编译论坛

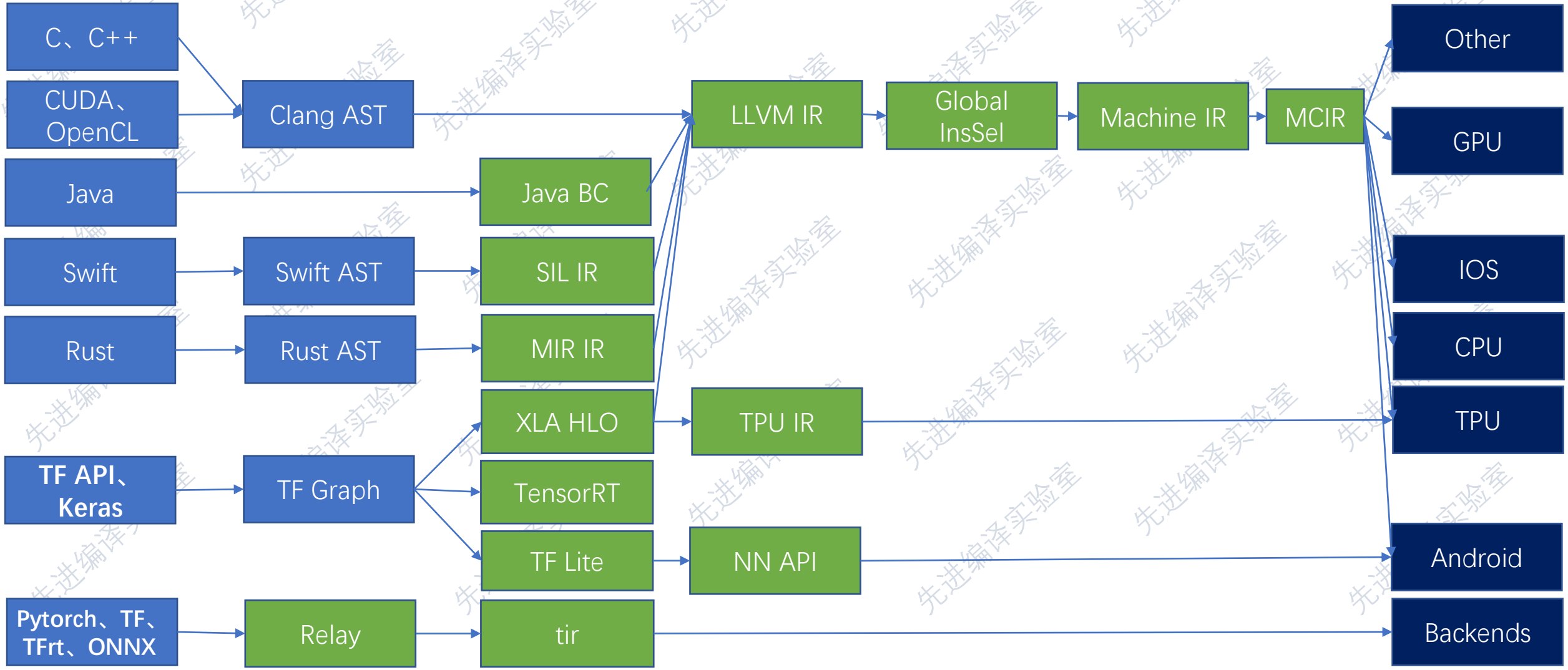
MLIR

嘉宾：桂中华

目录

常见的IR表示系统

Clang 对AST进行静态分析和转换操作，各个语言的AST都需要进行类似的优化转换成对应的IR



IR转换的问题

1. IR种类太多，针对不同种类IR开发的Pass可能重复，即不同IR的同类Pass不兼容，针对新的IR编写同类Pass需要重新学习IR语法门槛过高

2. 不同类型IR所做的Pass优化在下一层中不可见

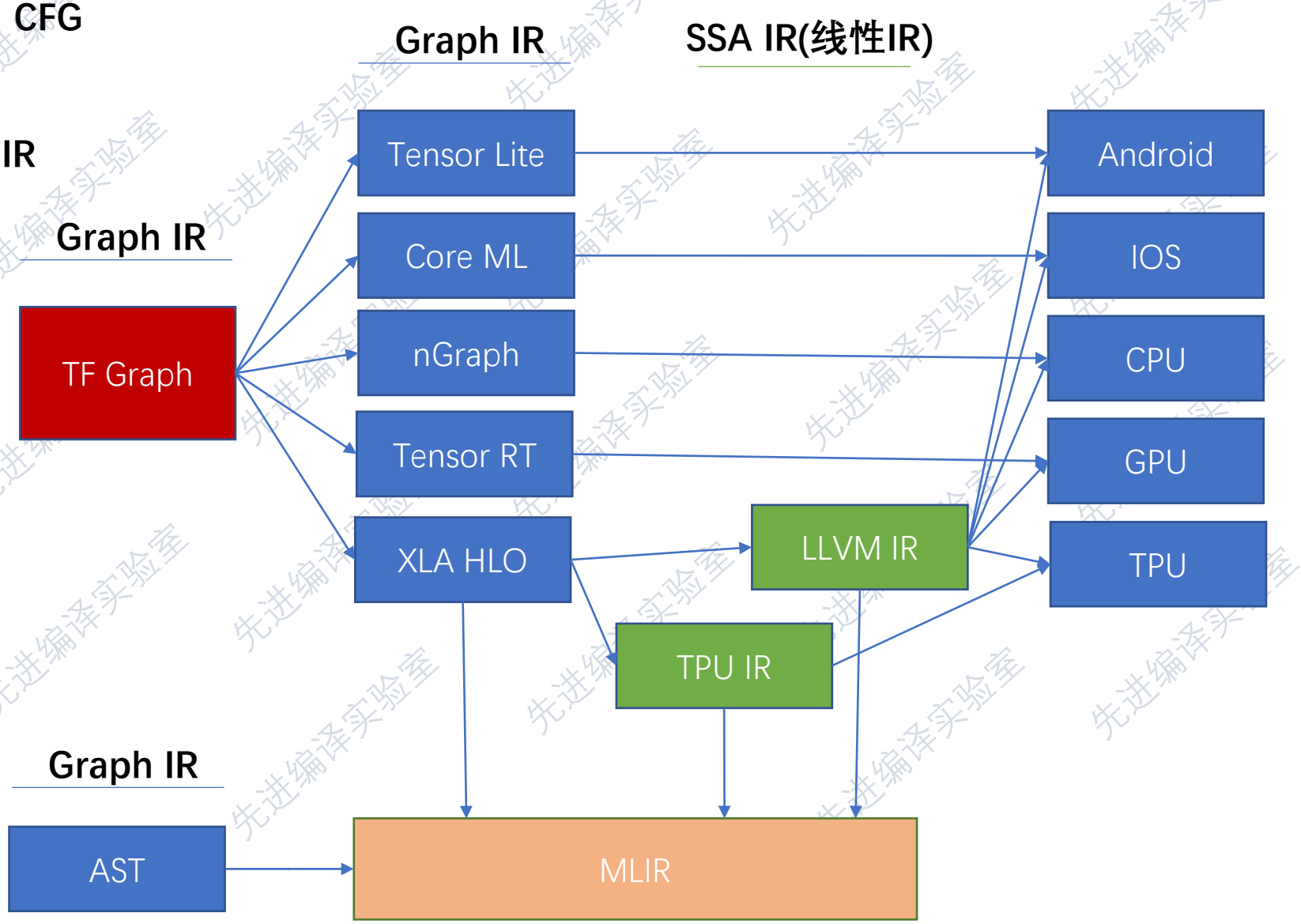
3. 不同类型IR间转换开销大，从图IR到LLVM IR直接转换存在较大开销，

MLIR简介-常见的IR种类

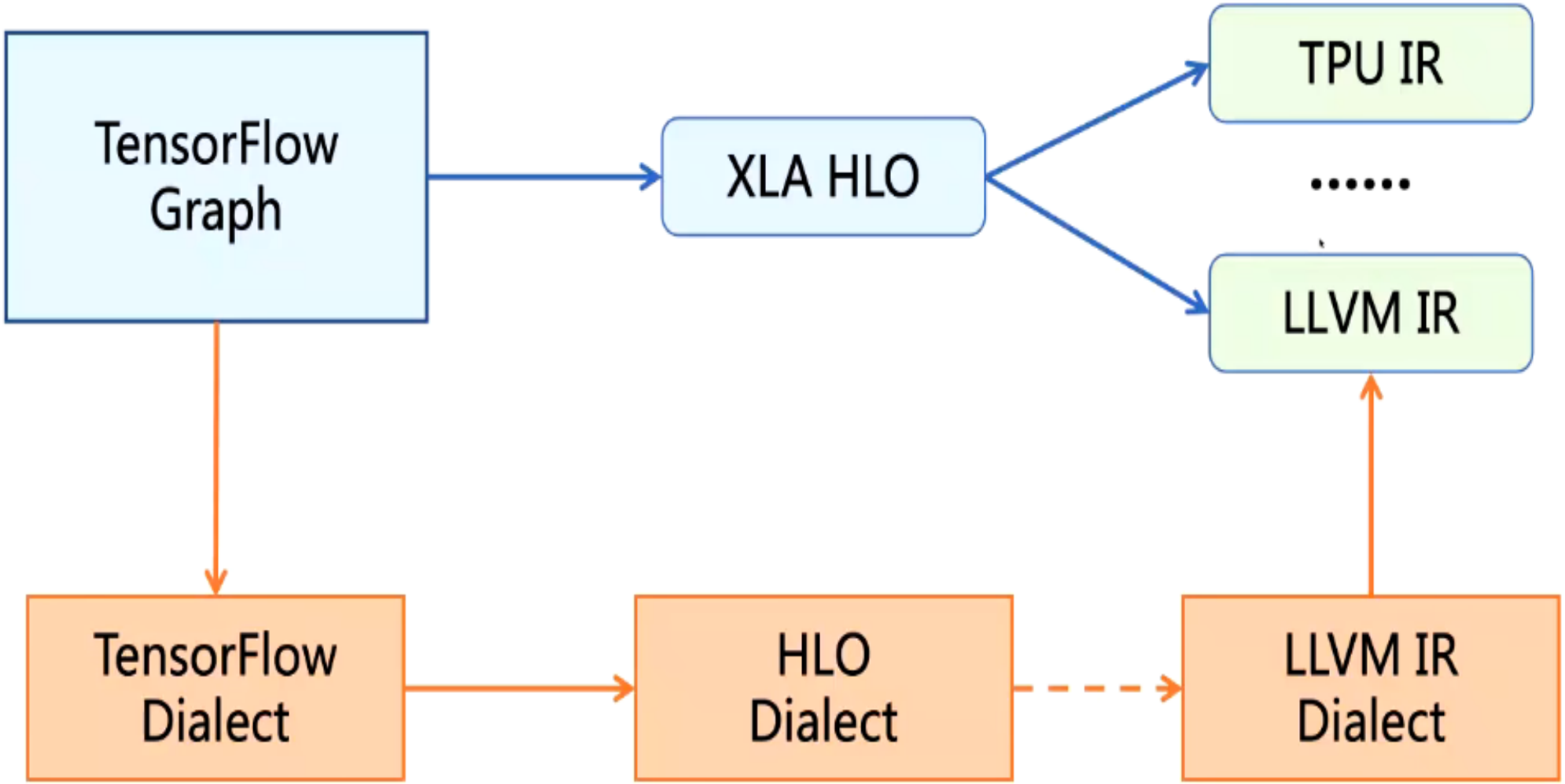
Graph IR:AST、DAG、CFG

线性　IR:三地址IR

图线性混合IR：LLVM IR



MLIR简介-使用Dialect构建IR表示系统



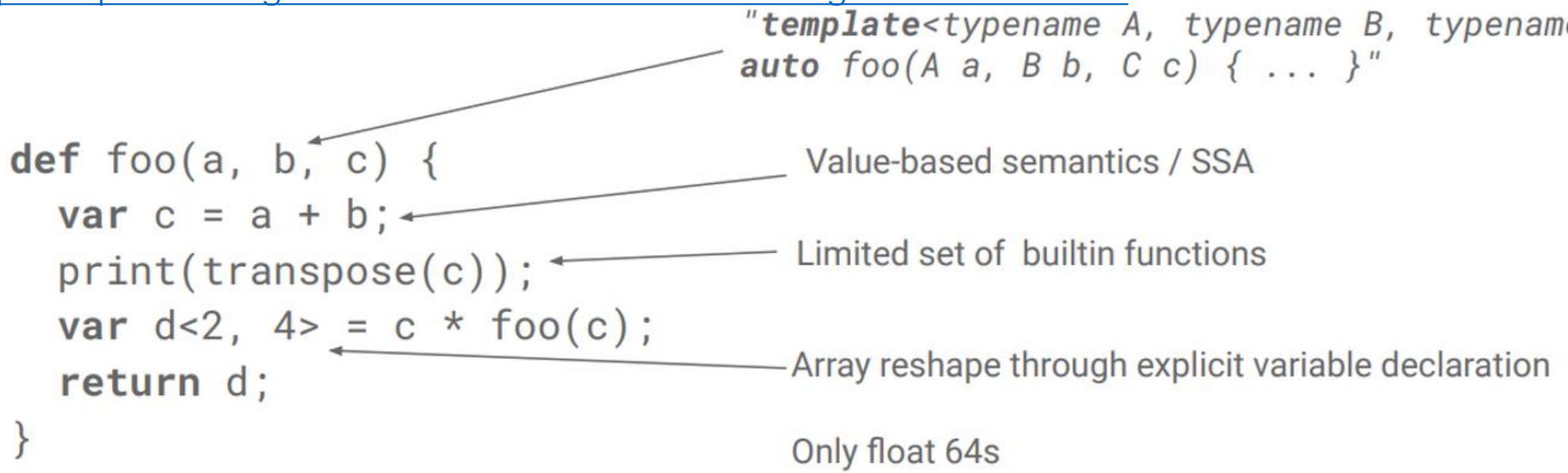
Toy接入MLIR

Let's Build a Toy Language

Toy语言，为了验证及演示MLIR系统的整个流程而开发的一种基于Tensor的语言，常见transpose和print两个'函数'

- Mix of scalar and array computations, as well as I/O
- Array shape Inference
- Generic functions
- Very limited set of operators and features (it's just a Toy language!)

[Kaleidoscope: Kaleidoscope Introduction and the Lexer — LLVM 15.0.0git documentation]
[Kaleidoscope: Implementing a Parser and AST — LLVM 15.0.0git documentation]



Toy源码和AST

```
278   //===----------------------------------------------------------------------===//
279   // TransposeOp
280   //===----------------------------------------------------------------------===//
281
282   void TransposeOp::build(mlir::OpBuilder &builder, mlir::OperationState &state,
283                           mlir::Value value) {
284     state.addTypes(UnrankedTensorType::get(builder.getF64Type()));
285     state.addOperands(value);
286   }
```

```
293     data.push_back(cast<NumberExprAST>(expr).getValue());
294   }
295
296   /// Emit a call expression. It emits specific operations for the `transpose`
297   /// builtin. Other identifiers are assumed to be user-defined functions.
298   mlir::Value mlirGen(CallExprAST &call) {
299     llvm::StringRef callee = call.getCallee();
300     auto location = loc(call.loc());
301
302     // Codegen the operands first.
303     SmallVector<mlir::Value, 4> operands;
304     for (auto &expr : call.getArgs()) {
305       auto arg = mlirGen(*expr);
```

```
:FuncOp mlirGen(PrototypeAST &proto) {
:FuncOp mlirGen(FunctionAST &funcAST) {
mlir::Value mlirGen(BinaryExprAST &binop) {
mlir::Value mlirGen(VariableExprAST &expr) {
LogicalResult mlirGen(ReturnExprAST &ret) {
mlir::Value mlirGen(LiteralExprAST &lit) {
mlir::Value mlirGen(CallExprAST &call) {
LogicalResult mlirGen(PrintExprAST &call) {
mlir::Value mlirGen(NumberExprAST &num) {
mlir::Value mlirGen(ExprAST &expr) {
mlir::Value mlirGen(VarDeclExprAST &vardecl)
```

MLIR表达式如何生成


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

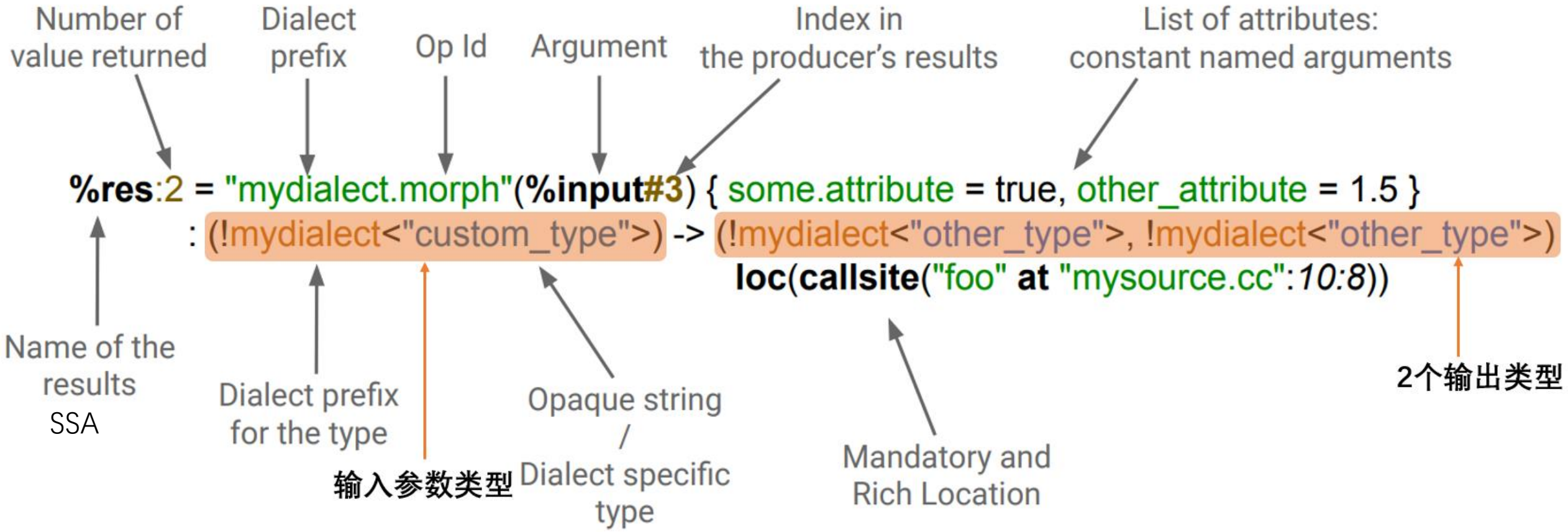
1.MLIR表达式如何理解？
2.TransposeOp在哪里定义呢？

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

MLIR表达式(类似LLVM IR可读形式)

Operations, Not Instructions

- No predefined set of instructions
- Operations are like “opaque functions” to MLIR

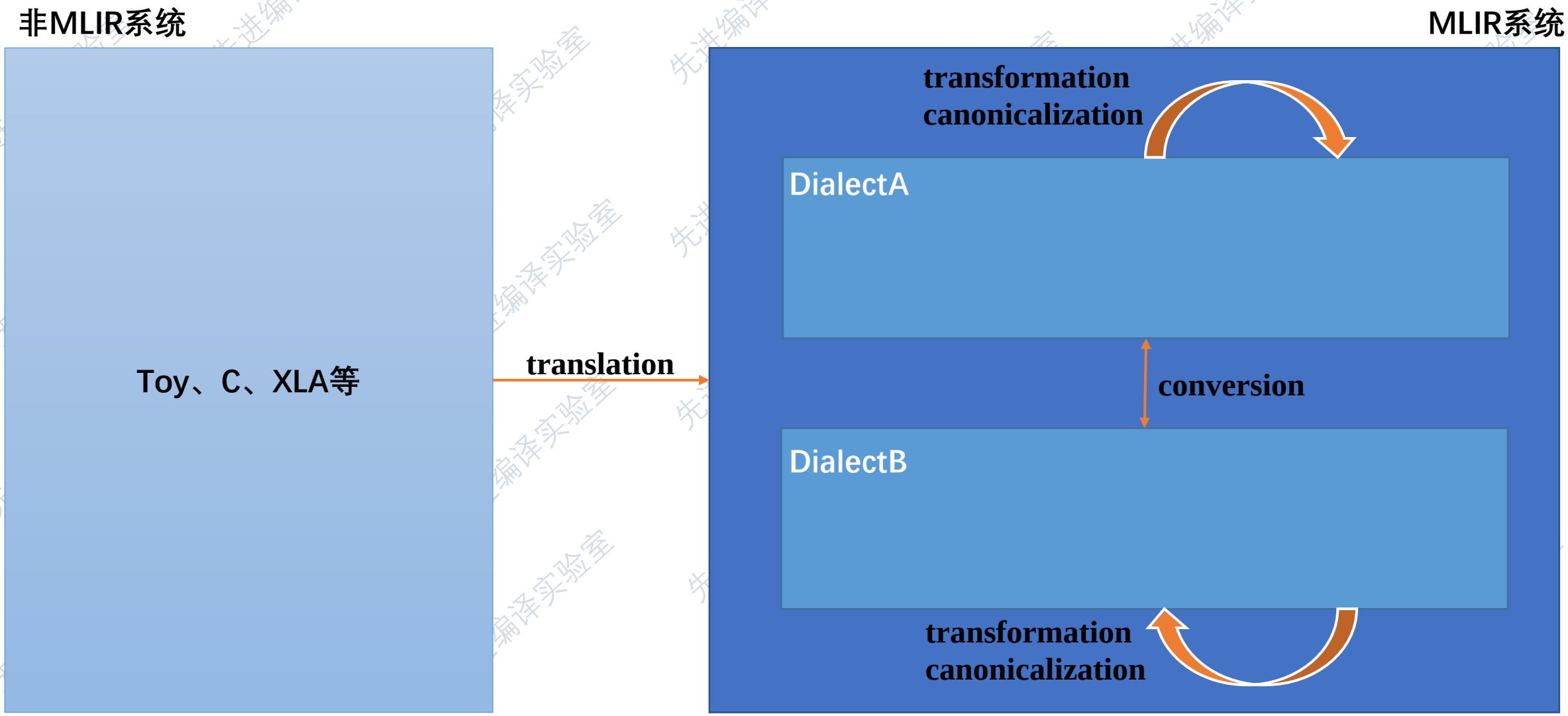


MLIR之Dialect简介

A MLIR dialect includes:

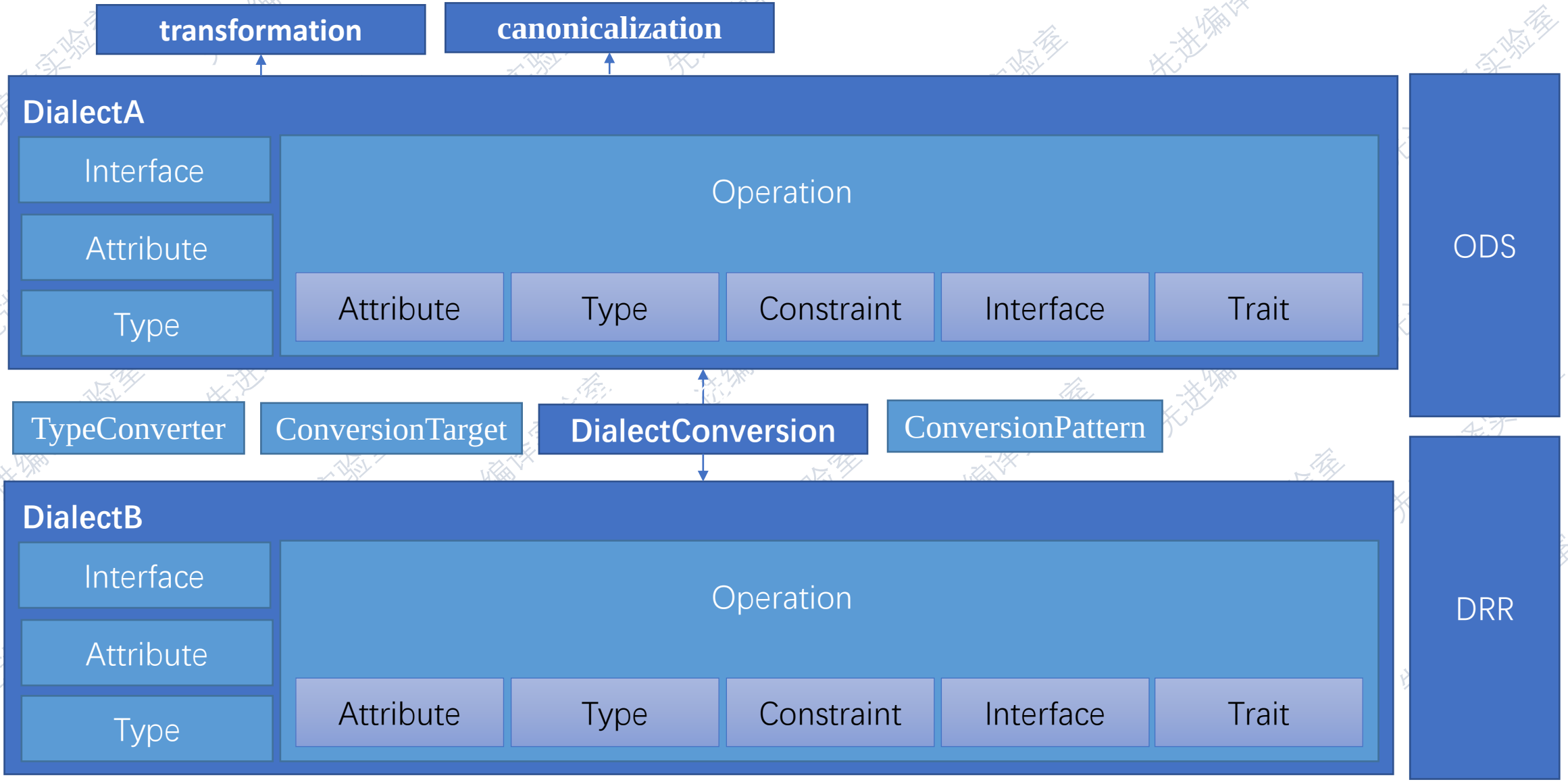
- A prefix (“namespace” reservation)
- A list of custom types, each its C++ class.
- A list of operations, each its name and C++ class implementation:
  - Verifier for operation invariants (e.g. *toy.print* must have a single operand)
  - Semantics (has-no-side-effects, constant-folding, CSE-allowed, ….)
- Possibly custom parser and assembly printer
- Passes: analysis, transformations, and dialect conversions.

Translation、Conversion和transformation区别



MLIR主要构件

Operation是Dialect的重要组成部分，可以看作方言语义的基本元素。



Dialect创建之ODS框架

A Toy Dialect: The Dialect

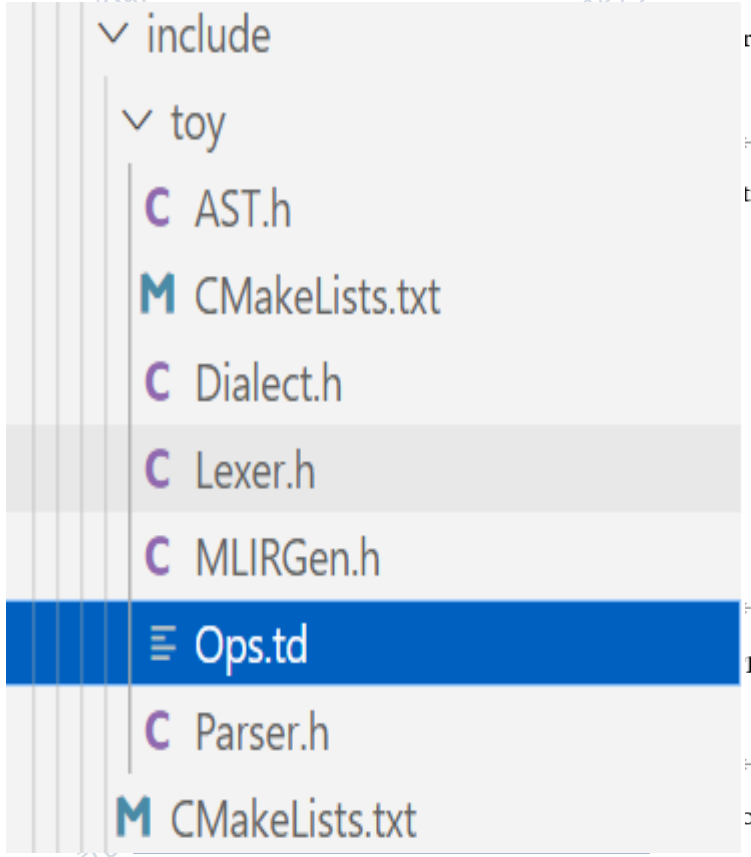
```
class ConstantOp : public mlir::Op<
                   ConstantOp,
                   mlir::OpTrait::ZeroOperands,
                   mlir::OpTrait::OneResult,

mlir::OpTraits::OneTypedResult<TensorType>::Impl> {
 public:
  using Op::Op;
  static llvm::StringRef getOperationName() { return
"toy.constant"; }
  mlir::DenseElementsAttr getValue();
  LogicalResult verifyInvariants();
  static void build(mlir::OpBuilder &builder,
mlir::OperationState &state,
                    mlir::Type result, mlir::DenseElementsAttr
value);
  static void build(mlir::OpBuilder &builder,
mlir::OperationState &state,
                    mlir::DenseElementsAttr value);
  static void build(mlir::OpBuilder &builder,
mlir::OperationState &state,
                    double value);
};
```

```
def ConstantOp : Toy_Op<"constant", [NoSideEffect]> {
  let summary = "constant";
  let description = [{
    Constant operation turns a literal into an SSA value. The
data is attached
    to the operation as an attribute. For example:
    ```mlir
      %0 = toy.constant dense<[[1.0, 2.0, 3.0], [4.0, 5.0,
6.0]]>
                          : tensor<2x3xf64>
    ```
  }];
  let arguments = (ins F64ElementsAttr:$value);
  let results = (outs F64Tensor);
  let hasCustomAssemblyFormat = 1;
  let builders = [
    OpBuilder<(ins "DenseElementsAttr":$value), [{
      build($_builder, $_state, value.getType(), value);
    }]>,
    OpBuilder<(ins "double":$value)>
  ];
  let hasVerifier = 1;
}
```

Operation创建之ODS框架

整个TableGen模块是基于Operation Definition Specification (ODS)框架进行编写以及发挥作用




```
//===----------------------------------------------------------------------===//
// toy::TransposeOp definitions
//===----------------------------------------------------------------------===//
ransposeOpOperandAdaptor::TransposeOpOperandAdaptor(ArrayRef<Value> values) {
 tblgen_operands = values;

tringRef TransposeOp::getOperationName() {
  23    /// Include the auto-generated header file containing the declaration of the toy
  24    /// dialect.
  25    #include "toy/Dialect.h.inc"
  26
  27    /// Include the auto-generated header file containing the declarations of the
  28    /// toy operations.
  29    #define GET_OP_CLASSES
  30    #include "toy/Ops.h.inc"

peration::result_range TransposeOp::getODSResults(unsigned index) {
 return {std::next(getOperation()->result_begin(), index), std::next(getOperation()->result_begin(), index + 1)};

oid TransposeOp::build(Builder *odsBuilder, OperationState &odsState, Type resultType0, Value input) {
 odsState.addOperands(input);
 odsState.addTypes(resultType0);
}
LogicalResult TransposeOp::verify() {
```

MLIR表达式变形(canonicalization优化)

方式1：手动编写代码进行表达式的匹配与重写

```
def transpose_transpose(x) {
  return transpose(transpose(x));
}
```

变形前MLIR表达式

```
func @transpose_transpose(%arg0: tensor<*xf64>) -> tensor<*xf64> {
  %0 = "toy.transpose"(%arg0) : (tensor<*xf64>) to tensor<*xf64>
  %1 = "toy.transpose"(%0) : (tensor<*xf64>) to tensor<*xf64>
  "toy.return"(%1) : (tensor<*xf64>) -> ()
}
```

变形后MLIR表达式

```
func @transpose_transpose(%arg0: tensor<*xf64>) -> tensor<*xf64> {
  "toy.return"(%arg0) : (tensor<*xf64>) -> ()
}
```

```
mlir::PatternMatchResult matchAndRewrite(TransposeOp op, mlir::PatternRewriter &rewriter) const override {
  // Look through the input of the current transpose.
  mlir::Value transposeInput = op.getOperand();
  TransposeOp transposeInputOp =
  llvm::dyn_cast_or_null<TransposeOp>(transposeInput.getDefiningOp());

  // If the input is defined by another Transpose, bingo!
  if (!transposeInputOp)
    return matchFailure();

  // Use the rewriter to perform the replacement.
  rewriter.replaceOp(op, {transposeInputOp.getOperand()});
  return matchSuccess();
}
```

```
// Transpose(Transpose(x)) = x
def TransposeTransposeOptPattern : Pat<
  (TransposeOp(TransposeOp $arg)),
  (replaceWithValue $arg)>;

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

表达式变形之DRR框架

方式2：DRR(Declarative Rewrite Rules) 使用基于规则的方式来自动生成表达式匹配和重写函数

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

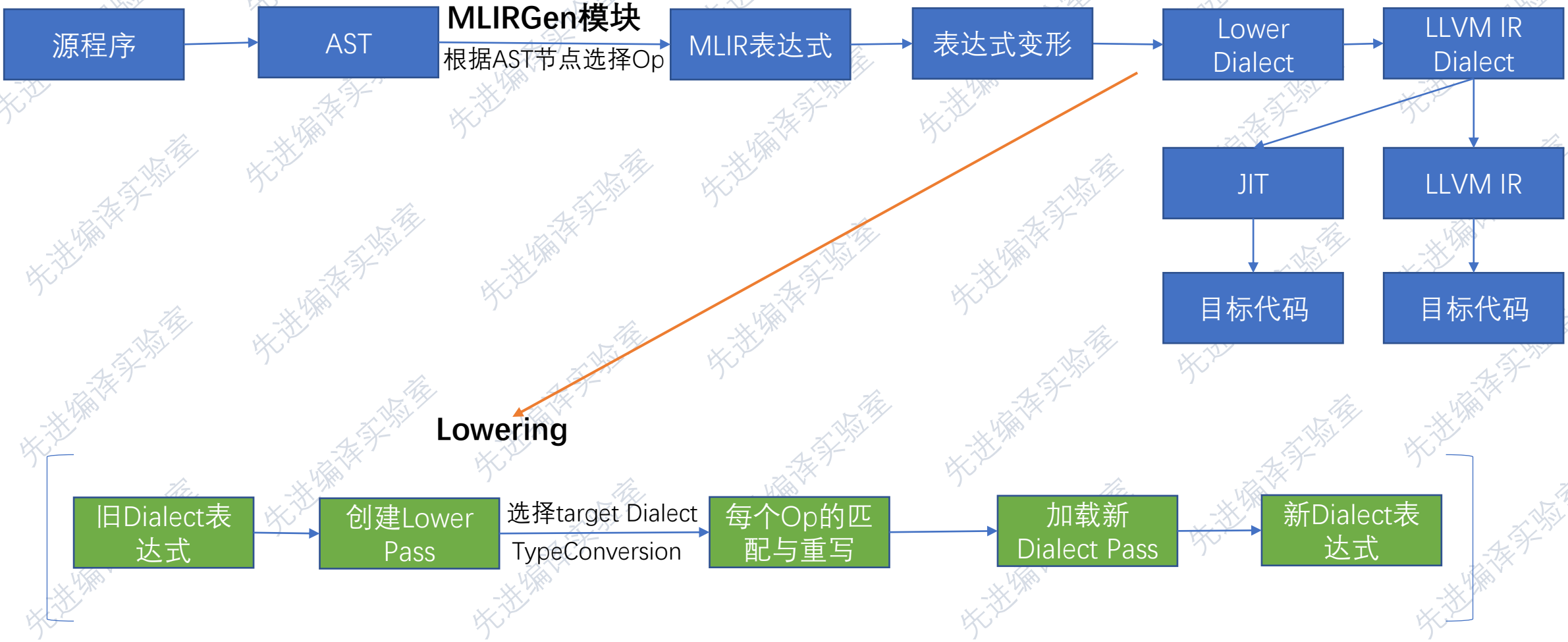
```
module {
  func @main() {
    %0 = "toy.constant"() {value = dense<[1.000000e+00, 2.000000e+00]> : tensor<2xf64>
                                  : () -> tensor<2xf64>
    %1 = "toy.reshape"(%0) : (tensor<2xf64>) -> tensor<2x1xf64>
    %2 = "toy.reshape"(%1) : (tensor<2x1xf64>) -> tensor<2x1xf64>
    %3 = "toy.reshape"(%2) : (tensor<2x1xf64>) -> tensor<2x1xf64>
    "toy.print"(%3) : (tensor<2x1xf64>) -> ()
    "toy.return"() : () -> ()
  }
}
```

```
module {
  func @main() {
    %0 = "toy.constant"() {value = dense<[[1.000000e+00], [2.000000e+00]]> \
                          : tensor<2x1xf64>} : () -> tensor<2x1xf64>
    "toy.print"(%0) : (tensor<2x1xf64>) -> ()
    "toy.return"() : () -> ()
  }
}
```

Lowering过程(Dialect Conversion)

- Converting a set of source dialects into one or more “legal” target dialects
  - The target dialects may be a subset of the source dialects
- Three main components:
  - Conversion Target:
    - Specification of what operations are legal and under what circumstances
  - Operation Conversion:
    - Dag-Dag patterns specifying how to transform illegal operations to legal ones
  - Type Conversion:
    - Specification of how to transform illegal types to legal ones
- Two Modes:
  - Partial: Not all input operations have to be legalized to the target
  - Full: All input operations have to be legalized to the target

Lowering过程



Lowering过程(初始源程序)

```
# User defined generic function that operates on unknown shaped arguments
def multiply_transpose(a, b) {
  return transpose(a) * transpose(b);
}

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

部分lowering-pattern match and rewrite

```
void ToyToAffineLoweringPass::runOnOperation() {
  // The first thing to define is the conversion target. This will define the
  // final target for this lowering.
  ConversionTarget target(getContext());

  // We define the specific operations, or dialects, that are legal targets for
  // this lowering. In our case, we are lowering to a combination of the
  // `Affine`, `Arithmetic`, `Func`, and `MemRef` dialects.
  target
      .addLegalDialect<AffineDialect, BuiltinDialect, arith::ArithmeticDialect,
                       func::FuncDialect, memref::MemRefDialect>();

  // We also define the Toy dialect as Illegal so that the conversion will fail
  // if any of these operations are *not* converted. Given that we actually want
  // a partial lowering, we explicitly mark the Toy operations that don't want
  // to lower, `toy.print`, as `legal`. `toy.print` will still need its operands
  // to be updated though (as we convert from TensorType to MemRefType), so we
  // only treat it as `legal` if its operands are legal.
  target.addIllegalDialect<toy::ToyDialect>();
  target.addDynamicallyLegalOp<toy::PrintOp>([](toy::PrintOp op) {
    return llvm::none_of(op->getOperandTypes(),
                         [](Type type) { return type.isa<TensorType>(); });
  });
  }
};
```

其中一部分Operation，

e

ing,

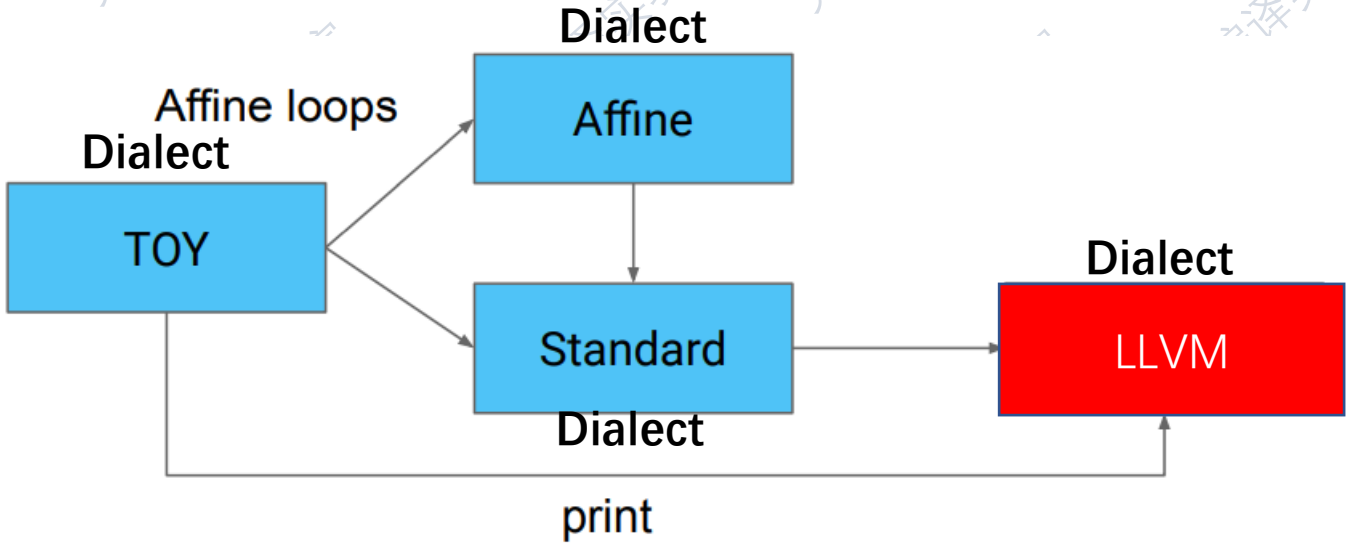
ToyToAffineLoweringPass

1. 匹配重写Operation pattern
2. 添加target和pattern

Lowering过程之full-lowering

前面已经将Toy Dialect转换为Affine Dialect、Standard Dialect以及包含Toy Dialect中的print Operation的混合操作，需要全部lowering到LLVM Dialect在lowering到LLVM IR接入LLVM 后端进行CodeGen

1. 创建Lower to LLVM Pass
2. Type Conversion
3. Conversion Target
4. 定义lowering pattern



Lowering过程之full-lowering

前面已经将Toy Dialect转换为Affine Dialect、Standard Dialect以及包含Toy Dialect中的print Operation的混合操作，需要全部lowering到LLVM Dialect在lowering到LLVM IR接入LLVM 后端进行CodeGen

1. 创建Lower to LLVM Pass

```
if(isLoweringToLLVM){
// to LLVM Dialect
pm.addPass(mlir::toy::createLowerToLLVMPass());
}
```

2. Type Conversion

```
LLVMTypeConverter typeConverter(&getContext());
```

3. Conversion Target

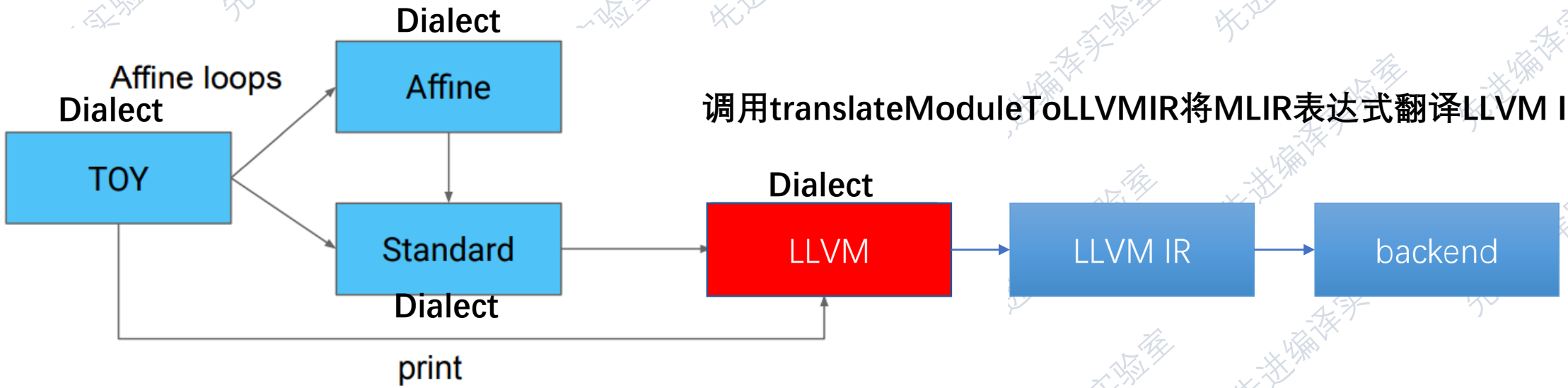
```
ConversionTarget target(getContext());
target.addLegalDialect<LLVM::LLVMDialect>();
target.addLegalOp<ModuleOp,ModuleTerminatorOp>();
```

4. 定义lowering pattern

```
if(failed(applyFullConversion(module,target,pattern,&typeConverter)))
  signalPassFailure();
```

Lowering过程之full-lowering

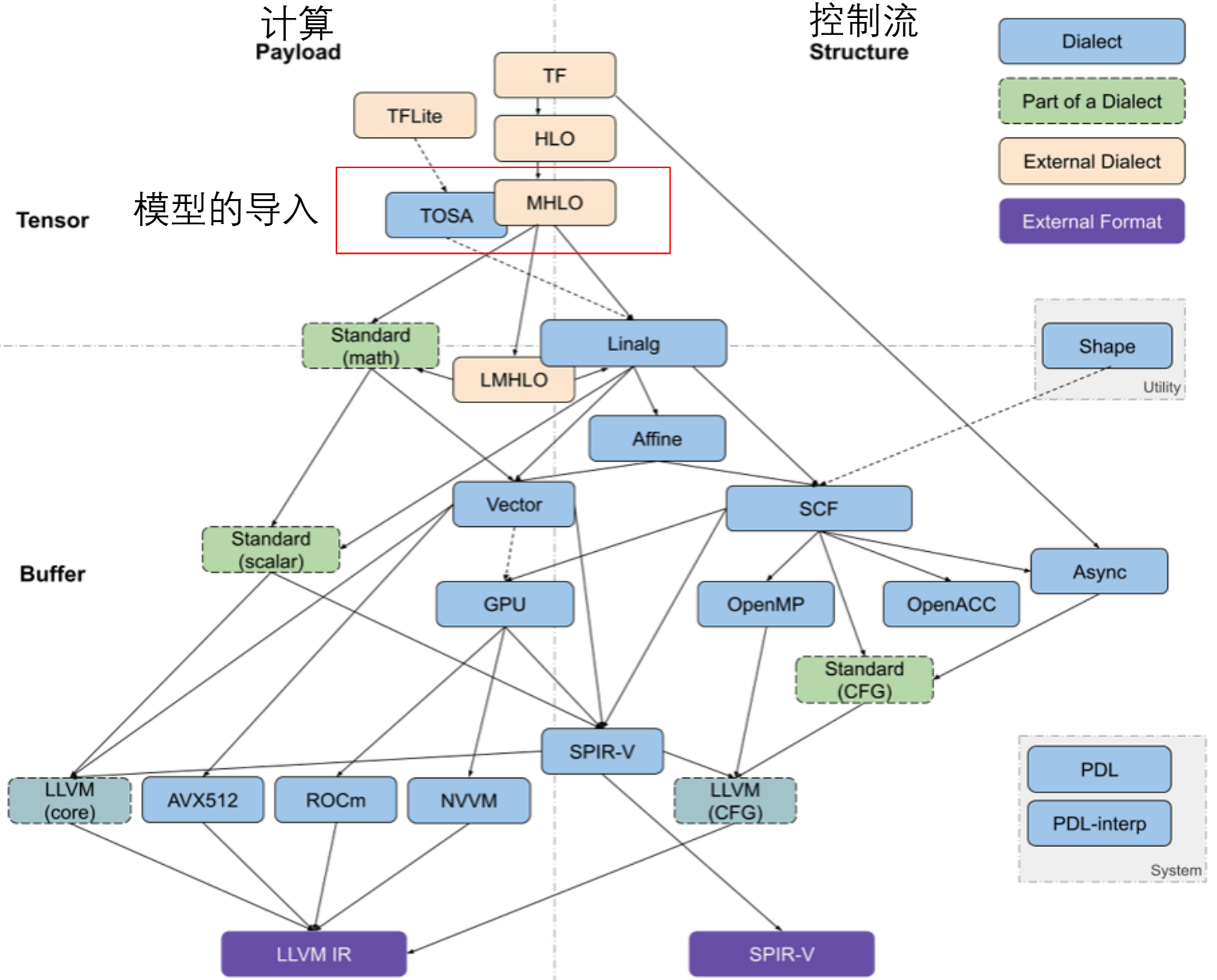
上述过程在Toy教程Ch6中直接执行优化即可获得最新lowering的LLVM Dialect对应的MLIR表达式



调用translateModuleToLLVMIR将MLIR表达式翻译LLVM IR表达式

最后使用LLVM tool chain或者MLIR自带的JIT引擎可执行输出结果

Dialect早期体系(Standard已拆分)



谢谢