Компиляторы без велосипедов:

что там в MLIR, Mojo 🔥 и вокруг

Василий Рябов

Эксперт Huawei

~~Что такое~~ Кто такой «новичок»?

Василий Рябов

С 2004 писал на C++, чистом C, Python (2008+), смеси Python и C.

В теме компиляторов – крайние 1,5 года, с нуля.

Что такое компиляторы?

Frontend (лексика+синтаксис)

Исходный код => токены (лексемы) => AST (Abstract Syntax Tree)

Генераторы парсеров: ANTLR, PEG, ...

Middle end (семантика)

Машинно-независимые IR'ы (intermediate repr. == промежут. представление)

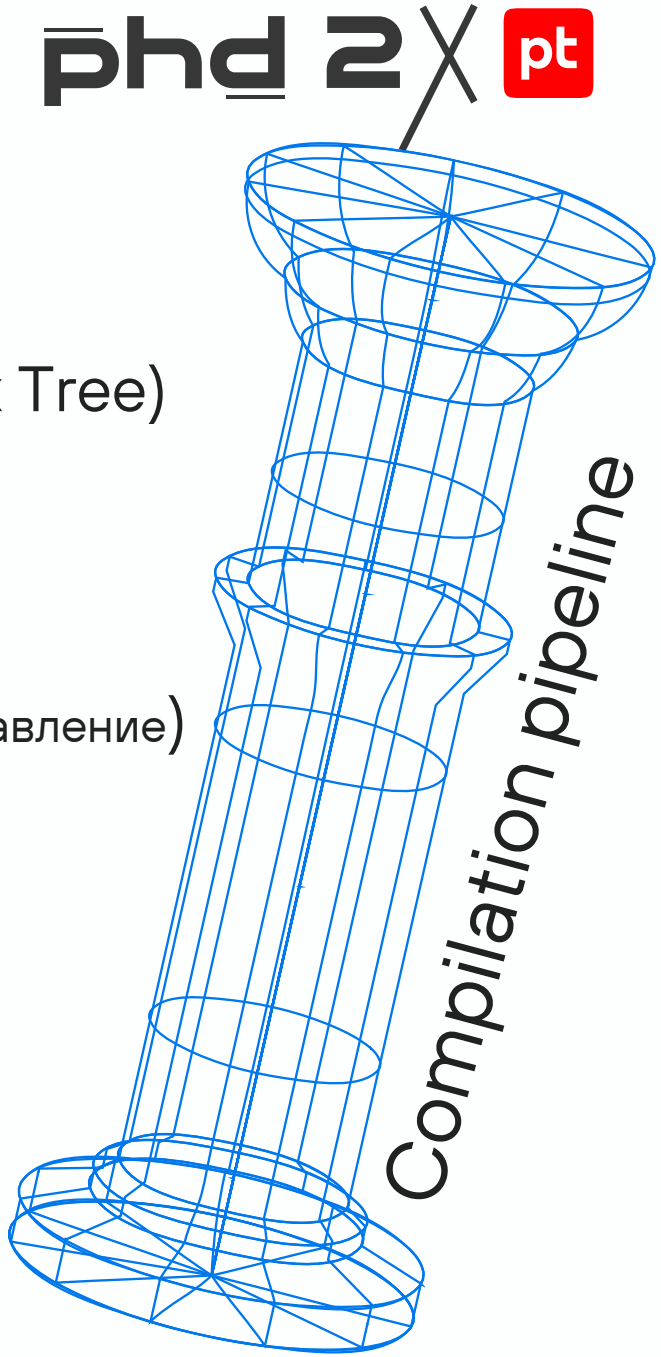
Проверка и вывод типов, общие оптимизации, исключения, ...

Backend («90% всей работы»)

Машинно-зависимые IR'ы

оптимизации => аллокация регистров => Assembler => машинный код.

+ JIT (Just-in-Time) движок



Что мы будем делать?

Frontend

«парсер есть, ума не надо!» (с)

Вызовем из Python 3.9

- т.к. в 3.10+ функция `PyParser_ASTFromFile(...)` скрыта из C API

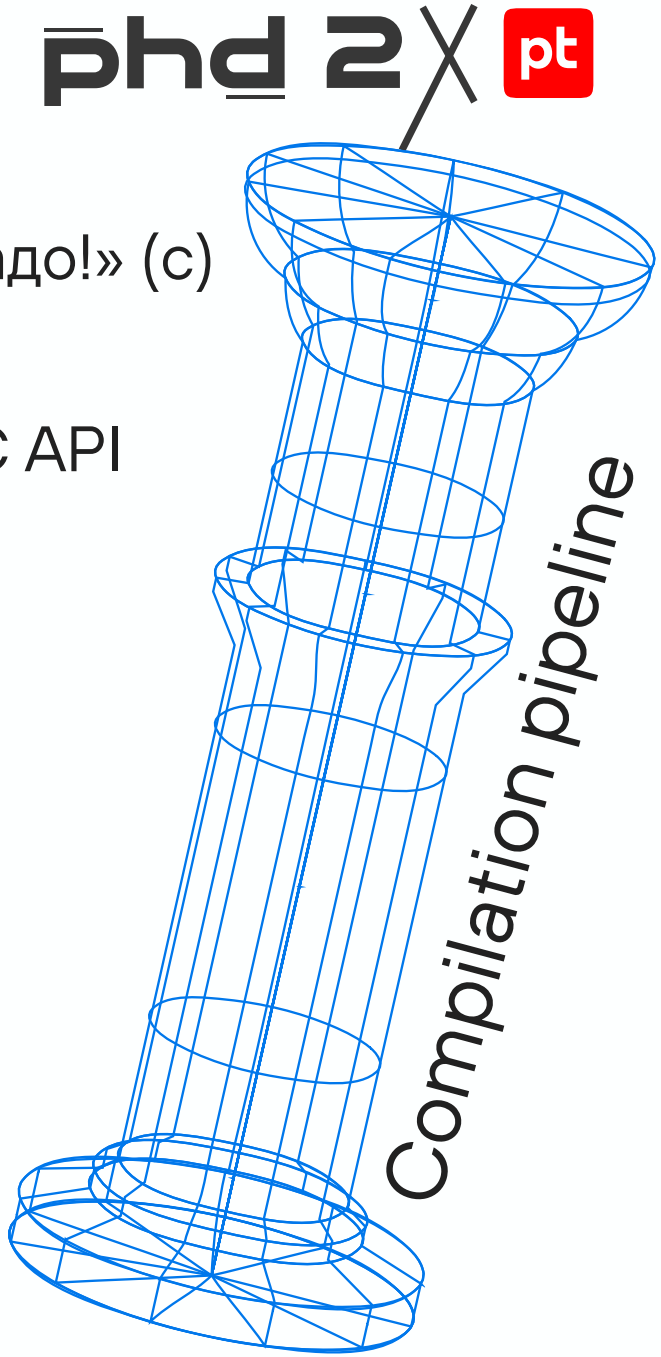
Middle end

Сгенерируем IR с помощью библиотеки MLIR

Понизим и отдадим ниже

Backend

Вызовем JIT (Just-in-Time) движок из LLVM



(middle end) Что такое MLIR?

- Библиотека Multi-Level IR (2019, Крис Латтнер)
  - Диалекты высокого и низкого уровня
  - Разные диалекты в одном IR (путь к совместимости языков через IR? ~10+ лет)
- Проект LLVM: llvm/llvm-project (папка mlir/)
- Построены на MLIR:

В процессе перехода на MLIR:

clang IR (~30%) (мой прогноз: ещё 2-3 года)

Как выглядит IR? (1/3)

- IR является SSA формой (Static Single Assignment) в MLIR и LLVM IR
  - каждое значение имеет уникальное имя/номер, присваивается один раз
  - каждое значение может использоваться много раз

У любого IR есть три представления:

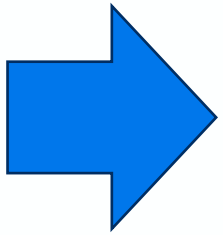
1. Представление в памяти (in-memory representation) ~= C++ классы
2. Байткод: сериализованное бинарное представление в файле (*.bc)
3. Текстовый IR (*.mlir): не просто dump, его можно парсить в память (представление 1)

- Функции не могут быть вложенными
- Нет аналога #include / import другого IR модуля (один IR == одна DLL / Shared Lib)

Как выглядит IR? (2/3)

```
%результат = диалект.операция @имя (%операнды) -> (тип_результата) { регион }
```

```
if True:
    a = 5
else:
    a = 7
b = a
```



```
func.func @main() -> i64 {
    %0 = arith.constant 1 : i1
    %2 = scf.if %0 -> (i64) {
            %1 = arith.constant 5 : i64
            scf.yield %1 : i64
        } else {
            %1 = arith.constant 7 : i64
            scf.yield %1 : i64
        }
  func.return %2 : i64
}
```

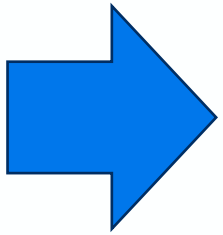
Как выглядит IR? (2/3)

```
%результат = диалект.операция @имя (%операнды) -> (тип_результата) { регион }
```

Диалекты:

- “scf” – structured control flow
- “func” – функции высшего порядка
- “arith” – арифметические операции

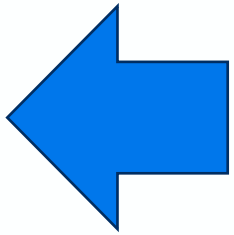
```
if True:
    a = 5
else:
    a = 7
b = a
```



```
func.func @main() -> i64 {
    %0 = arith.constant 1 : i1
    %2 = scf.if %0 -> (i64) {
            %1 = arith.constant 5 : i64
            scf.yield %1 : i64
        } else {
            %1 = arith.constant 7 : i64
            scf.yield %1 : i64
        }
  func.return %2 : i64
}
```

Как выглядит IR? (3/3)

```
llvm.func @main() -> i64 {
    %0 = llvm.mlir.constant(true) : i1
    llvm.cond_br %0, ^bb1, ^bb2
^bb1:
    %1 = llvm.mlir.constant(5 : i64) : i64
    llvm.br ^bb3(%1 : i64)
^bb2:
    %2 = llvm.mlir.constant(7 : i64) : i64
    llvm.br ^bb3(%2 : i64)
^bb3(%3: i64):
    llvm.br ^bb4
^bb4:
    llvm.return %3 : i64
}
```



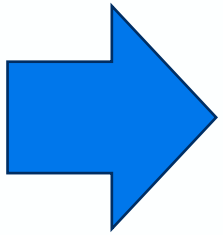
https://godbolt.org/z/j5WeEf8a4

```
func.func @main() -> i64 {
    %0 = arith.constant 1 : i1
    %2 = scf.if %0 -> (i64) {
            %1 = arith.constant 5 : i64
            scf.yield %1 : i64
        } else {
            %1 = arith.constant 7 : i64
            scf.yield %1 : i64
        }
  func.return %2 : i64
}
```

В чём отличия MLIR и LLVM IR?

MLIR диалект “llvm”

```
    %0 = llvm.mlir.constant(true) : i1
    llvm.cond_br %0, ^bb1, ^bb2
^bb1:
    %1 = llvm.mlir.constant(5 : i64) : i64
    llvm.br ^bb3(%1 : i64)
^bb2:
    %2 = llvm.mlir.constant(7 : i64) : i64
    llvm.br ^bb3(%2 : i64)
^bb3(%3: i64):
    llvm.br ^bb4
^bb4:
    llvm.return %3 : i64
```



LLVM IR

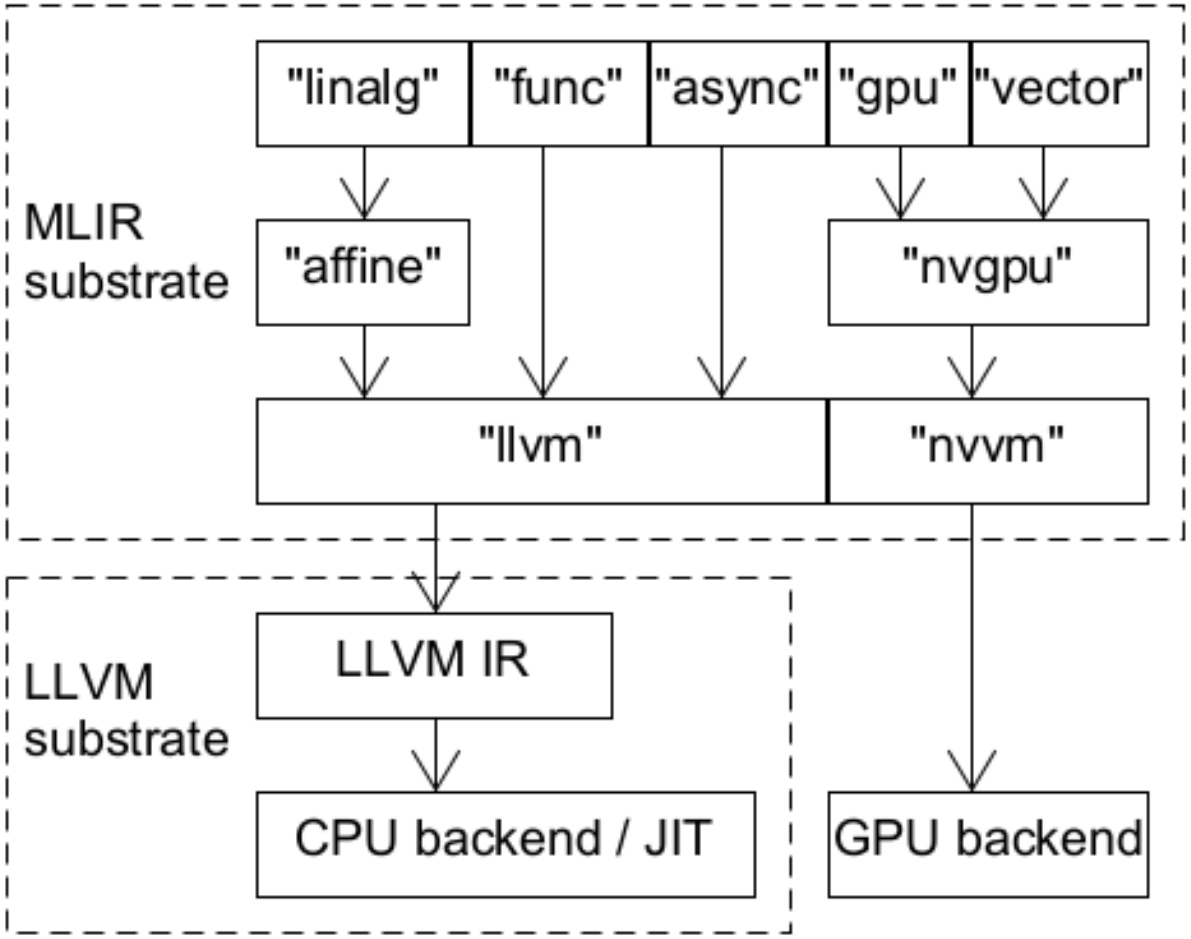
```
    br i1 true, label %1, label %2
1:
    br label %3
2:
    br label %3
3:
    %4 = phi i64 [ 7, %2 ], [ 5, %1 ]
    br label %5
5:
    return %4 : i64
```

Что установить для MLIR?

- GitHub репозиторий к данному докладу: vasily-v-ryabov/phdays24
  - Скрипты, CMake файлы и исходный код

- Для Ubuntu/Debian (WSL на Windows) есть apt репозитории
  - Уже собранные библиотеки и утилиты LLVM и MLIR версий 17.x и 18.x
  - install.sh
- Для Windows придётся взять LLVM репозиторий и собрать MLIR
  - Сборка занимает примерно 1 час
  - Должна быть установлена Visual Studio 2022 (достаточно Community Edition)
  - install.bat

Что происходит в MLIR?



- доклад о субстратах

1. **Генерация IR верхнего уровня**
   (через обход Python AST)
2. **Понижение шаг за шагом**

(Lowering/Conversion – часть диалекта)

3. **Трансляция в LLVM IR**
4. **Исполнение на LLVM backend'е**

Как будем генерировать IR?

Проект “Toy”: llvm/llvm-project: ./mlir/examples/toy/ (7 глав)

- 1) фронтенд, **2) MLIR gen**, 3) диалект, 4) pass, 5) свой lowering pass, **6) ->LLVM IR, 7) JIT**.

В vasily-v-ryabov/phdays24 свои 5 глав:

1)	Простейший IR	(22 строки на C++)
2)	IR с переменными (с фронтендом из Python 3.9)	(+199 строк)
3)	IR с условиями, выводом типов и понижением	(+59 строк)
4)	Понижаем и транслируем в LLVM IR (с опциями)	(+49 строк)
5)	Подключаем JIT движок	(+34 строки)
		всего 363 строки

Как будем генерировать IR?

В vasily-v-ryabov/phdays24 свои 5 глав:

1) **Простейший IR**
2) IR с переменными (с фронтендом из Python 3.9)
3) IR с условиями, выводом типов и понижением
4) Понижаем и транслируем в LLVM IR (с опциями)
5) Подключаем JIT движок

Ура! Первый код: main.cpp (1/3)

./phdays24/01_MLIR_gen/main.cpp

```
#include "mlir/IR/MLIRContext.h"
#include "mlir/IR/Verifier.h"

#include "mlir/Dialect/LLVMIR/LLVMDialect.h"

mlir::ModuleOp mlirGen(mlir::MLIRContext &context) {
  mlir::OpBuilder builder(&context);
  context.getOrLoadDialect<mlir::LLVM::LLVMDialect>();
  ...
  auto loc = builder.getUnknownLoc();

  auto module = mlir::ModuleOp::create(loc);
  builder.setInsertionPointToEnd(module.getBody());
  ...
```

Ура! Первый код: main.cpp (2/3)

```
  ...
  // create function main()
  auto mainFuncType = mlir::LLVM::LLVMFunctionType::get(builder.getI32Type(), {});
  auto mainFunc = builder.create<mlir::LLVM::LLVMFuncOp>(loc, "main", mainFuncType);

  mlir::Block *entryBlock = mainFunc.addEntryBlock();
  builder.setInsertionPointToStart(entryBlock);

  // function body (return 0;)
  auto constOp = builder.create<mlir::LLVM::ConstantOp>(loc, builder.getI32Type(), 0);
  builder.create<mlir::LLVM::ReturnOp>(loc, constOp->getResult(0));
  return module;
}
```

Ура! Первый код: main.cpp (3/3)

```
int main(int argc, char **argv) {
  mlir::MLIRContext context;
  mlir::OwningOpRef<mlir::ModuleOp> module = mlirGen(context);
  if (!module)
    return 1;
  if (mlir::failed(mlir::verify(*module))) {
    module->emitError("Module verification failed!");
    module->dump(); // dump incorrect IR anyway
    return 2;
  }
  module->dump(); // to stderr
  return 0;
} // 22 non-empty lines of code
```

Ура! Первый код: main.cpp (3/3)

```cpp
int main(int argc, char **argv) {
  mlir::MLIRContext context;
  mlir::OwningOpRef<mlir::ModuleOp> module = mlirGen(context);
  if (!module)
    return 1;
  if (mlir::failed(mlir::verify(*module))) {
    module->emitError("Module verification failed!");
    module->dump(); // dump incorrect IR anyway
    return 2;
  }
  module->dump(); // to stderr
  return 0;
} // 22 non-empty lines of code
```

```
root@MSI:~/phdays24/01_MLIR_gen/build# ./bin/py39compiler
module {
  llvm.func @main() -> i32 {
    %0 = llvm.mlir.constant(0 : i32) : i32
    llvm.return %0 : i32
  }
}
```

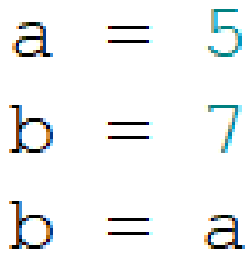
Как будем генерировать IR?

```
a = 5
b = 7
b = a
```

В vasily-v-ryabov/phdays24 свои 5 глав:

1) Простейший IR
2) **IR с переменными** (с фронтендом из Python 3.9)
3) IR с условиями, выводом типов и понижением
4) Понижаем и транслируем в LLVM IR (с опциями)
5) Подключаем JIT движок

Как добавить переменные? (1/4)



./phdays24/02_MLIR_gen_pyvars/include/py_ast.h

- класс-обёртка для функции `RỳRắsṣês AŞŢGsộṇGîľê`

```
class PyAST {
...
  bool parse_file(const char *name) { ... }
  mod_ty mod() { ... } // mod_ty is a pointer to C structure in Python.h
};
```

Как добавить переменные? (2/4)

./phdays24/02_MLIR_gen_pyvars/include/MLIRGen.h

```
mlir::LogicalResult mlirGen(mod_ty pyModule) { ... }
mlir::LogicalResult mlirGen(asdl_seq *statements) { ... }
mlir::LogicalResult mlirGen(stmt_ty statement) {
  switch (statement->kind) {
    ...
    case Assign_kind: {
      auto valOrErr = mlirGen(/*expr_ty*/statement->v.Assign.value); // right side
      ... // left side
      expr_ty astTarget = (expr_ty)asdl_seq_GET(statement->v.Assign.targets, 0);
      ...
}
mlir::FailureOr<mlir::Value> mlirGen(expr_ty expr) { ... } // case Constant_kind:
```

Как добавить переменные? (3/4)

a = 5
b = 7
b = a

- В правой части есть **имя**, но откуда взять **mlir::Value** ?
- Нам поможет символьная таблица! Но не простой hash_map...

```
#include "llvm/ADT/ScopedHashTable.h"
...
llvm::ScopedHashTable<llvm::StringRef, mlir::Value> symbolTable;
...
// at least one scope is required
llvm::ScopedHashTableScope<llvm::StringRef, mlir::Value> scope(symbolTable);
```

Как добавить переменные? (4/4)

```
void defineVariable(llvm::StringRef name, mlir::Value value) {
  llvm::outs() << "Add variable '" << name << "' = '" << value << "'\n";
  llvm::MallocAllocator ma;
  symbolTable.insert(name.copy(ma), value);
}
mlir::FailureOr<mlir::Value> getVariable(mlir::Location loc, llvm::StringRef name) {
    auto value = symbolTable.lookup(name);
    if (value) // may be nullptr!
      return value;
    mlir::emitError(loc, "Variable '") << name << "' is not defined\n";
    return mlir::failure();
  }
```

- **MLIRGen.h**: ~200 строк, в репозитории: папка **02_MLIR_gen_pyvars/**
- **py_ast.h**: ~50 строк

Как добавить переменные? (4/4)

```
void defineVariable(llvm::StringRef name, mlir::Value value) {
  llvm::outs() << "Add variable '" << name << "' = '" << value << "'\n";
  llvm::MallocAllocator ma;
  symbolTable.insert(name.copy(ma), value);
}
```

```
root@MSI:~/phdays24/02_MLIR_gen_pyvars# ./build/bin/py39compiler script.py
Add variable 'a' = '%0 = "llvm.mlir.constant"() <{value = 5 : i64}> : () -> i64'
Add variable 'b' = '%1 = "llvm.mlir.constant"() <{value = 7 : i64}> : () -> i64'
Add variable 'b' = '%0 = "llvm.mlir.constant"() <{value = 5 : i64}> : () -> i64'
module {
  llvm.func @main() -> i32 {
    %0 = llvm.mlir.constant(5 : i64) : i64
    %1 = llvm.mlir.constant(7 : i64) : i64
    %2 = llvm.mlir.constant(0 : i32) : i32
    llvm.return %2 : i32
  }
}
```

А что с глобальными?

1) Создать LLVM::GlobalOp (“llvm.mlir.global”) – это всегда указатель
2) Получить адрес по имени: LLVM::AddressOfOp (“llvm.addressof”)
3) Загрузка по указателю: LLVM::LoadOp (“llvm.load”)
4) Или запись по указателю: LLVM::StoreOp (“llvm.store”)

- Нету scopes!
- “llvm.mlir.global” – аналог insert, но в runtime
- “llvm.addressof”+“llvm.load” – аналог lookup, но в runtime
- Функции тоже являются global’ами
  - Есть `mlir::SymbolTable globalTable(module);`
  - Есть insert (не в IR) и lookup (полезен, чтобы наполнять тело функции позднее)
  - Есть module.lookupSymbol(name);
  - Если “llvm.addressof” вернул функцию, её можно позвать по указателю (indirect call)

Как будем генерировать IR?

```
if True:
    a = 5
else:
    a = 2
b = a
```

В vasily-v-ryabov/phdays24 свои 5 глав:

1) Простейший IR
2) IR с переменными (с фронтендом из Python 3.9)
3) **IR с условиями и выводом типов**
4) Понижаем и транслируем в LLVM IR (с опциями)
5) Подключаем JIT движок

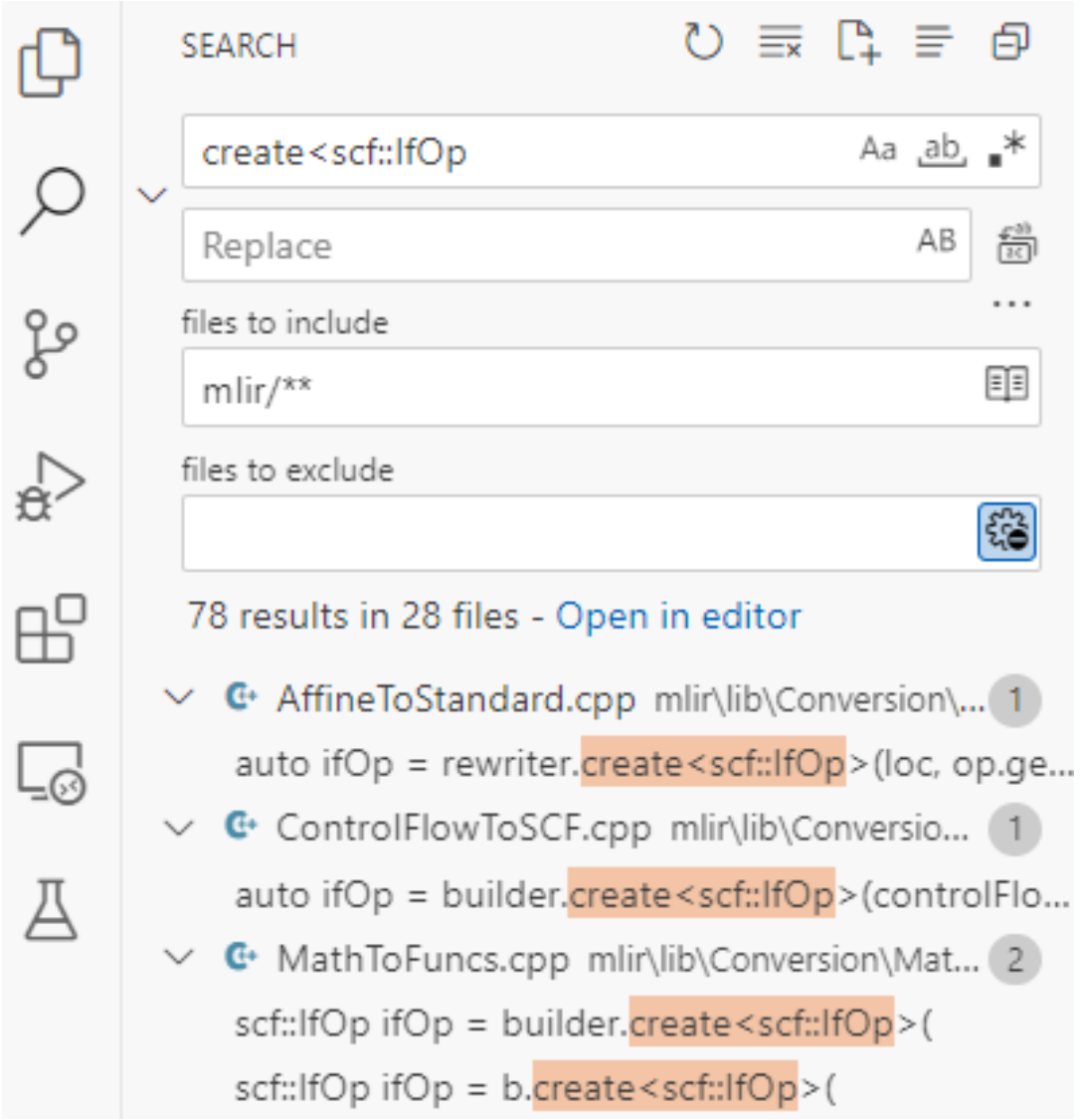
Как добавить условия? (1/5)

```
#include "mlir/Dialect/SCF/IR/SCF.h"
#include "mlir/Dialect/ControlFlow/IR/ControlFlow.h"
...
context.getOrLoadDialect<mlir::scf::SCFDialect>();
context.getOrLoadDialect<mlir::cf::ControlFlowDialect>();
...

auto ifOp = builder.create<mlir::scf::IfOp>(loc, ...); // parameters?
```

```
if True:
    a = 5
else:
    a = 2
b = a
```

Как находить билдеры?

- В доках по диалекту их нет!
- Искать придётся в исходном коде MLIR



Как находить билдеры?

- В доках по диалекту их нет!
- Искать придётся в исходном коде MLIR

- Если пойти в класс scf::IfOp, ...

```
static void build(::mlir::OpBuilder &odsBuilder, ::mlir::OperationState &odsState, TypeRange resultTypes, Value cond);
static void build(::mlir::OpBuilder &odsBuilder, ::mlir::OperationState &odsState, TypeRange resultTypes, Value cond, bool addThenBlock, bool addElseBlock);
static void build(::mlir::OpBuilder &odsBuilder, ::mlir::OperationState &odsState, Value cond, bool withElseRegion);
static void build(::mlir::OpBuilder &odsBuilder, ::mlir::OperationState &odsState, TypeRange resultTypes, Value cond, bool withElseRegion);
static void build(::mlir::OpBuilder &odsBuilder, ::mlir::OperationState &odsState, Value cond, function_ref<void(OpBuilder &, Location)> thenBuilder = buildT
```

Как находить билдеры?

- В доках по диалекту их нет!
- Искать придётся в исходном коде MLIR

- Если пойти в класс scf::IfOp, ...

```
, TypeRange resultTypes, Value cond);
, TypeRange resultTypes, Value cond, bool addThenBlock, bool addElseBlock);
, Value cond, bool withElseRegion);
, TypeRange resultTypes, Value cond, bool withElseRegion);
, Value cond, function_ref<void(OpBuilder &, Location)> thenBuilder = buildTerminatedBody,
```

Как добавить условия? (2/5)

```
auto valueOrError = mlirGen(statement->v.If.test);
if (mlir::failed(valueOrError))
  return mlir::failure();
auto condition = valueOrError.value(); // TODO: check type is i1 (integer of 1 bit)
auto ifOp = builder.create<mlir::scf::IfOp>( // it calls inferReturnTypes(...)
    loc, condition,
    /*thenBuilder=*/[&](mlir::OpBuilder &b, mlir::Location loc) {

      b.create<mlir::scf::YieldOp>(loc /*returned values*/);
    },
    /*elseBuilder=*/[&](mlir::OpBuilder& b, mlir::Location loc) {

      b.create<mlir::scf::YieldOp>(loc /*returned values*/);
    }
);
```

Как добавить условия? (2/5)

```
auto valueOrError = mlirGen(statement->v.If.test);
if (mlir::failed(valueOrError))
  return mlir::failure();
auto condition = valueOrError.value(); // TODO: check type is i1 (integer of 1 bit)
auto ifOp = builder.create<mlir::scf::IfOp>( // it calls inferReturnTypes(...)
    loc, condition,
    /*thenBuilder=*/[&](mlir::OpBuilder &b, mlir::Location loc) {
      mlirGen(statement->v.If.body); // Assign_kind AST-nodes can be inside
      b.create<mlir::scf::YieldOp>(loc /*returned values*/);
    },
    /*elseBuilder=*/[&](mlir::OpBuilder& b, mlir::Location loc) {
      mlirGen(statement->v.If.orelse); // Assign_kind AST-nodes can be inside
      b.create<mlir::scf::YieldOp>(loc /*returned values*/);
    }
);
```

Как добавить условия? (3/5)

```
std::stack<std::set<llvm::StringRef>> ifElseVariables;
llvm::MallocAllocator ma; // for copying llvm::StringRef's
...
void defineVariable(llvm::StringRef name, mlir::Value value) {
  symbolTable.insert(name.copy(ma), value);
  if (!ifElseVariables.empty())
    ifElseVariables.top().insert(name.copy(ma));
}
```

Как добавить условия? (4/5)

```
/*thenBuilder=*/
[&](mlir::OpBuilder &b, mlir::Location loc) {
  ifElseVariables.push(std::set<llvm::StringRef>());
  result = mlirGen(statement->v.If.body);
  auto varsSet = ifElseVariables.top();
  llvm::SmallVector<mlir::Value> returnValues;
  for (auto it = varsSet.begin(); it != varsSet.end(); it++) {
    auto value = symbolTable.lookup(*it);
    if (value)
      returnValues.push_back(value);
  }
  ifElseVariables.pop();
  b.create<mlir::scf::YieldOp>(loc, returnValues);
},
```

Как добавить условия? (5/5)

```
if True:
    a = 5
else:
    a = 2
b = a
```

(слайд 7 –> слайд 35) == 🔥

```
root@MSI:~/phdays24/03_MLIR_gen_if_else# ./build/bin/py39compiler script.py
module {
  llvm.func @main() -> i32 {
    %0 = llvm.mlir.constant(true) : i1
    %1 = scf.if %0 -> (i64) {
      %3 = llvm.mlir.constant(5 : i64) : i64
      scf.yield %3 : i64
    } else {
      %3 = llvm.mlir.constant(2 : i64) : i64
      scf.yield %3 : i64
    }
    %2 = llvm.mlir.constant(0 : i32) : i32
    llvm.return %2 : i32
  }
}
```

Как будем генерировать IR?

В vasily-v-ryabov/phdays24 свои 5 глав:

1) Простейший IR
2) IR с переменными (с фронтендом из Python 3.9)
3) IR с условиями и выводом типов
4) **Понижаем и транслируем в LLVM IR (с опциями)**
5) Подключаем JIT движок

Как запустить «понижайку»?

1. “scf” – structured control flow
2. “cf” – control flow
3. “llvm”

- mlir-opt-18 --help | grep to-llvm
- mlir-opt-18 --convert-scf-to-cf input.mlir
- mlir-opt-18 --convert-cf-to-llvm input.mlir

Как запустить «понижайку»?

1. “scf” – structured control flow
2. “cf” – control flow
3. “llvm”
4. LLVM IR

- mlir-opt-18 --help | grep to-llvm
- mlir-opt-18 --convert-scf-to-cf input.mlir
- mlir-opt-18 --convert-cf-to-llvm input.mlir
- mlir-translate-18 --mlir-to-llvmir input.mlir

Как запустить «понижайку»?

1. “scf” – structured control flow
2. “cf” – control flow
3. “llvm”
4. LLVM IR
5. Execute on JIT engine

- mlir-opt-18 --help | grep to-llvm
- mlir-opt-18 --convert-scf-to-cf input.mlir
- mlir-opt-18 --convert-cf-to-llvm input.mlir
- mlir-translate-18 --mlir-to-llvmir input.mlir
- lli output.ll *или* mlir-cpu-runner input.mlir

Как запустить «понижайку»?

1. “scf” – structured control flow
2. “cf” – control flow
3. “llvm”
4. LLVM IR
5. Execute on JIT engine

(а если упало? а там портянка!)

- mlir-opt-18 --help | grep to-llvm
- mlir-opt-18 --convert-scf-to-cf input.mlir
- mlir-opt-18 --convert-cf-to-llvm input.mlir
- mlir-translate-18 --mlir-to-llvmir input.mlir
- lli output.ll *или* mlir-cpu-runner input.mlir
- mlir-opt-18 --mlir-pass-pipeline-crash-reproducer=<output_filepath.mlir>

Как запустить «понижайку»?

1. “scf” – structured control flow
2. “cf” – control flow
3. “llvm”
4. LLVM IR
5. Execute on JIT engine

- mlir-opt-18 --help | grep to-llvm
- mlir-opt-18 --convert-scf-to-cf input.mlir
- mlir-opt-18 --convert-cf-to-llvm input.mlir
- mlir-translate-18 --mlir-to-llvmir input.mlir
- lli output.ll *или* mlir-cpu-runner input.mlir

(а если упало? а там портянка!)

- mlir-opt-18 --mlir-pass-pipeline-crash-reproducer=<output_filepath.mlir>

(а обратно?)

- 4. LLVM IR → 3. MLIR “llvm”

- clang -S -emit-llvm foo.c >foo.ll
- mlir-translate-18 --import-llvm foo.ll >foo.mlir

Как закодить «понижайку»?

```
#include "mlir/Conversion/SCFToControlFlow/SCFToControlFlow.h"
#include "mlir/Conversion/ControlFlowToLLVM/ControlFlowToLLVM.h“
#include "mlir/Pass/Pass.h"
#include "mlir/Pass/PassManager.h"
...
int main(int argc, char **argv) {
  ...
  mlir::PassManager passes(&context);
  passes.addPass(mlir::createConvertSCFToCFPass());
  passes.addPass(mlir::createConvertControlFlowToLLVMPass());
  if (mlir::failed(passes.run(module.get())))
    return 5;
}
```

Как транслировать в LLVM IR?

```
#include "mlir/Target/LLVMIR/Dialect/Builtin/BuiltinToLLVMIRTranslation.h"
#include "mlir/Target/LLVMIR/Dialect/LLVMIR/LLVMToLLVMIRTranslation.h"
#include "mlir/Target/LLVMIR/Export.h"
...
int main(int argc, char **argv) {
  ...
  mlir::registerBuiltinDialectTranslation(*module->getContext());
  mlir::registerLLVMDialectTranslation(*module->getContext());

  llvm::LLVMContext llvmContext;
  auto llvmModule = mlir::translateModuleToLLVMIR(*module, llvmContext);
  llvm::errs() << *llvmModule << "\n"; // dump LLVM IR
}
```

Как парсить CmdLine опции?

«парсер есть, ума не надо!» (с)

```
#include "llvm/Support/CommandLine.h"
...
enum Action { DumpMLIR, DumpLLVM, DumpLLVMIR };
namespace cl = llvm::cl;
static cl::opt<enum Action> emitAction("emit", cl::desc("Select the output"),
    cl::values(clEnumValN(DumpMLIR, "mlir", "output the MLIR dump")),
    cl::values(clEnumValN(DumpLLVM, "llvm", "dump the MLIR \"llvm\" dialect")),
    cl::values(clEnumValN(DumpLLVMIR, "llvm-ir", "output the LLVM IR dump")));

static cl::opt<std::string> srcPy(cl::Positional, cl::desc("<input .py file>"),
                                  cl::init("-"), cl::value_desc("filename"));
int main(int argc, char **argv) {
  cl::ParseCommandLineOptions(argc, argv, "Python 3.9 demo compiler\n");
```


...

Как парсить CmdLine опции?

```
root@MSI:~/phdays24/04_MLIR_gen_LLVM_IR# ./build/bin/py39compiler -emit=llvm-ir script.py
; ModuleID = 'LLVMDialectModule'
source_filename = "LLVMDialectModule"

define i32 @main() {
  br i1 true, label %1, label %2

1:                                                ; preds = %0
  br label %3

2:                                                ; preds = %0
  br label %3

3:                                                ; preds = %1, %2
  %4 = phi i64 [ 2, %2 ], [ 5, %1 ]
  br label %5

5:                                                ; preds = %3
  ret i32 0
}

!llvm.module.flags = !{!0}

!0 = !{i32 2, !"Debug Info Version", i32 3}
```

Как будем генерировать IR?

В vasily-v-ryabov/phdays24 свои 5 глав:

1) Простейший IR
2) IR с переменными (с фронтендом из Python 3.9)
3) IR с условиями и выводом типов
4) Понижаем и транслируем в LLVM IR (с опциями)
5) **Подключаем JIT движок**

Как подключить JIT движок?

```
#include "mlir/ExecutionEngine/ExecutionEngine.h"
#include "mlir/ExecutionEngine/OptUtils.h"
#include "llvm/ExecutionEngine/Orc/JITTargetMachineBuilder.h"
#include "llvm/Support/TargetSelect.h"
```

```
llvm::InitializeNativeTarget();
llvm::InitializeNativeTargetAsmPrinter();
mlir::ExecutionEngineOptions engineOptions;
auto maybeEngine = mlir::ExecutionEngine::create(*module, engineOptions);
auto &engine = maybeEngine.get();
...
llvm::SmallVector<void *> argsAndReturn; // int32_t main() without argc, argv[]
int32_t exitCode; argsAndReturn.push_back(&exitCode); // address of return value
auto invocationResult = engine->invokePacked("main", argsAndReturn);
```

Что ещё есть в JIT движке?

- Детекция «горячего» кода и его «дооптимизация»
  - пример: -O0 по умолчанию, -O2 – только для «горячего» кода
- JIT callback: можно из runtime позвать обратно компилятор
  - где-то в ./llvm/unittests/ExecutionEngine/Orc/
- Поддержка отладчика для вашего языка (библиотека ORCDebugging)
- Materialization layers (используются в Mojo 🔥)
- https://www.llvm.org/docs/ORCv2.html

Чего нет в диалекте “scf”?

- “scf.break”
- “scf.continue”

- ранний return (early return)
  - “func.return”: HasParent<FuncOp> (терминатор только для “func.func”)
  - “llvm.return”: он ни во что не понизится, а надо делать “llvm.br” на финальный блок

- Можно:
  - использовать диалект “cf”: “cf.br” и “cf.cond_br” (или “llvm.br” и “llvm.cond_br”)

Чего нет в диалекте “scf”?

- “scf.break”
- “scf.continue”

- ранний return (early return)
  - “func.return”: HasParent<FuncOp> (терминатор только для “func.func”)
  - “llvm.return”: он ни во что не понизится, а надо делать “llvm.br” на финальный блок

- Можно:
  - использовать диалект “cf”: “cf.br” и “cf.cond_br” (или “llvm.br” и “llvm.cond_br”)

«бро» и «условный бро» - вот и весь «control flow»! (c)

Что есть в Mojo 🔥 ?

В Mojo 🔥 есть “hlcf”: (YouTube) 2023 LLVM Dev mtg Mojo (time=17:00)

(High-Level Control Flow)

- “hlcf.break”

```
hlcf.loop {
  %1 = lit.ref.load %i : <mut !Int, *"`i0">
  %2 = kgen.param.constant: !Int = <{value = 10}>
  %3 = kgen.call @Int::@__lt__(%1, %2)
  %4 = kgen.call @Bool::@__mlir_i1__(%3)
  hlcf.if %4 {
    hlcf.yield
  } else {
    hlcf.break
  }
```

“hlcf.continue”

```
hlcf.loop (%arg2 = %idx0 : index) {
  %0 = index.cmp slt(%arg2, %idx10)
  hlcf.if %0 {
    hlcf.yield
  } else {
    hlcf.break
  }
  %1 = kgen.call @print(%arg2)
  %2 = index.add %arg2, %idx1
  hlcf.continue %2 : index
}
```

- ранний return (early return): неизвестно, в какой форме (в языке есть)

Что есть в clangIR?

clang/include/clang/CIR/Dialect/IR/CIROps.td#L797

- “cir.break”
- “cir.continue”

- “cir.return“ (early return): clang/include/clang/CIR/Dialect/IR/CIROps.td#L540
  - ParentOneOf<["FuncOp", "ScopeOp", "IfOp", "SwitchOp", "DoWhileOp", "WhileOp", "ForOp"]

Чего нет в диалекте “func”?

- “func.invoke”
  - это вызов функции, которая может выбросить exception
  - есть “llvm.invoke” (в теории для обработки достаточно “llvm” диалекта, но много кода)
- Зато “func.func” может возвращать много значений (“llvm.func” – одно)
  - можно обойти, возвращая тип “llvm.struct” (например, “llvm.struct “A”<i64,i64,f64>”)

- Если заглянуть в операции clangIR (CIROps.td):
  - “cir.try_call” (cir::TryCallOp), “cir.try” (cir::TryOp)
  - “cir.catch” (cir::CatchOp), “cir.catch_param” (cir::CatchParamOp)
  - “cir.alloc_exception” (cir::AllocExceptionOp), “cir.throw” (cir::ThrowOp)
  - “cir.stack_save” (cir::StackSaveOp), “cir.stack_restore” (cir::StackRestoreOp)
  - (CIRTypes.td) тип “cir.eh.info”
- Для исключений нужна реализация классов (есть в диалекте “cir”)

Какие диалекты юзает Mojo 🔥 ?

- Встроенные “llvm” и “index”

- “hlcf” – High level control flow
- “mosh” – Shape диалект (shape – размеры вектора, матрицы или тензора)
- “kgen” – мета диалект, в нём **ElaborationPass**, который делает почти всё!
  - для функций с [type params] в духе: var a = func**[t1, t2]**(arg1, arg2)
  - для структур с [type params] – аналог C++ templates
  - для compile time интроспекции

Что уникального в Mojo 🔥 ?

- AST (абстрактное синтаксическое дерево) вообще не строится!
  - Из парсера исходного кода сразу генерируется MLIR
  - В MLIR есть constant folding хуки (типичная для AST оптимизация)
  - Интервью с Крисом Латтнером (май 2024)

- Стандартный модуль TargetInfo используется для loop unrolling

- JIT движок ORC (из LLVM) – на всех стадиях (!) для адаптивной компиляции
  - и just-in-time (REPL), и ahead-of-time!
  - создаются materialization layers

Что в Mojo 🔥 из LLVM?

[(c) Доклад про Mojo в 2023 (на 27:15)]

Параллелизм 80 левел (на 27:40):

- Один LLVMContext на функцию

LLVM is good for:

- GVN, Load/Store Optimization, LSR, etc
- scalar optimization (e.g. instcombine)
- target-specific code generation

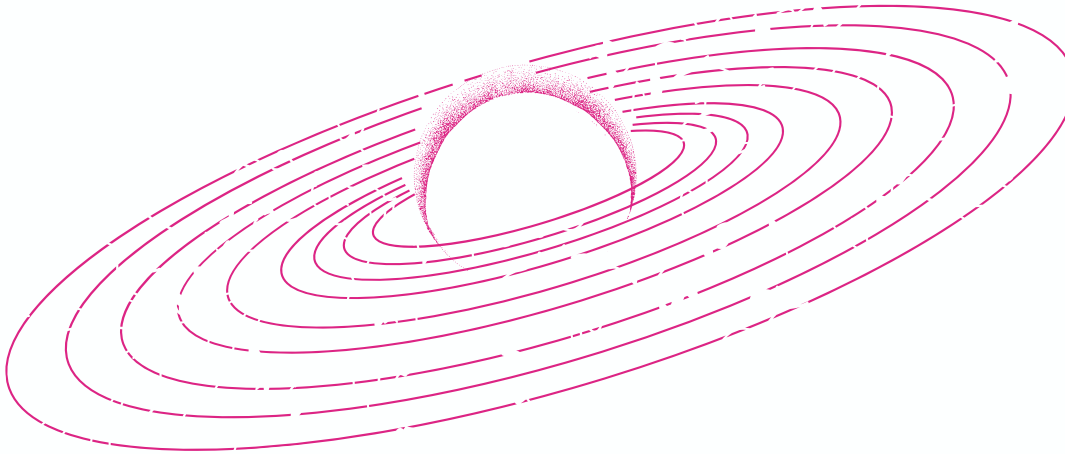
We need to disable:

- Vectorizer, loop unroller, etc
- Inliner and other IPO passes

Solution: replace these!

- Build new MLIR passes
- Replace others with Mojo libraries

Почему рано учить Mojo 🔥 ?



Язык постоянно меняется:

- let переменные убрали недавно
- динамич. типы в runtime – на ре-дизайне

Не открыты MLIR диалекты:

- “kgen”, “hlcf”, … (не зрелые?)

Приоритеты (как кажется сейчас):

- P1: ИИ движок
- P2: MLIR frontend (язык для компиляторщиков: юзать диалекты)
- P3: Python superset (superset++, т.е. сильно шире, чем Python)

Вывод: изучайте MLIR и JIT движок ORC => быстро понять Mojo 🔥

Что ещё посмотреть?

phd 2 x pt

← плейлист

На русском:

- Лекция Константина Владимирова про LLVM IR (2019) – GEP, load, alloca

На английском:

- Extending Dominance to MLIR Regions (2023) – дерево доминации в MLIR
- MLIR Dialect Design (EuroLLVM 2023) – классификация диалектов
- What's new in MLIR? (2023 vs 2019)
- JITLink: Native Windows JITing in LLVM и ORCv2 - LLVM JIT APIs Deep Dive

- www.youtube.com/@LLVMPROJ – LLVM Dev meetings & EuroLLVM

Оценка/отзыв о докладе:

@vasily_v_ryabov

vasily.v.ryabov@gmail.com

github.com/vasily-v-ryabov

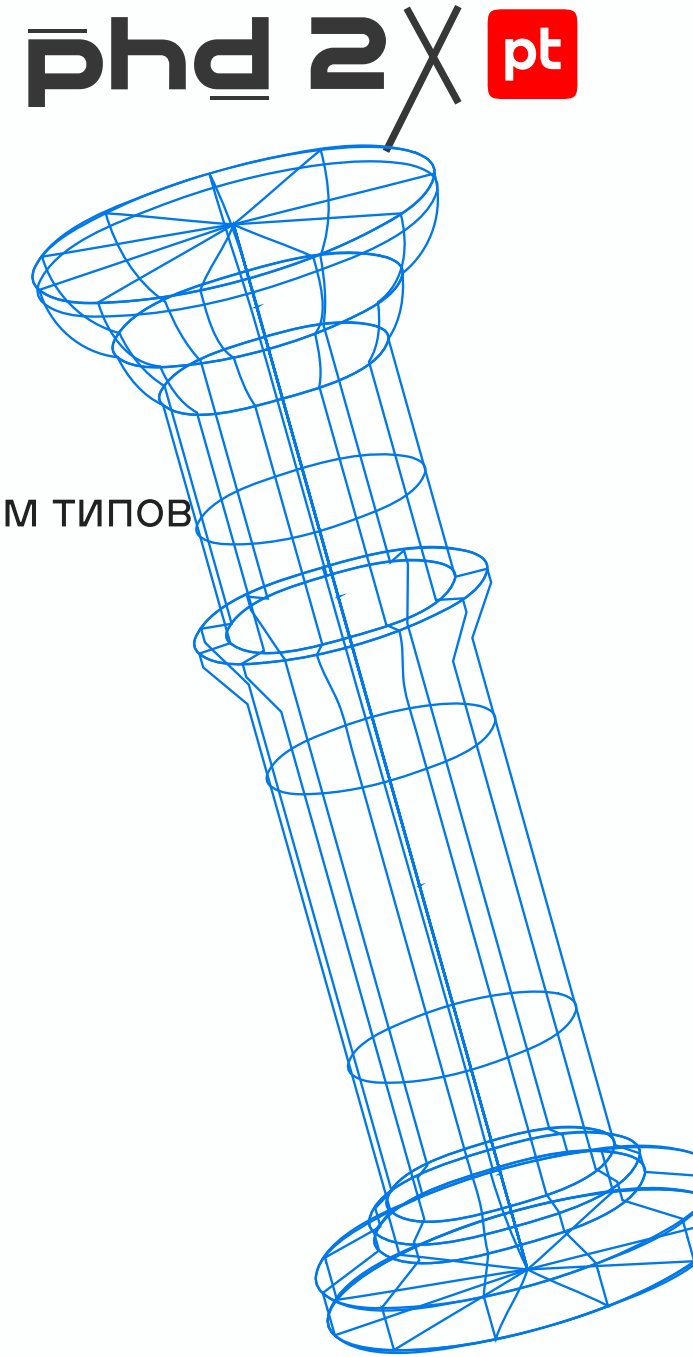
@vasily_v_ryabov

Спасибо!

OK, собираем MLIR

```
git clone --depth 1 -b llvmorg-18.1.5 https://github.com/llvm/llvm-project.git

cd ./llvm-project/ && mkdir build && cd build/

pip install -U cmake ninja

cmake -G Ninja ../llvm/ -DLLVM_ENABLE_PROJECTS="mlir" -DLLVM_ENABLE_ASSERTIONS=ON -DCMAKE_BUILD_TYPE=Debug -DLLVM_TARGETS_TO_BUILD="Native"

cmake --build . --target check-mlir      # (засекаем час, пьём чай)

# готово!


# на Windows (до сборки!) нужно запустить regedit, пойти в папку:

# HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\FileSystem\

# выставить LongPathsEnabled = 1 и перезагрузиться!

# вместо -G Ninja берём, например, -G "Visual Studio 17 2022"
```

Что мне кажется сложным?



Middle end

Вывод типов – алгоритмически неразрешим для многих систем типов

(в датасете ManyType4Py выводится до 60-70% типов)

Дизайн диалектов – искусство (ну, и TableGen и C++)