MLIR

Multi-Level IR Compiler Framework

MLIR & python binding 简介

张洪滨

2020.02.05

什么是MLIR？

不是Machine Learning，但为Machine learning而生

MLIR (Multi-Level Intermediate Representation)

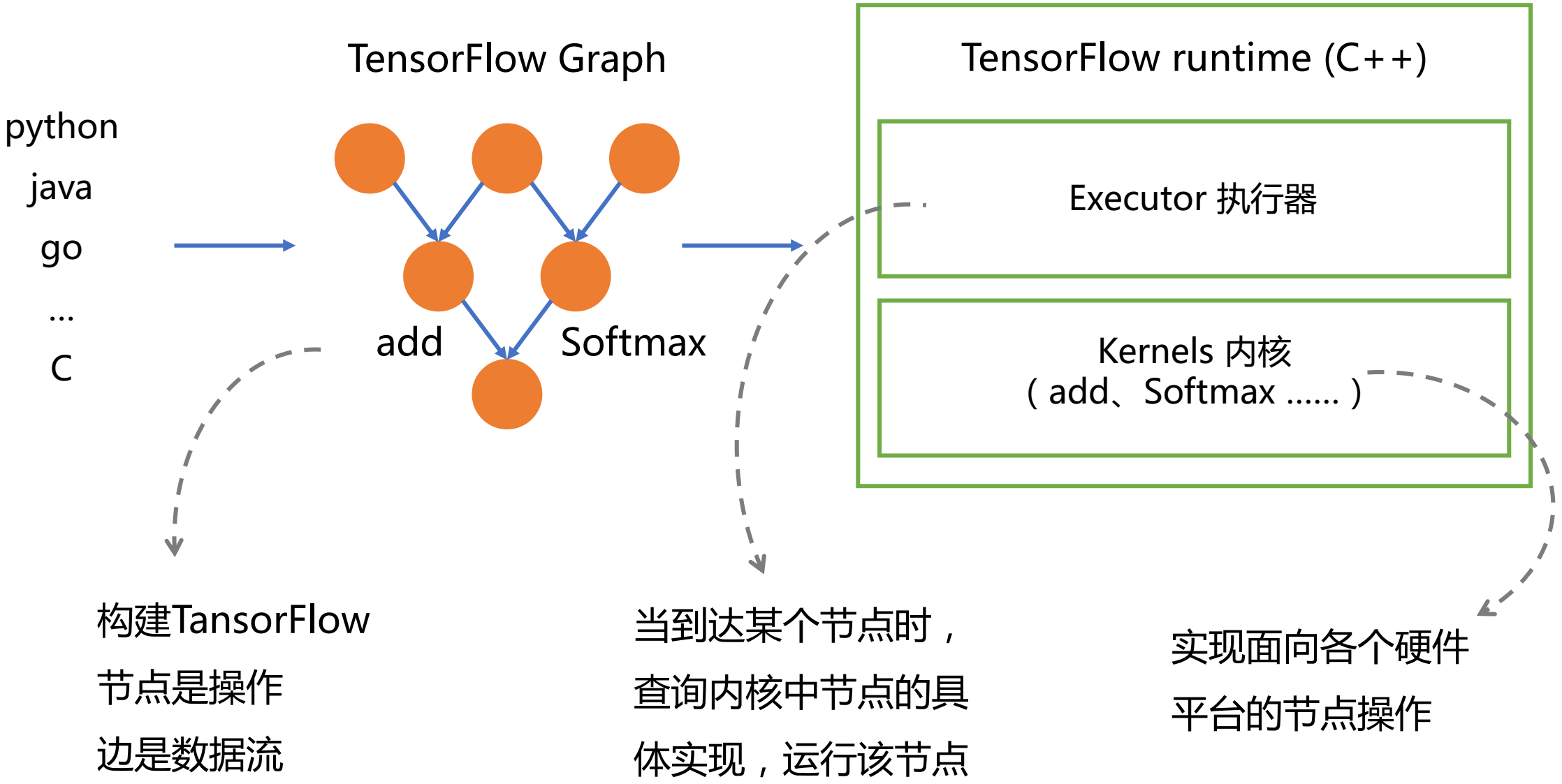
The MLIR project is a novel approach to building reusable and extensible compiler infrastructure. MLIR aims to address software fragmentation, improve compilation for heterogeneous hardware, significantly reduce the cost of building domain specific compilers, and aid in connecting existing compilers together.

一个可重用、可扩展的开源代码优化框架

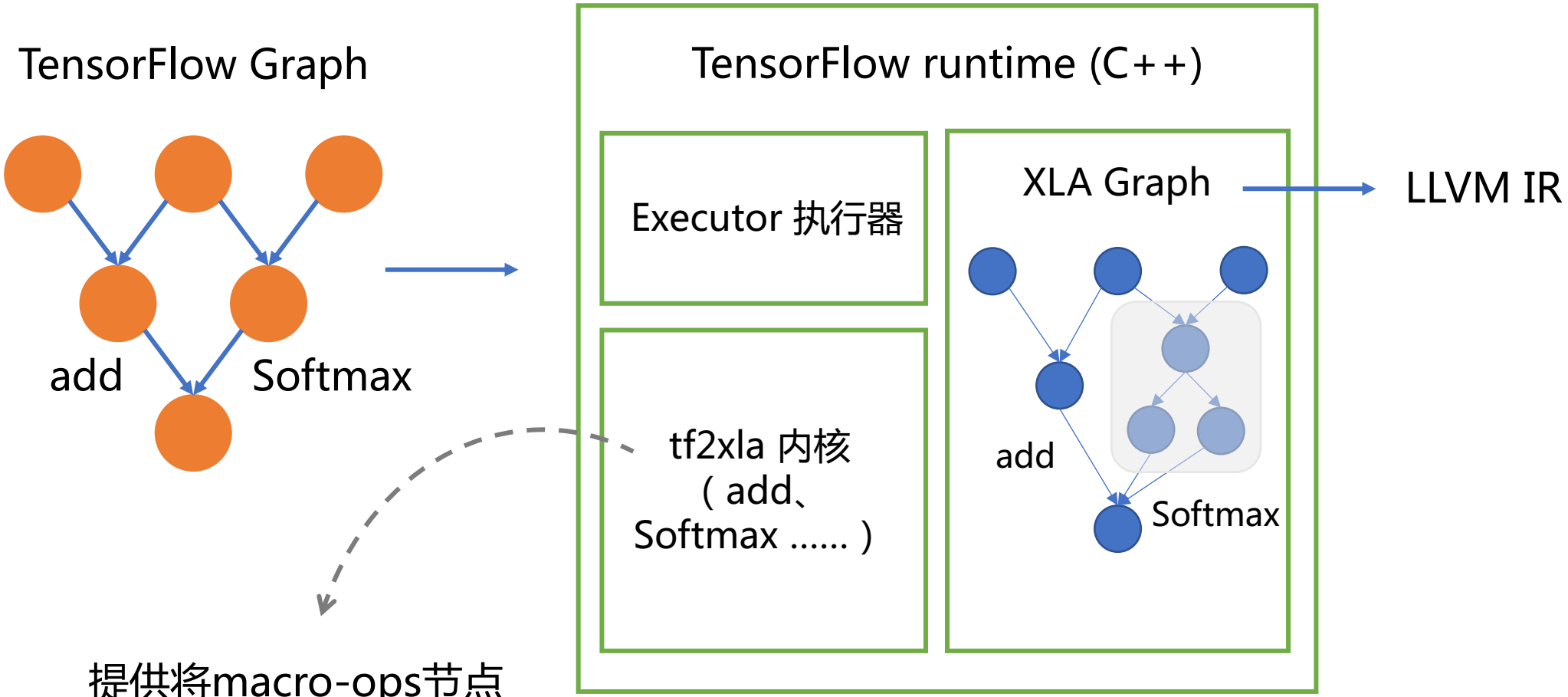
- 处理软件的碎片化
- 为面向多种硬件的编译提供支持
- 为领域专用编译器的开发减少开销
- 连接已有的各种编译器

→ 逐级抽象 代码优化 避免组合爆炸

TensorFlow执行方式

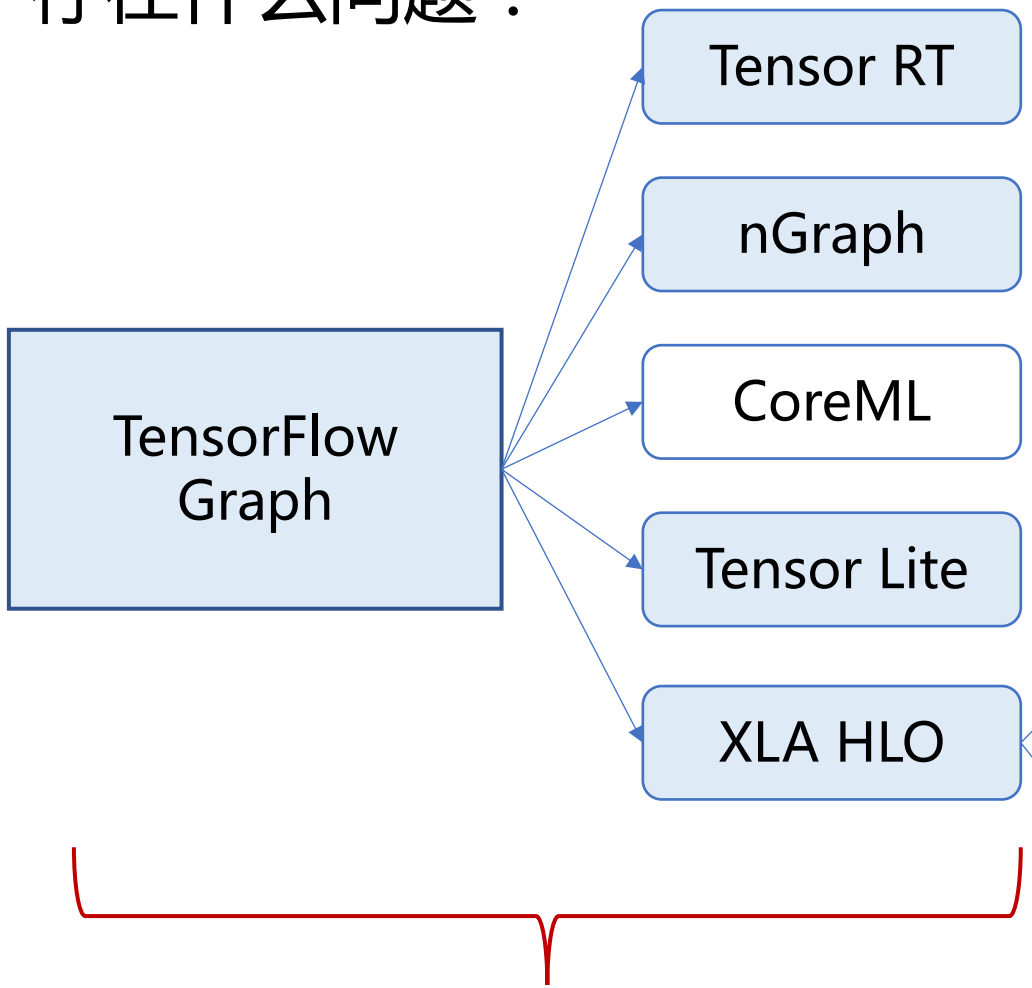


使用XLA加速TensorFlow



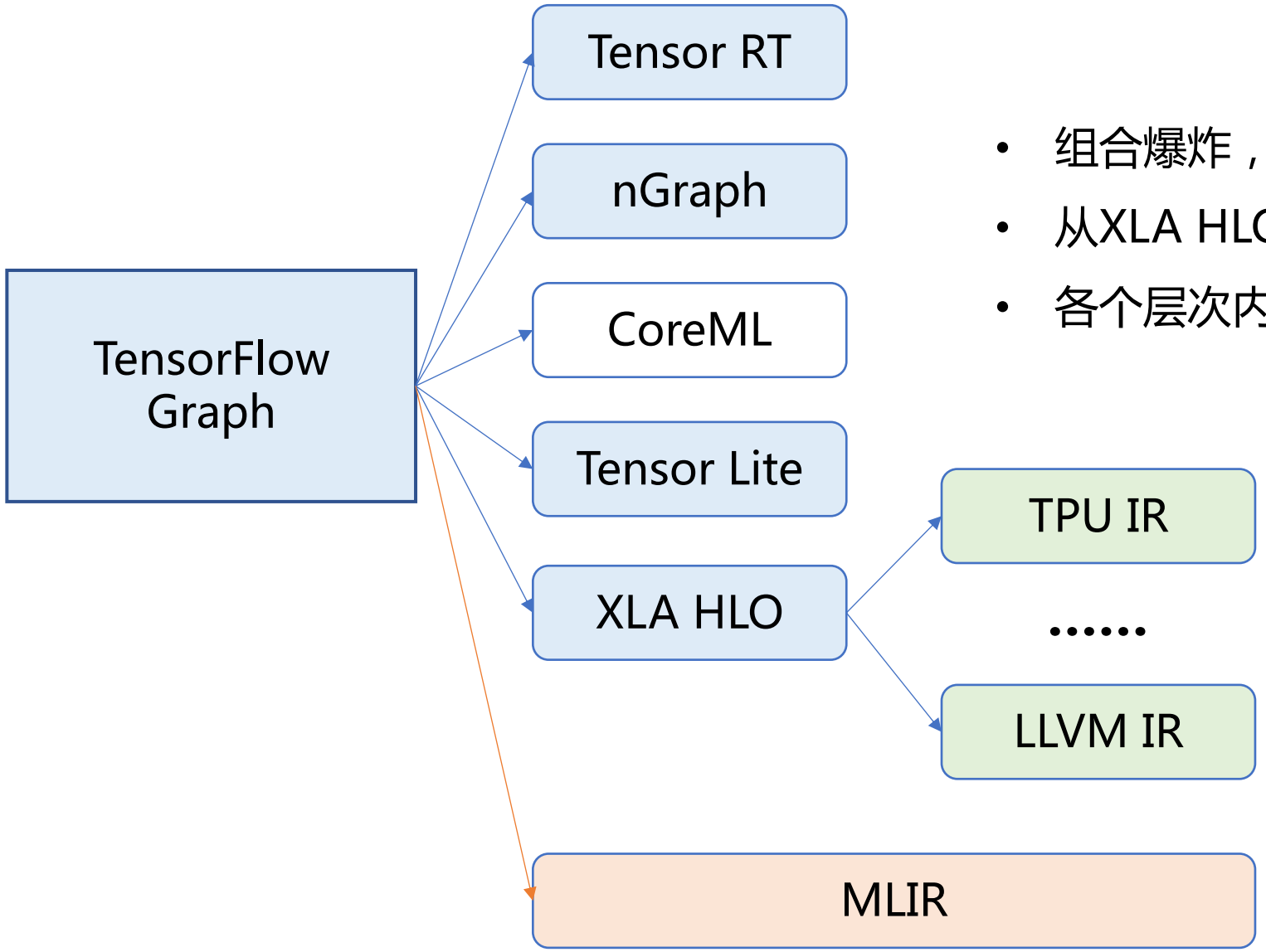
提供将macro-ops节点
转换为一系列初级节点
的组合

存在什么问题？



- 组合爆炸，很多组件无法重用
- 从XLA HLO到LLVM IR跨度太大，实现开销大
- 各个层次内部优化无法迁移

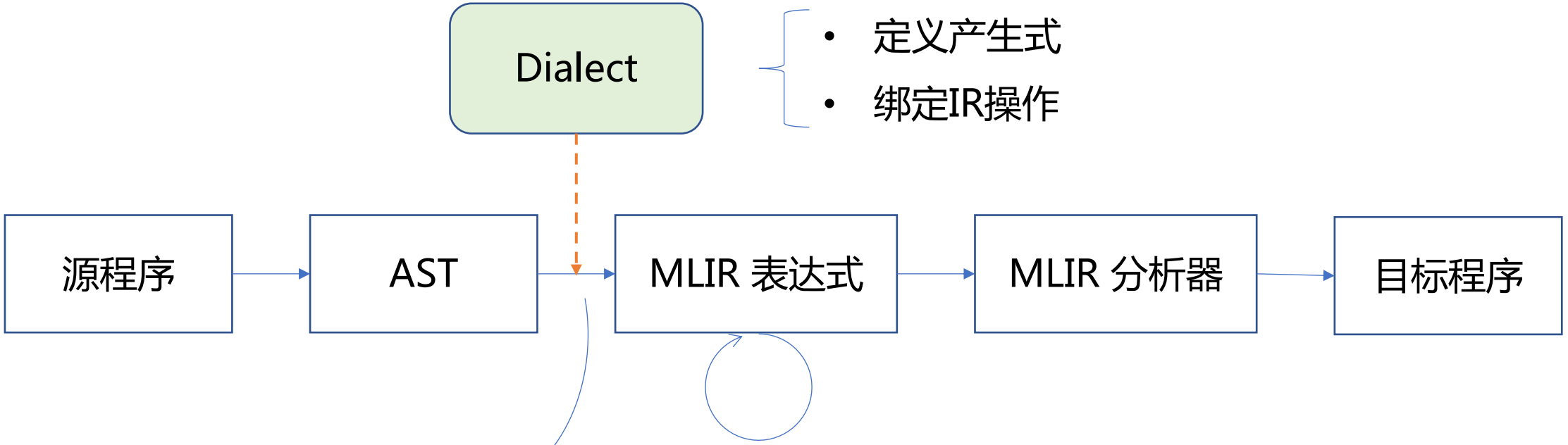
MLIR如何解决？



- 组合爆炸，很多组件无法重用
- 从XLA HLO到LLVM IR跨度太大，实现开销大
- 各个层次内部优化无法迁移

- 统一IR，重用组件，避免组合爆炸
- 采用多级抽象， Multi-Level的真意
- 各个层次进行协同优化

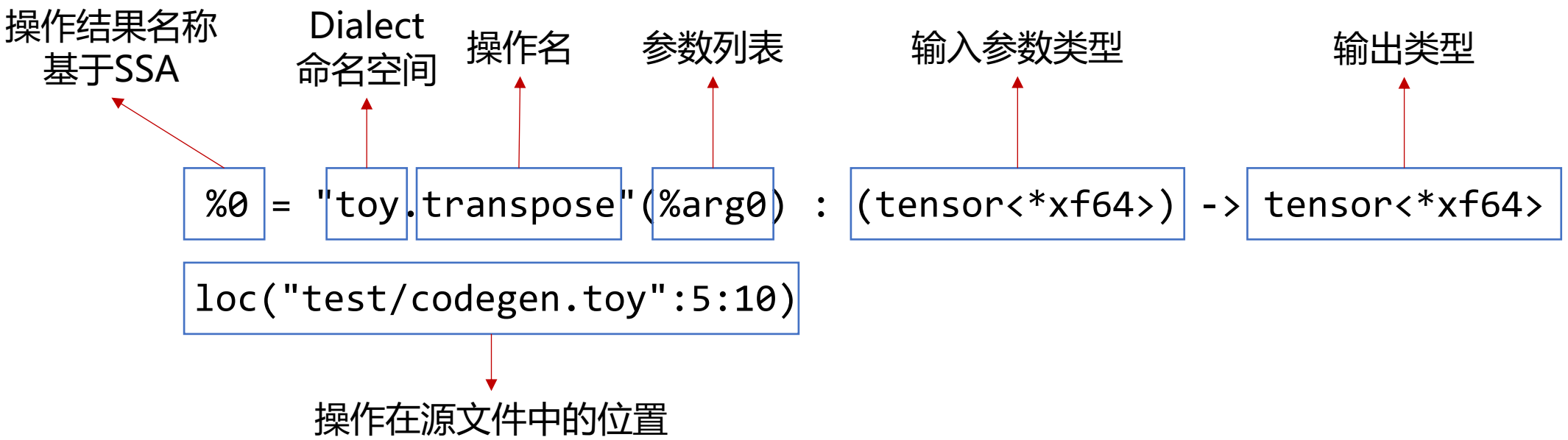
MLIR的法宝 -- Dialect

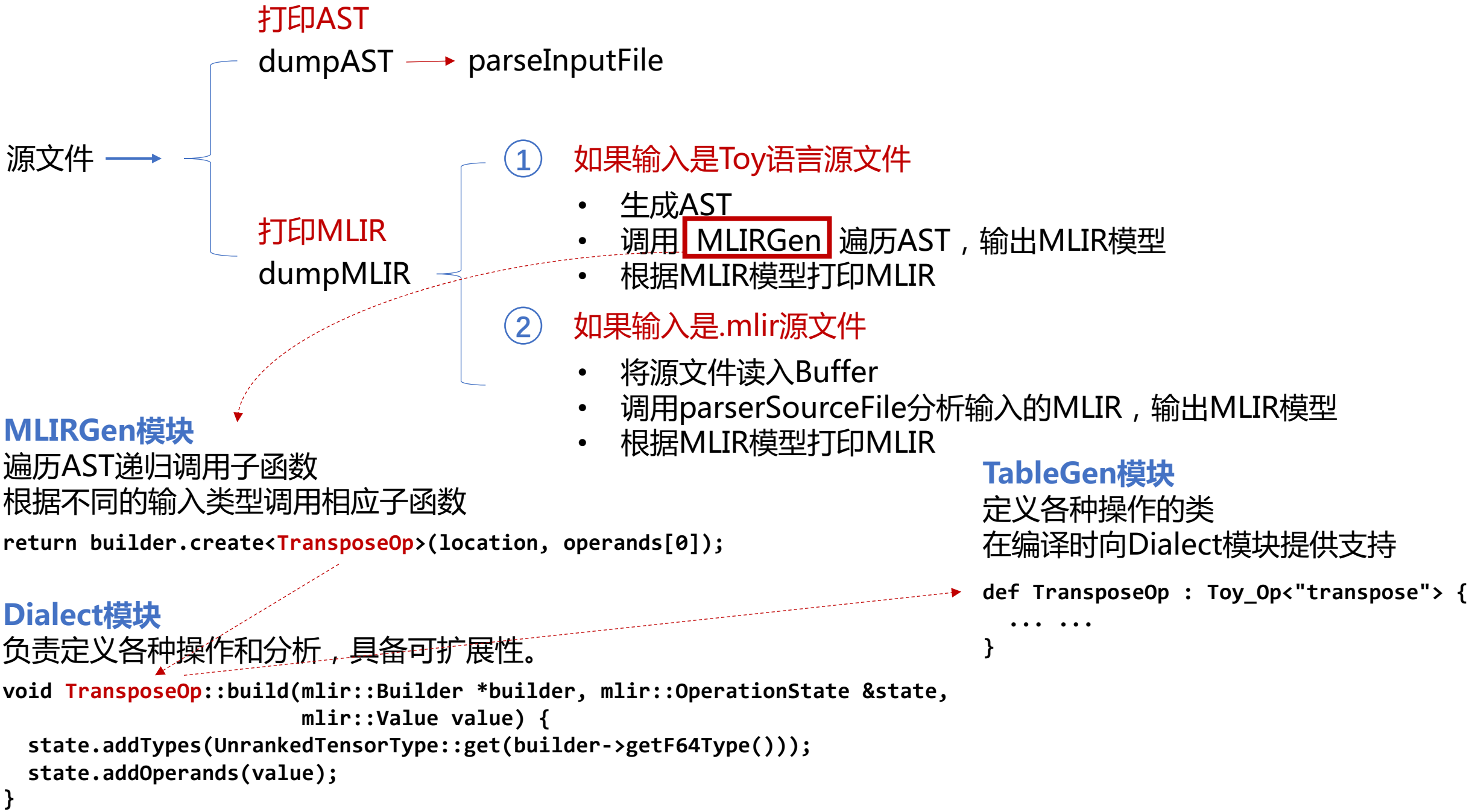


MLIR的惊鸿一瞥

```
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);
  var d = multiply_transpose(b, a);
  print(d);
}
```




打印AST
dumpAST
parseInputFile
源文件
打印MLIR
dumpMLIR
① 如果输入是Toy语言源文件
• 生成AST
• 调用 MLIRGen 遍历AST，输出MLIR模型
• 根据MLIR模型打印MLIR
② 如果输入是.mlir源文件
• 将源文件读入Buffer
• 调用parserSourceFile分析输入的MLIR，输出MLIR模型
• 根据MLIR模型打印MLIR
MLIRGen模块
遍历AST递归调用子函数
根据不同的输入类型调用相应子函数
return builder.create<TransposeOp>(location, operands[0]);
TableGen模块
定义各种操作的类
在编译时向Dialect模块提供支持
def TransposeOp : Toy_Op<"transpose"> {
... ...
}
Dialect模块
负责定义各种操作和分析，具备可扩展性。
void TransposeOp::build(mlir::Builder *builder, mlir::OperationState &state,
mlir::Value value) {
state.addTypes(UnrankedTensorType::get(builder->getF64Type()));
state.addOperands(value);
}

TableGen模块

- 开发的单源性，避免冗余开发
- 促进自动化生成，减少Operation手动开发

① 定义一个和 Toy Dialect 的链接

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

② 创建 Toy Dialect Operation 的基类

```
class Toy_Op<string mnemonic, list<OpTrait> traits = []> :
    Op<Toy_Dialect, mnemonic, traits>;
```

③ 创建 Toy Dialect 各种 Operation

```
def TransposeOp : Toy_Op<"transpose"> {
  let summary = "transpose operation";
  let arguments = (ins F64Tensor:$input);
  let results = (outs F64Tensor); let builders = [
    OpBuilder<"Builder *b, OperationState &state, Value input">
  ];
  let verifier = [{ return ::verify(*this); }];
}
```

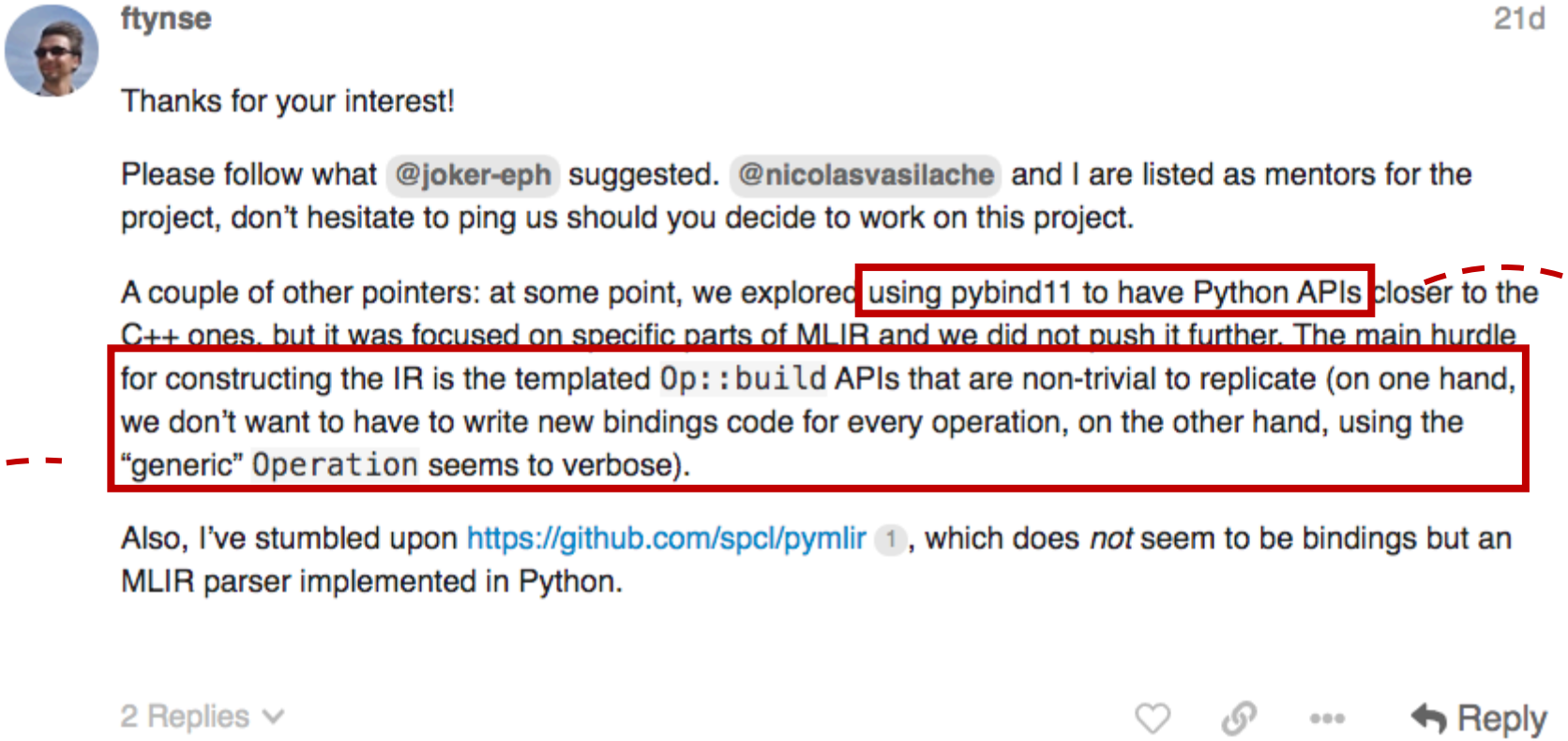
mlir-tblgen
工具

→ C++文件

Google Summer of Code 2020 – MLIR python bindings

- bugpoint/llvm-reduce and llvm-canon kind of tools for MLIR (mentor: Mehdi Amini, Jacques Pienaar)
- Rework the MLIR python bindings, add a C APIs for core concepts (mentor: Nicolas Vasilache, Alex Zinenko)
- Automatic formatter for TableGen (similar to clang-format for C/C++)
- LLVM IR declaratively defined. (mentor: Alex Zinenko)
- MLIR Binary serialization / bitcode format (Mehdi Amini)
- SPIR-V module combiner
    - Basic: merging modules and remove identical functions
    - Advanced: comparing logic and use features like spec constant to reduce similar but not identical functions
- GLSL to SPIR-V dialect frontend
    - Requires: building up graphics side of the SPIR-V dialect
    - Purpose: give MLIR more frontends :) improve graphics tooling
    - Potential real-world usage: providing a migration solution from WebGL (shaders represented as GLSL) to WebGPU (shaders represented as SPIR-V)
- TableGen “front-end dialect” (mentor: Jacques Pienaar)
- Making MLIR interact with existing polyhedral tools: isl, pluto (mentor: Alex Zinenko)
- MLIR visualization (mentor: Jacques Pienaar)

Google Summer of Code 2020 – MLIR python bindings



ftynse 21d

Thanks for your interest!

Please follow what @joker-eph suggested. @nicolasvasilache and I are listed as mentors for the project, don't hesitate to ping us should you decide to work on this project.

A couple of other pointers: at some point, we explored using pybind11 to have Python APIs closer to the C++ ones, but it was focused on specific parts of MLIR and we did not push it further. The main hurdle for constructing the IR is the templated `Op::build` APIs that are non-trivial to replicate (on one hand, we don't want to have to write new bindings code for every operation, on the other hand, using the "generic" `Operation` seems to verbose).

Also, I've stumbled upon https://github.com/spcl/pymlir 1, which does *not* seem to be bindings but an MLIR parser implemented in Python.

2 Replies Reply

目前的困难在于对Op::build进行python binding，存在过多的重复操作。

使用pybind11作为python binding的工具

MLIR python binding 简易模型

MLIRGen模块

输入整型a，输出结果result
调用builder的create方法产生结果result

```
void MLIRGen::generate(int a) {
  int result = builder.create<TransposeOp>(a);
  cout << "result: " << result << endl;
}
```

Python binding模块

绑定MLIRGen模块
使用python调用C++实现的方法

```
PYBIND11_MODULE(MLIR, m) {
  py::class_<MLIRGen>(m, "MLIRGen")
    .def(py::init())
    .def("generate", &MLIRGen::generate);
}
```

```
Python 3.7.4
>>> import MLIR
>>> gen = MLIR.MLIRGen()
>>> gen.generate(1)
TransposeOp: 1
result: 1
```

Builder模块

构建MLIR表达式
模版函数create调用泛型的方法build

```
template <typename OpTy>
int create(int a) {
  OpTy::build(a);
  return a;
}
```

Dialect模块

定义各Operation的类以及build方法。
每个build方法打印出Operation名字以及传入的整型值。

```
void TransposeOp::build(int a) {
  cout << "TransposeOp: " << a << endl;
}

void ConstantOp::build(int a) {
  cout << "ConstantOp: " << a << endl;
}
```

Google Summer of Code 2020 – MLIR python bindings

zhanghb97

ftynse 19d

Hi @ftynse,

After having quick learning about Toy tutorial and pybind11, I come to understand the main hurdle for constructing the IR. In my opinion, when we define a Dialect, there are lots of corresponding operations. And for each operation, we should implement the templated `Op::build` APIs. As for the python bindings file, we should create a binding for each `Op::build` API in the `PYBIND11_MODULE`, which causes duplication of work.

Is my understanding correct? If I catch the point, how could I learn more about it? And I can't find MLIR python bindings examples in the llvm-project, could I have a demo to try out the python bindings?

Thanks!

Reply

- 定义Dialect时，要定义一系列的Operation
- 对于每个Operation，都要在Dialect里面定义build函数
- 每一个build函数都要在PYBIND11_MODULE里面进行绑定

Google Summer of Code 2020 – MLIR python bindings

Dialect.cpp中定义各个Operation的build

```
void TransposeOp::build(……) {……}
```

Builder.h中通过函数模版定义create，调用build函数

```
template <typename OpTy, typename... Args>
OpTy create(Location location, Args &&... args)
{……}
```

MLIRGen.cpp中传递泛型参数选择使用哪个Operation类中的build函数

```
return builder.create<TransposeOp>(…)
```


ftynse 19d

Yes, your evaluation goes in the right direction. `Op::build` methods are different for all operations and, furthermore, they are called indirectly through `OpBuilder::create` function template. Users are not expected to call `*Op::build` APIs themselves. In C++, we rely on templates to forward arguments from `OpBuilder::create` to the relevant `Op::build` function, but it is unclear how this can be achieved in Python.

Also, many of the Ops are generated from ODS (https://mlir.llvm.org/docs/OpDefinitions/), which we could try and use for generating Python bindings as well.

> And I can't find MLIR python bindings examples in the llvm-project, could I have a demo to try out the python bindings?

We only *explored* it, there is no publicly available code for the bindings, hence the open project. 😉

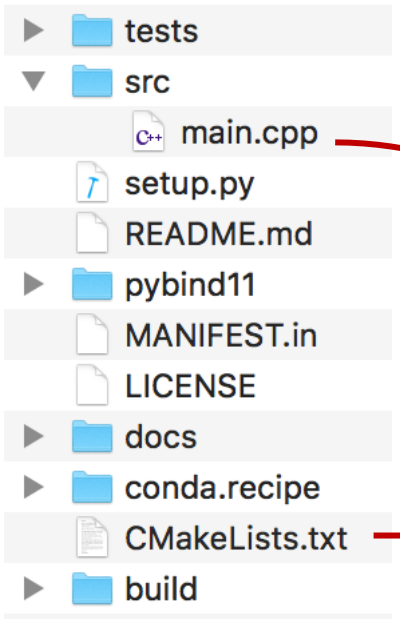
Please consider also looking at different ways of exposing the bindings as `@joker-eph` mentioned

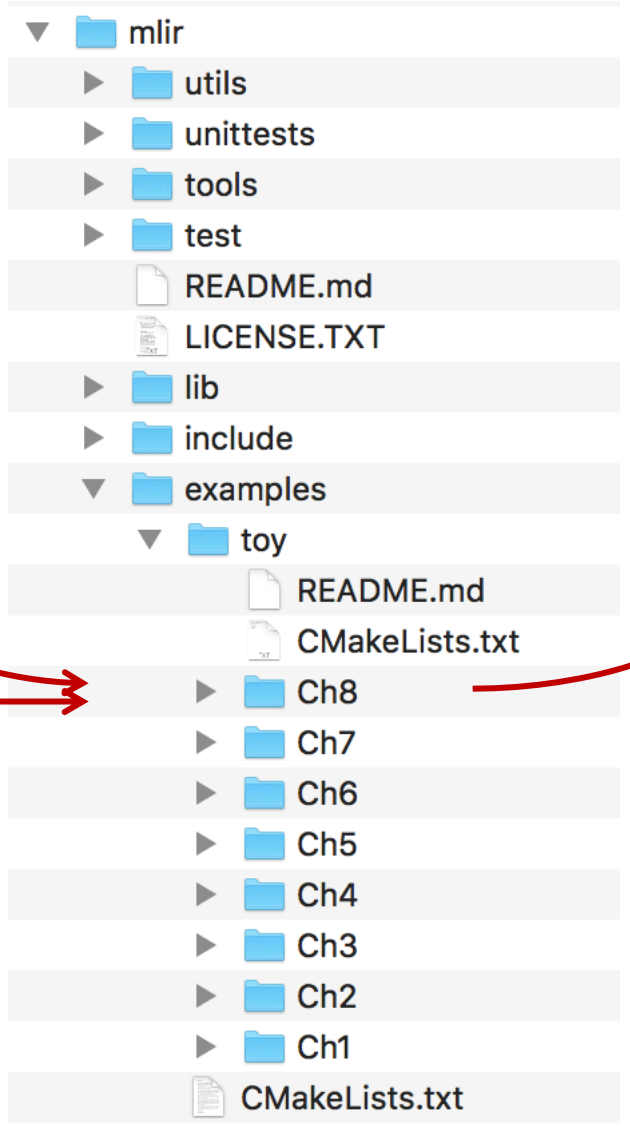
> LLVM is exposing a C API that is then wrapped in python using ctypes. But there are other possibilities, for example LLDB is using swig. More recent approaches include clif 1 and pybind11.

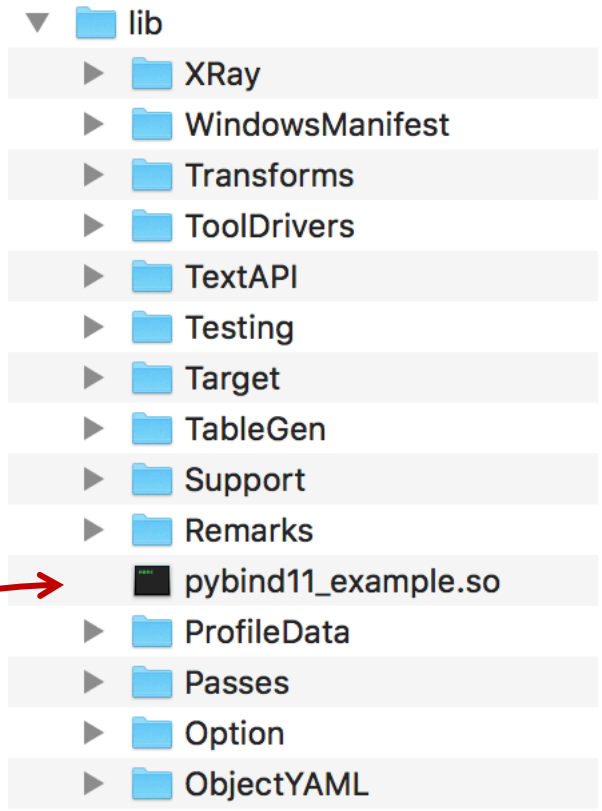
we experimented with pybind11 because that was the one we knew best.

python对这样的结构怎么进行实现和绑定？

将pybind11的示例程序嵌入到MLIR路径中







将示例程序中的main.cpp和CMakeLists.txt放入MLIR example的Ch8路径下

用LLVM的Cmake结构，把共享库文件编译到build/lib路径下，可以使用python的命令行进行测试

```
>>> import pybind11_example
>>> pybind11_example.add(1,2)
3
```

cmake命令，选择构建的target

```
cmake --build . --target check-mlir
```

给check-mlir增加依赖

llvm-project/mlir/test/CMakeLists.txt

```
......
add_lit_testsuite(check-mlir "Running the MLIR regression tests"
  ${CMAKE_CURRENT_BINARY_DIR}
  DEPENDS ${MLIR_TEST_DEPENDS}
  )
......
```

定义&增加 MLIR_TEST_DEPENDS

llvm-project/mlir/test/CMakeLists.txt

```
set(MLIR_TEST_DEPENDS
  ......
)

if(LLVM_BUILD_EXAMPLES)
  list(APPEND MLIR_TEST_DEPENDS
    ......
    pybind11_example
    )
endif()
```

定义pybind11共享库

llvm-project/mlir/examples/toy/Ch8/CMakeLists.txt

```
find_package(pybind11 REQUIRED)
pybind11_add_module(pybind11_example main.cpp)
```

将MLIR_TEST_DEPENDS 添加到check-mlir的依赖中

llvm-project/llvm/cmake/modules/AddLLVM.cmake

```
function(add_lit_testsuite target comment)
  ......
  # Produce a specific suffixed check rule.
  add_lit_target(${target} ${comment}
    ${ARG_UNPARSED_ARGUMENTS}
    PARAMS ${ARG_PARAMS}
    DEPENDS ${ARG_DEPENDS}
    ARGS ${ARG_ARGS}
    )
endfunction()
```

```
function(add_lit_target target comment)
  ......
  if (ARG_DEPENDS)
    add_dependencies(${target} ${ARG_DEPENDS})
  endif()
  ......
endfunction()
```

未来工作

- 完成MLIR Toy的学习路线
- 调研各种python binding，比较哪种更适合项目
- 学习Operation Definition Specification (ODS)框架

相关文章

- 初见MLIR：https://zhuanlan.zhihu.com/p/101879367
- MLIR的法宝：Dialects：https://zhuanlan.zhihu.com/p/102212806
- MLIR的惊鸿一瞥：https://zhuanlan.zhihu.com/p/102395938
- MLIR的生产线--Dialects和他的小伙伴们：https://zhuanlan.zhihu.com/p/102565792
- MLIR Dialect的零件生产者 – TableGen：https://zhuanlan.zhihu.com/p/102727417
- MLIR 开放项目 -- python bindings：https://zhuanlan.zhihu.com/p/102934213
- MLIR python bindings的问题&Dialect Operation build方法：https://zhuanlan.zhihu.com/p/103102332
- MLIR python binding 简易模型建立：https://zhuanlan.zhihu.com/p/103524807
- MLIR python binding -- pybind11：https://zhuanlan.zhihu.com/p/103836518
- 将pybind11示例嵌入到MLIR中：https://zhuanlan.zhihu.com/p/104717000
- 基于CMake构建系统的MLIR Example扩展：https://zhuanlan.zhihu.com/p/104948867