способа линковки привязок:

1. динамическая;
2. статическая генерация привязок.

[8] https://github.com/yallop/ocaml-ctypes (дата доступа: 4 января 2024 г.).

При динамической линковке ctypes использует библиотеку «libffi»[9] для динамического открытия библиотек C, поиска соответствующих символов вызываемой функции и упорядочивания аргументов функции в соответствии с ABI (Application Binary Interface) операционной системы. Многое из этого происходит «под капотом» ctypes, однако такой подход имеет следующие минусы:

- понижение безопасности: библиотеки C не содержат информацию о типах функций, которые они содержат, поэтому не существует способа проверить соответствуют ли переданные типы в «foreign» фактическим типам связываемой функции;
- накладные расходы на интерпретацию.

К счастью, ctypes предоставляет альтернативный способ линковки привязок — статический. В этом случае линковка выполняется один раз во время сборки, вместо того, чтобы выполнять её при каждом вызове функции. При статической линковке:

1. Используя те же описания функций, как на листинге 12, генерируется .ml файл, который содержит объявления внешних функций, и .c файл с заглушками;
2. .ml и .c компилируются и линкуются в одну библиотеку;

## 2.5. Существующие решения

На просторах GitHub была найдена попытка[10] создания привязок к MLIR для OCaml. Данные привязки создавались к форку[11] MLIR, базирующемуся на древней версии MLIR — одиннадцатой, не примитивного примера использования привязок в репозитории не было, и на момент начала работы репозиторий был заброшен. Показалось, что использование некоторой базы может сильно сэкономить время, поэтому эти привязки были взяты за основу.

[9] https://github.com/libffi/libffi (дата доступа: 4 января 2024 г.).
[10] https://github.com/tachukao/ocaml-mlir (дата доступа: 4 января 2024 г.).
[11] https://github.com/tachukao/llvm-project (дата доступа: 5 января 2024 г.).

# 3. Реализация

## 3.1. Привязки

### 3.1.1. Выбор версии MLIR

MLIR является подпроектом LLVM и входит в монорепозиторий LLVM[12], поэтому версия MLIR совпадает с версией LLVM. На момент начала работы версия LLVM 16.0.6 являлась последней версией, к которой существовали привязки[13] для OCaml, поэтому было решено реализовывать привязки к версии MLIR 16.0.6.

### 3.1.2. C API

Из-за применения компиляторами C++ name mangling и различий в модели памяти создать привязки к C++ проблематично. Из-за этого обычно привязки делают к C, который уже вызывает C++. Специально для создания привязок MLIR имеет C API[14], предоставляющий API к базовым возможностям MLIR. Поэтому привязки будут строится именно к этому API.

### 3.1.3. Первая сборка

Привязки к версии 16.0.6 строились поверх форка привязок[15] к форку MLIR одиннадцатой версии, поэтому первым делом было необходимо обновить все зависимости и удалить устаревший код. Кроме того, чтобы не тратить время на сборку MLIR из исходников, было решено сделать возможным построение привязок поверх dev пакетов[16] MLIR и LLVM.

[12] https://github.com/llvm/llvm-project/ (дата доступа: 4 января 2024 г.).
[13] https://opam.ocaml.org/packages/llvm/llvm.16.0.6+nnp/ (дата доступа: 4 января 2024 г.).
[14] https://mlir.llvm.org/docs/CAPI/ (дата доступа: 4 января 2024 г.).
[15] https://github.com/tachukao/ocaml-mlir (дата доступа: 4 января 2024 г.).
[16] https://apt.llvm.org/ (дата доступа: 4 января 2024 г.).

### 3.1.4. Написание привязок

С API одиннадцатой версии MLIR беднее версии 16.0.6. Поэтому следующим шагом необходимо было реализовать недостающие привязки к C API версии 16.0.6 при помощи библиотеки ctypes 2.4.

### 3.1.5. Архитектура

Так как привязки создавались к C, а C относительно OCaml это довольно низкоуровневый язык, поэтому над реализованными привязками находится ещё один уровень абстракции, который позволяет пользователям привязок не задумываться над приведением сложных OCaml типов в примитивные типы C.

Вышеупомянутый дополнительный уровень абстракции над привязками[17] представляет из себя модуль[18], содержащий мини документацию к привязкам и инкапсулирующий преобразование сложных OCaml типов в необходимые для привязок типы, предоставляемые ctypes, которые представляют типы C. Например, список OCaml преобразуется в пару: указатель на первый элемент и число элементов в списке.

## 3.2. Мини язык программирования

У MLIR есть учебное пособие[19], в котором реализуется «игрушечный» язык программирования, для быстрого освоения базовых концепций MLIR.

В качестве доказательства работоспособности и примера использования реализованных привязок было решено реализовать вышеупомянутый язык программирования на OCaml с использованием разработанных привязок.

[17] https://github.com/s-khechnev/ocaml-mlir/tree/llvm-16.0.6/src/stubs (дата доступа: 4 января 2024 г.).

[18] https://github.com/s-khechnev/ocaml-mlir/blob/llvm-16.0.6/src/mlir/mlir.mli (дата доступа: 4 января 2024 г.).

[19] https://mlir.llvm.org/docs/Tutorials/Toy/ (дата доступа: 4 января 2024 г.).

### 3.2.1. Описание языка

Основным и единственным типом данных в данном языке программирования являются тензоры (ранга 0–2, т. е константы, векторы и матрицы) с элементами типа `float64`. Язык позволяет производить различные математические операции над тензорами: поэлементное сложение и умножение, транспонирование и изменение формы тензора. Кроме того, можно определять функции и печатать тензоры. Переменные в языке неизменяемы. В языке статическая типизация, есть вывод типов.

**Листинг 13: Пример программы на мини языке программирования.**

```
def mul_transpose(a, b) {
    return transpose(a) * transpose(b);
}

def main() {
    var a = [[1.1, 2, 3.3], [4, 5.5, 6]];

    # "<2, 3>" reshapes <6> to <2, 3>
    var b<2, 3> = [1, 2.2, 3, 4.4, 5, 6.6];

    var c = mul_transpose(a, b);
    print(c);
}
```

**Процесс компиляции:**

1. Парсинг в AST (Abstract Syntax Tree).
2. Генерация из AST промежуточного представления, которое состоит из операций диалекта высокого уровня, определённого для данного языка при помощи MLIR.
3. Вывод типов.

4. Оптимизация промежуточного представления из шага 2.

5. Частичное понижение (от англ. lowering) промежуточного представления сгенерированного в шаге 2 в промежуточное представление более низкого уровня, состоящее из следующих диалектов:

   - arith[20];
   - affine[21];
   - func[22];
   - memref[23].

6. Оптимизация промежуточного представления из шага 5.

7. Понижение промежуточного представления из шага 5 в промежуточное представление, основанное на диалекте LLVM[24].

8. Понижение промежуточного представления из шага 7 в LLVM IR.

9. Компиляция LLVM IR при помощи JIT компилятора LLVM.

### 3.2.2. AST

Корнем AST является модуль — список функций. Функция представляется как пара: прототип — имя и аргументы, тело. Тело функции состоит из списка выражений.

**Листинг 14: AST языка.**

```
type shape = int array
type var = string

type expr =
    | Num of float
```

[20] https://mlir.llvm.org/docs/Dialects/ArithOps/ (дата доступа: 4 января 2024 г.).
[21] https://mlir.llvm.org/docs/Dialects/Affine/ (дата доступа: 4 января 2024 г.).
[22] https://mlir.llvm.org/docs/Dialects/Func/ (дата доступа: 4 января 2024 г.).
[23] https://mlir.llvm.org/docs/Dialects/MemRef/ (дата доступа: 4 января 2024 г.).
[24] https://mlir.llvm.org/docs/Dialects/LLVM/ (дата доступа: 4 января 2024 г.).

```
    | Literal of shape * expr list
    | Var of var
    | VarDecl of string * shape * expr
    | Return of expr option
    | BinOp of [ `Add | `Mul ] * expr * expr
    | Call of string * expr list
    | Print of expr

type proto = Prototype of string * var list
type func = Function of proto * expr list
type modul = func list
```

### 3.2.3. Парсер

Парсер для мини языка программирования реализован при помощи парсер-комбинаторов. В качестве библиотеки парсер-комбинаторов использовалась библиотека angstrom[25], т.к она является самой популярной среди таковых и обладает лучшей производительностью и документацией среди аналогов.

### 3.2.4. Диалект для мини языка

**Листинг 15: Определение диалекта.**

```
def Toy_Dialect : Dialect {
    let name = "toy";
    let cppNamespace = "::mlir::toy";
}
```

Диалект содержит 10 операций:

1. toy.constant — операция, представляющая константу (тензор);
2. toy.add — операция сложения тензоров;

[25] https://github.com/inhabitedtype/angstrom (дата доступа: 4 января 2024 г.).

3. toy.mul — операция умножения тензоров;
4. toy.cast — вспомогательная операция для встраивания (inlining) функций;
5. toy.func — операция, моделирующая функции;
6. toy.return — операция возврата значения из функции (операции 5);
7. toy.generic_call — операция, представляющая вызов функции (операции 5);
8. toy.print — операция печати тензора в стандартный вывод;
9. toy.reshape — операция изменения формы тензора;
10. toy.transpose — операция транспонирования тензора.

**Листинг 16: Определение при помощи TableGen операции toy.constant.**

```
def ConstantOp : Toy_Op<"constant", [Pure]> {
    let summary = "constant";
    let description = [{
        Constant operation turns a literal into an SSA value.
        The data is attached to the operation as an attribute.

        ```mlir
        %0 = toy.constant dense<[[1.0, 2.0], [3.0, 4.0]]>
                                            : tensor<2x2xf64>
        ```
    }];

    // The constant operation takes an attribute as
    // the only input.
    let arguments = (ins F64ElementsAttr);
```

```
    // The constant operation returns a single
    // value of TensorType.
    let results = (outs F64Tensor);
    ...
}
```

### 3.2.5. Генерация из AST промежуточного представления высокого уровня

Для генерации первого промежуточного представления, содержащего операции из вышеупомянутого диалекта, производился обход AST и последовательная генерация операций диалекта из узлов AST. Создание операции происходит при помощи модуля «OperationState», путём определения имени, позиции в файле, операндов, результатов, и именованных атрибутов будущей операции.

**Листинг 17: Пример программы на мини языке программирования.**

```
def multiply_transpose(a, b) {
    return transpose(a) * transpose(b);
}

def main() {
    var a<2, 3> = [[1, 2, 3], [4, 5, 6]];
    var b<2, 3> = [1, 2, 3, 4, 5, 6];
    var c = multiply_transpose(a, b);
    print(c);
}
```

**Листинг 18: Сгенерированное промежуточное представление высокого уровня для программы листинга 17.**

```
module {
 toy.func @multiply_transpose(%arg0: tensor<*xf64>,
```

```
                                %arg1:tensor<*xf64>) -> tensor<*xf64>
 {
    %0 = toy.transpose(%arg0 : tensor<*xf64>) to tensor<*xf64>
    %1 = toy.transpose(%arg1 : tensor<*xf64>) to tensor<*xf64>
    %2 = toy.mul %0, %1 : tensor<*xf64>
    toy.return %2 : tensor<*xf64>
 }

 toy.func @main() {
    %0 = toy.constant dense<[[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]]>
                                                : tensor<2x3xf64>
    %1 = toy.reshape(%0 : tensor<2x3xf64>) to tensor<2x3xf64>
    %2 = toy.constant dense<[1.0, 2.0, 3.0, 4.0, 5.0, 6.0]>
                                                :tensor<6xf64>
    %3 = toy.reshape(%2 : tensor<6xf64>) to tensor<2x3xf64>
    %4 = toy.generic_call @multiply_transpose(%1, %3)
      : (tensor<2x3xf64>, tensor<2x3xf64>) -> tensor<*xf64>
    toy.print %4 : tensor<*xf64>
    toy.return
 }
}
```

### 3.2.6. Вывод типов (форм тензоров)

Для дальнейших оптимизаций и генераций более низкоуровневых промежуточных представлений необходимо, чтобы все значения имели тензоры определённой формы (на данный момент это не так. Например, на листинге 18 переменная %4 имеет тип `tensor<*xf64>` — безранговый тензор). Поэтому необходимо вывести формы для всех значений из известных форм. Алгоритм вывода следующий:

1. встроить (inline) все вызовы функций в main;
2. распространить (propagate) формы через все вычисления:

(a) составить список всех операций которые возвращают безранговый тензор (это и будут те операции, для которых необходимо произвести вывод);

(b) найти в списке 2a операцию, у которой все операнды имеют тензор с определённой формой (если таковой не нашлось, то закончить вывод);

(c) удалить операцию из списка 2a;

(d) вывести форму для возвращаемого значения операции 2b из формы её операндов (например, если выводим для операции умножения (операция умножения поэлементная!), то форма возвращаемого значения равна форме любого операнда; для транспонирования форма возвращаемого значения равна перевёрнутой (reversed) форме операнда);

(e) если список 2a пустой, то завершить вывод, иначе перейти к 2b.

### 3.2.7. Оптимизации для промежуточного представления, основанного на диалекте мини языка

Хорошим примером иметь такое высокоуровневое промежуточное представление является возможность произвести оптимизацию, которая исключает последовательность двух транспонирований тензоров.

Для реализации этой оптимизаций было декларативно (при помощи TableGen) определено такое же правило перезаписи, как и на листинге 7.

Кроме того, были добавлены оптимизации для операции изменения формы тензора («toy.reshape»):

- если производилась попытка изменить форму тензора на ту же форму, то операция «toy.reshape» удалялась;
- при использовании операнда изменения формы тензора, форма изменялась непосредственно в операции «toy.constant».

**Замечание**. После этих оптимизаций операции «toy.reshape» больше нет в промежуточном представлении.

Более того, был добавлен проход (от англ. pass) «cse» — common subexpression elimination (удаление общих подвыражений), который уже определён в MLIR.

**Листинг 19: Сгенерированное промежуточное представление для программы листинга 17 после вывода форм и определения вышеупомянутых оптимизаций.**

```
module {
 toy.func @main() {
    %0 = toy.constant dense<[[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]]>
                                                : tensor<2x3xf64>
    %1 = toy.transpose(%0 : tensor<2x3xf64>) to tensor<3x2xf64>
    %2 = toy.mul %1, %1 : tensor<3x2xf64>
    toy.print %2 : tensor<3x2xf64>
    toy.return
 }
}
```

### 3.2.8. Понижение промежуточного представления в более низкоуровневое

Для внедрения новых оптимизаций и дальнейшего понижения до LLVM IR необходимо понизить текущее высокоуровневое промежуточное представление в более низкоуровневое, которое в дальнейшем будем называть аффинным, состоящее из следующих диалектов:

- affine[26] — предоставляет абстракцию над аффинными преобразованиями. Для наших целей предоставляет операцию «affine.for»[27], которая моделирует цикл;

[26] https://mlir.llvm.org/docs/Dialects/Affine/ (дата доступа: 4 января 2024 г.).

[27] https://mlir.llvm.org/docs/Dialects/Affine/#affinefor-affineaffineforop (дата доступа: 4 января 2024 г.).

- arith[28] — предоставляет операции базовых математических операций;
- func[29] диалект содержит операции, моделирующие создание и взаимодействие с функциями;
- memref[30] — предоставляет операции для управления памятью, которые используют тип `memref`[31], представляющий многомерный буфер.

Каждая операция из 3.2.4 транслировалась в набор операций из вышеупомянутых диалектов.

**Замечание**. После раннее добавленных проходов в промежуточном представлении больше нет операций «toy.cast», «toy.generic_call» и «toy.reshape».

- Для операции «toy.constant» сначала генерировались операции «arith.constant», которые представляют обычные `float64` константы, затем аллоцировался и инициализировался этими константами буфер «memref», который представлял в памяти тензор.
- Для операций «toy.add» и «toy.mul» генерировались вложенные циклы «affine.for», итерирующиеся по всем элементам буфера, в теле которых происходило поэлементное складывание или умножение элементов буферов при помощи операций «arith.mulf» и «arith.addf».
- Для операции «toy.transpose» генерировался дополнительный буфер с перевёрнутой (reversed) формой, и создавался цикл, который заполнял новый буфер в соответствии с семантикой операции транспонирования.
- Операции «toy.func» и «toy.return» прямолинейно транслировались в операции «func.func» и «func.return» соответственно.

[28] https://mlir.llvm.org/docs/Dialects/ArithOps/ (дата доступа: 4 января 2024 г.).
[29] https://mlir.llvm.org/docs/Dialects/Func/ (дата доступа: 4 января 2024 г.).
[30] https://mlir.llvm.org/docs/Dialects/MemRef/ (дата доступа: 4 января 2024 г.).
[31] https://mlir.llvm.org/docs/Dialects/Builtin/#memreftype (дата доступа: 4 января 2024 г.).

- Операция «toy.print» транслируется в вызов внешней функции «printMemrefF64», которая содержится в библиотеке «libmlir_runner_utils» (эта библиотека содержится в MLIR и представляет из себя небольшую runtime библиотеку для облегчения ввода/вывода). Для этого сначала генерируется forward-declaration функции «printMemrefF64», затем, т. к эта функция принимает безранговый буфер, для операнда операции «toy.print» генерируется операция «memref.cast», которая конвертирует буфер с рангом в безранговый, потом генерируется операция «func.call» для вызова функции «printMemrefF64».

### 3.2.9. Оптимизации аффиного промежуточного представления

После понижения в аффинное промежуточное представление были добавлены уже реализованные в MLIR проходы: «cse» (удаление общих подвыражений), «affine-loop-fusion»[32] (слияние циклов), «affine-scalar-replacement»[33] (удаляет избыточные обращения к буферам).

**Листинг 20: Сгенерированное промежуточное представление для программы листинга 17 после понижения до аффинного промежуточного представления и добавления вышеупомянутых оптимизаций.**

```
module {
  func.func @printMemrefF64(memref<*xf64>)
  func.func @main() {
   %cst = arith.constant 1.000000e+00 : f64
   %cst_0 = arith.constant 2.000000e+00 : f64
   %cst_1 = arith.constant 3.000000e+00 : f64
   %cst_2 = arith.constant 4.000000e+00 : f64
   %cst_3 = arith.constant 5.000000e+00 : f64
   %cst_4 = arith.constant 6.000000e+00 : f64
   %alloc = memref.alloc() : memref<3x2xf64>
   %alloc_5 = memref.alloc() : memref<2x3xf64>
```

[32] https://mlir.llvm.org/docs/Passes/#-affine-loop-fusion (дата доступа: 4 января 2024 г.).
[33] https://mlir.llvm.org/docs/Passes/#-affine-scalrep (дата доступа: 4 января 2024 г.).

```
    affine.store %cst_4, %alloc_5[1, 2] : memref<2x3xf64>
    affine.store %cst_3, %alloc_5[1, 1] : memref<2x3xf64>
    affine.store %cst_2, %alloc_5[1, 0] : memref<2x3xf64>
    affine.store %cst_1, %alloc_5[0, 2] : memref<2x3xf64>
    affine.store %cst_0, %alloc_5[0, 1] : memref<2x3xf64>
    affine.store %cst, %alloc_5[0, 0] : memref<2x3xf64>
    affine.for %arg0 = 0 to 3 {
      affine.for %arg1 = 0 to 2 {
        %0 = affine.load %alloc_5[%arg1, %arg0] : memref<2x3xf64>
        %1 = arith.mulf %0, %0 : f64
        affine.store %1, %alloc[%arg0, %arg1] : memref<3x2xf64>
      }
    }
    %cast = memref.cast %alloc : memref<3x2xf64> to memref<*xf64>
    call @printMemrefF64(%cast) : (memref<*xf64>) -> ()
    memref.dealloc %alloc_5 : memref<2x3xf64>
    memref.dealloc %alloc : memref<3x2xf64>
    return
  }
}
```

### 3.2.10. Понижение промежуточного представления в диалект LLVM

Диалект LLVM[34] полностью соответствует LLVM IR. Для большинства диалектов, которые находятся в репозитории MLIR, понижение в диалект LLVM уже определено. Поэтому понизить аффинное промежуточное представление не представляло труда.

### 3.2.11. Понижение в LLVM IR и компиляция при помощи JIT

После всех проделанных манипуляций оставалось лишь передать MLIR-модуль JIT-компилятору, который неявно понизит LLVM диа-

[34] https://mlir.llvm.org/docs/Dialects/LLVM/ (дата доступа: 4 января 2024 г.).

лект в LLVM IR и скомпилирует программу, а также не забыть слинковать библиотеку «libmlir_runner_utils», которая используется для печати тензоров.

**Листинг 21: Вывод для программы листинга 17.**

```
[[1,   16],
 [4,   25],
 [9,   36]]
```

# 4. Тестирование

## Привязки

У привязок уже были тесты, которые повторяют часть тестов MLIR C API одиннадцатой версии. Поэтому оставалось внести небольшие правки в существующих тестах, чтобы они заработали для версии MLIR 16.0.6.

## Мини язык программирования

Для каждого этапа компиляции мини языка программирования были реализованы тесты.

## Тестовое покрытие

Итоговое тестовое покрытие всего проекта составило 55%. При этом тестовое покрытие компилятора мини языка программирования составляет 96%, привязок — 46%.

## Continuous integration

При помощи GitHub Actions было реализовано Continuous integration, в котором происходит сборка и запуск тестов, а также обновление отчёта о покрытии.

# Заключение

Таким образом, были выполнены все поставленные задачи:

- реализованы привязки к MLIR;
- разработан компилятор для мини языка программирования с использованием разработанных привязок;
- реализованы тесты.

Исходный код проекта: https://github.com/s-khechnev/ocaml-mlir.
Имя аккаунта: s-khechnev.

## Дальнейшие планы

В дальнейшем планируется создать компилятор для подмножества некоторого ML языка с использованием MLIR и идеи, основанной на использовании регионов MLIR как функций первого класса [1].

# Список литературы


[1] Bhat Siddharth, Grosser Tobias. Lambda the Ultimate SSA: Optimizing Functional Programs in SSA // CoRR. — 2022. — Vol. abs/2201.07272. — arXiv : 2201.07272.

[2] Efficiently Computing Static Single Assignment Form and the Control Dependence Graph / Ron Cytron, Jeanne Ferrante, Barry K. Rosen et al. // ACM Trans. Program. Lang. Syst. — 1991. — oct. — Vol. 13, no. 4. — P. 451–490. — URL: https://doi.org/10.1145/115372.115320.

[3] Lattner Chris, Adve Vikram. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation // Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization. — CGO '04. — USA : IEEE Computer Society, 2004. — P. 75.

[4] MLIR: A Compiler Infrastructure for the End of Moore's Law / Chris Lattner, Jacques A. Pienaar, Mehdi Amini et al. // CoRR. — 2020. — Vol. abs/2002.11054. — arXiv : 2002.11054.