MLIR Tutorial

Jaeho Lee

Corelab @ Yonsei University

Course Information

Introduction to MLIR

2 Chapters
2 Practices

Based on LLVM 15.0.0

Course Outline

- Basic Concept of MLIR
  - Goal: Learn the basic concept of LLVM IR

- Key components of MLIR
  - Goal: Learn the key components and essentials of MLIR

List of Practices

- Basic Concept of MLIR

- Key components of MLIR
  - Practice 1: First Compilation
  - Practice 2: Create transform pass

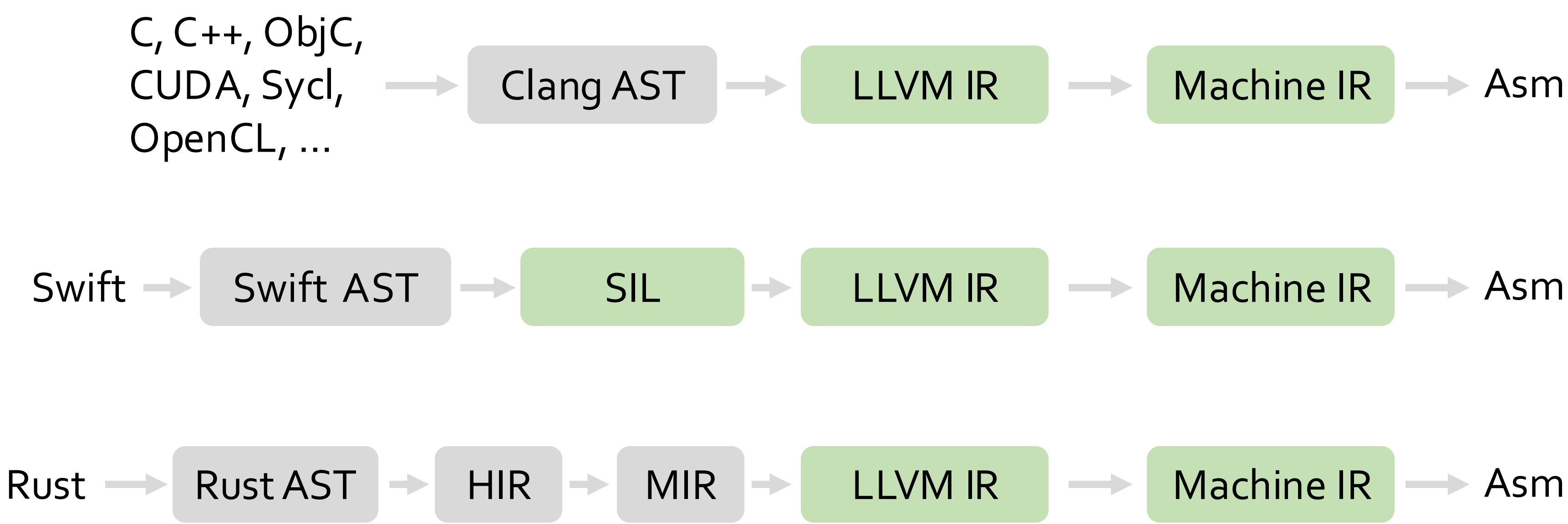
Basic Concept of MLIR

MLIR: Concept

- **M**ulti-**L**evel **I**ntermediate **R**epresentation
- Compiler Infrastructure
  - Domain-specific intermediate representation
  - High-level optimizations and portability

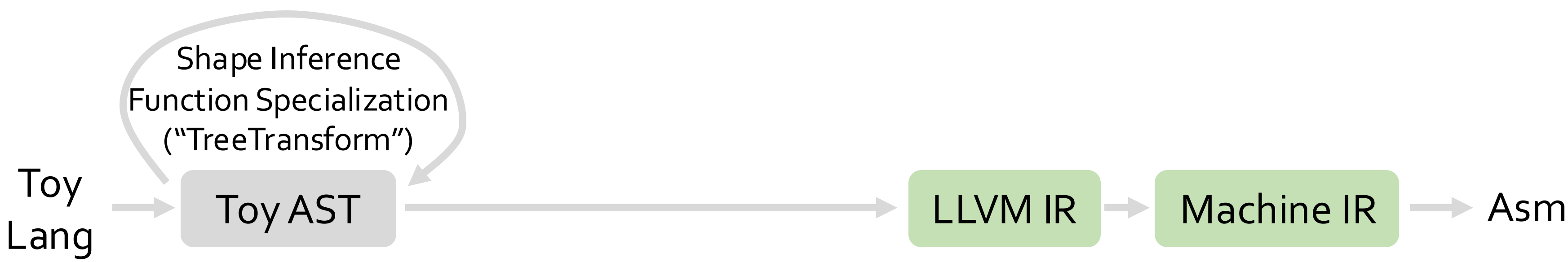
Why we need MLIR rather LLVM

- Existing Successful Compilation Models



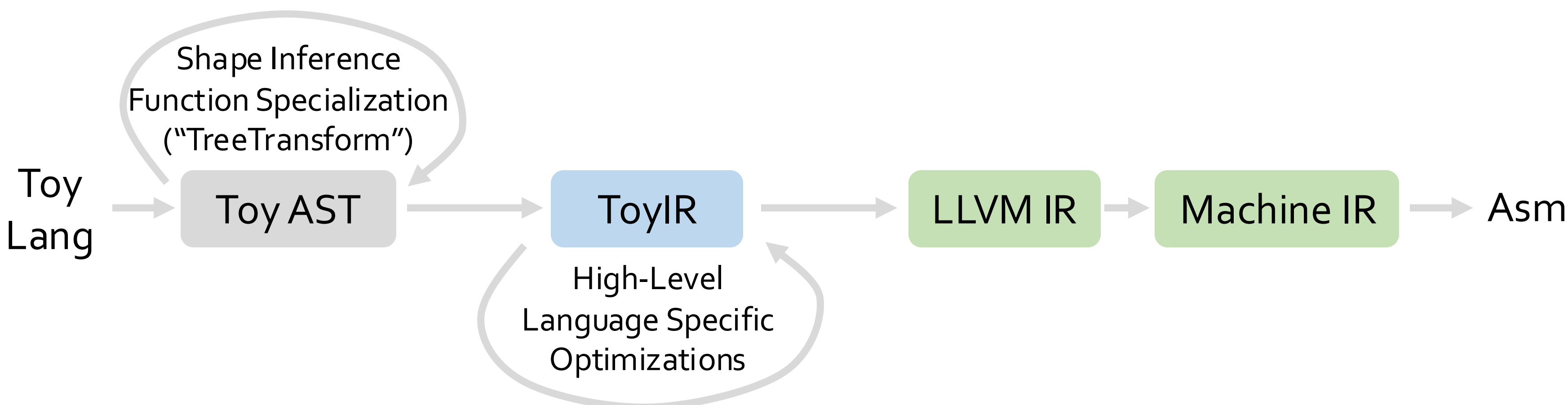
Why we need MLIR rather LLVM

- **The Toy Compiler:** the “Simpler” Approach of Clang



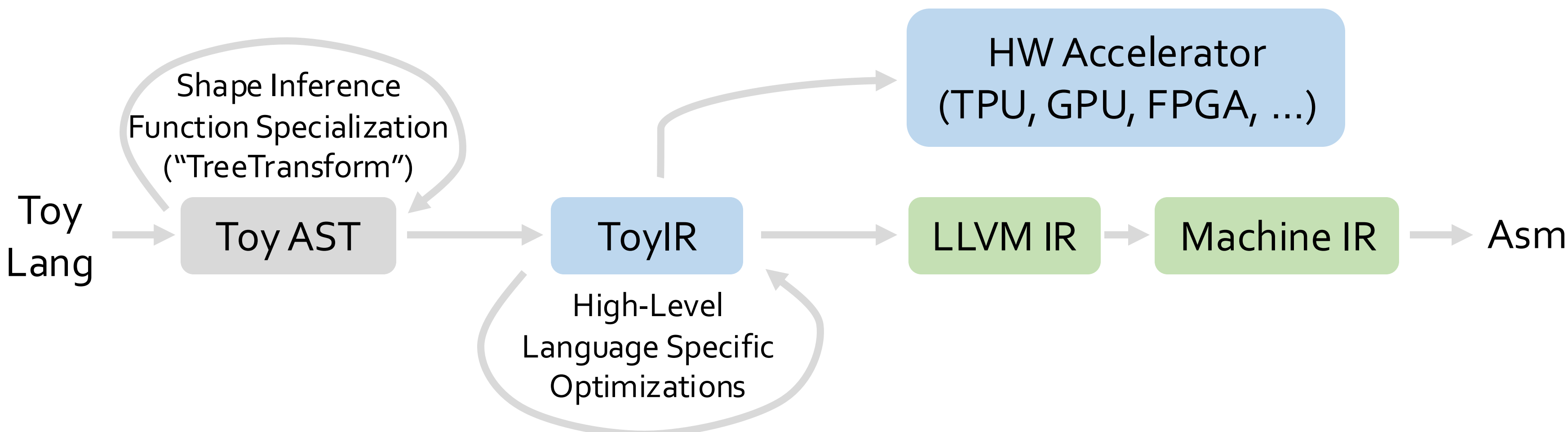
Why we need MLIR rather LLVM

- **The Toy Compiler:** With Language Specific Optimizations



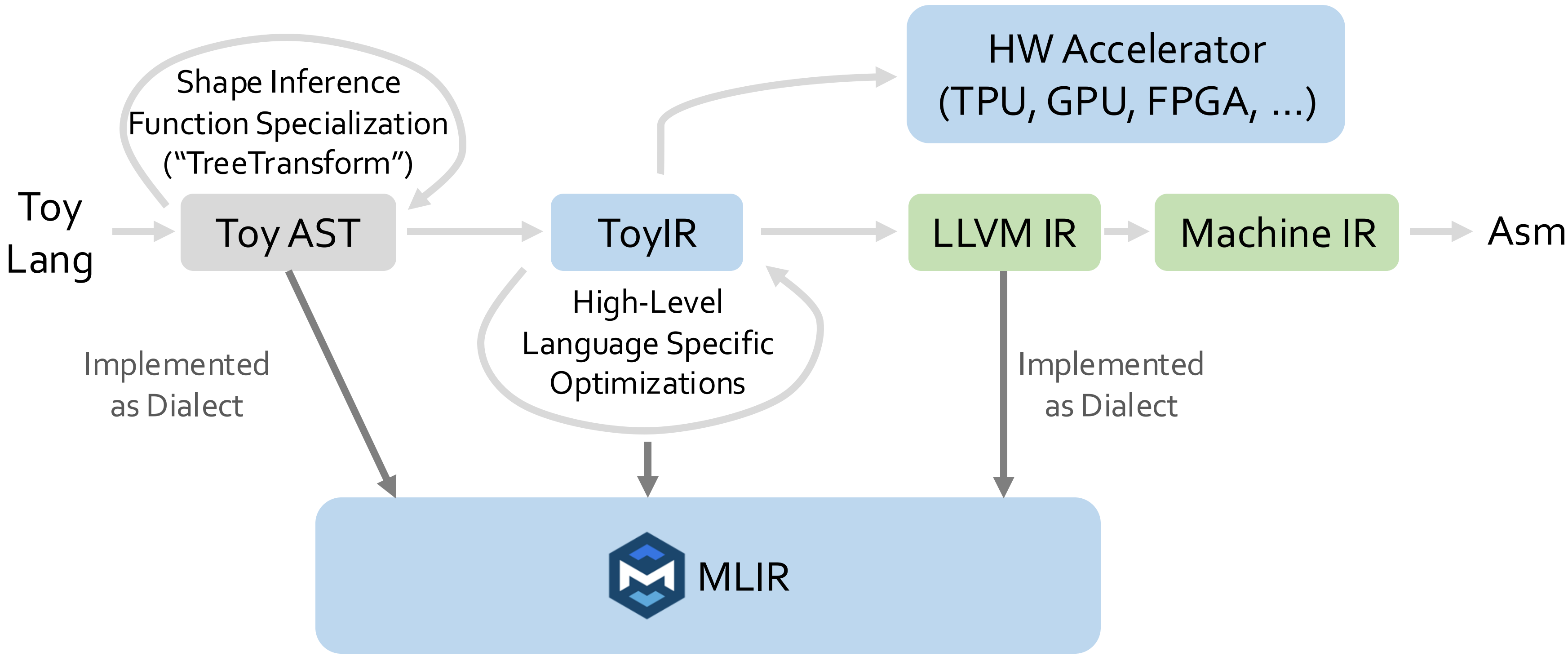
Why we need MLIR rather LLVM

- **The Toy Compiler:** Compilers in a Heterogenous World



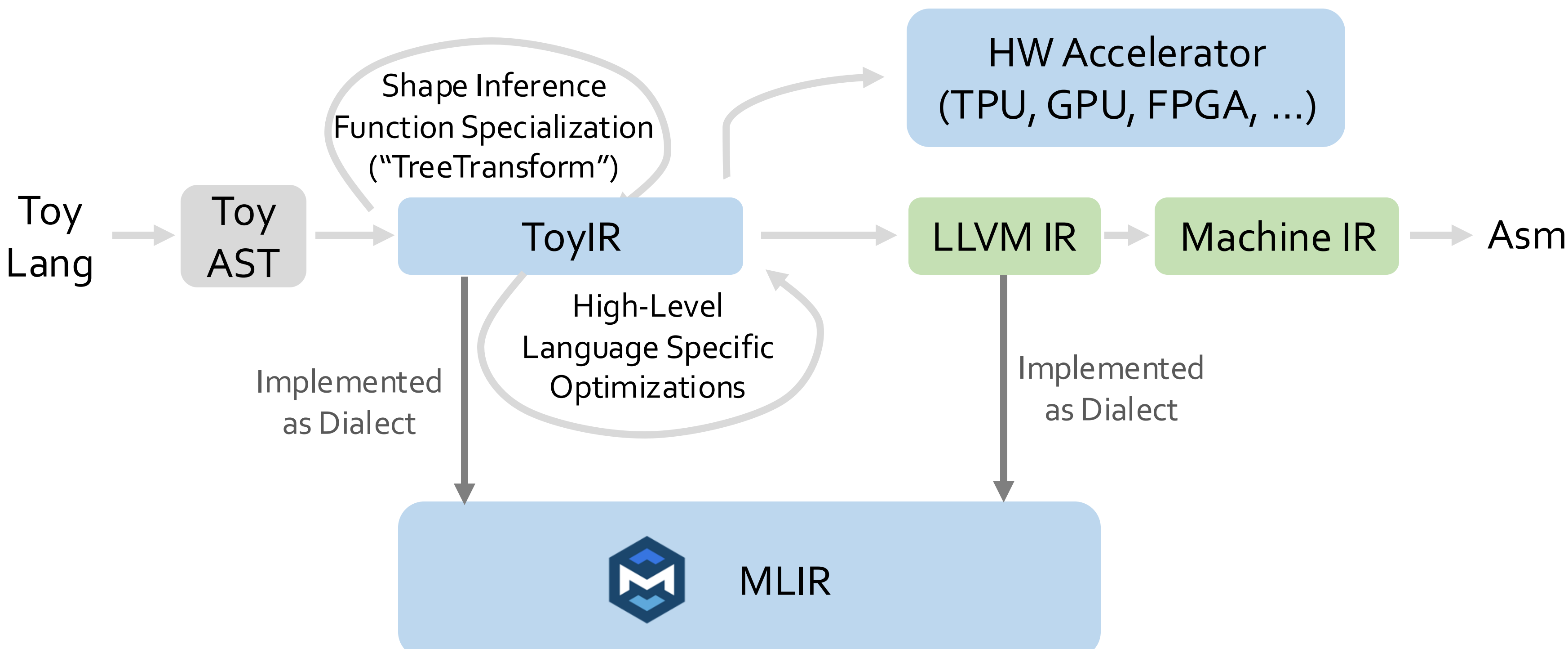
Why we need MLIR rather LLVM

- **It's all about Dialects**



Why we need MLIR rather LLVM

- **Dialect for Toy IR:** still <u>flexible enough</u> to perform shape inference and some high-level optimizations



MLIR design principles

Little built-in, everything customizable

- fully customizable IR
- express many different abstractions with Common abstractions
  - machine learning graphs
  - mathematical abstractions
  - instruction-level intermediate representations
  - etc

Progressive Lowering

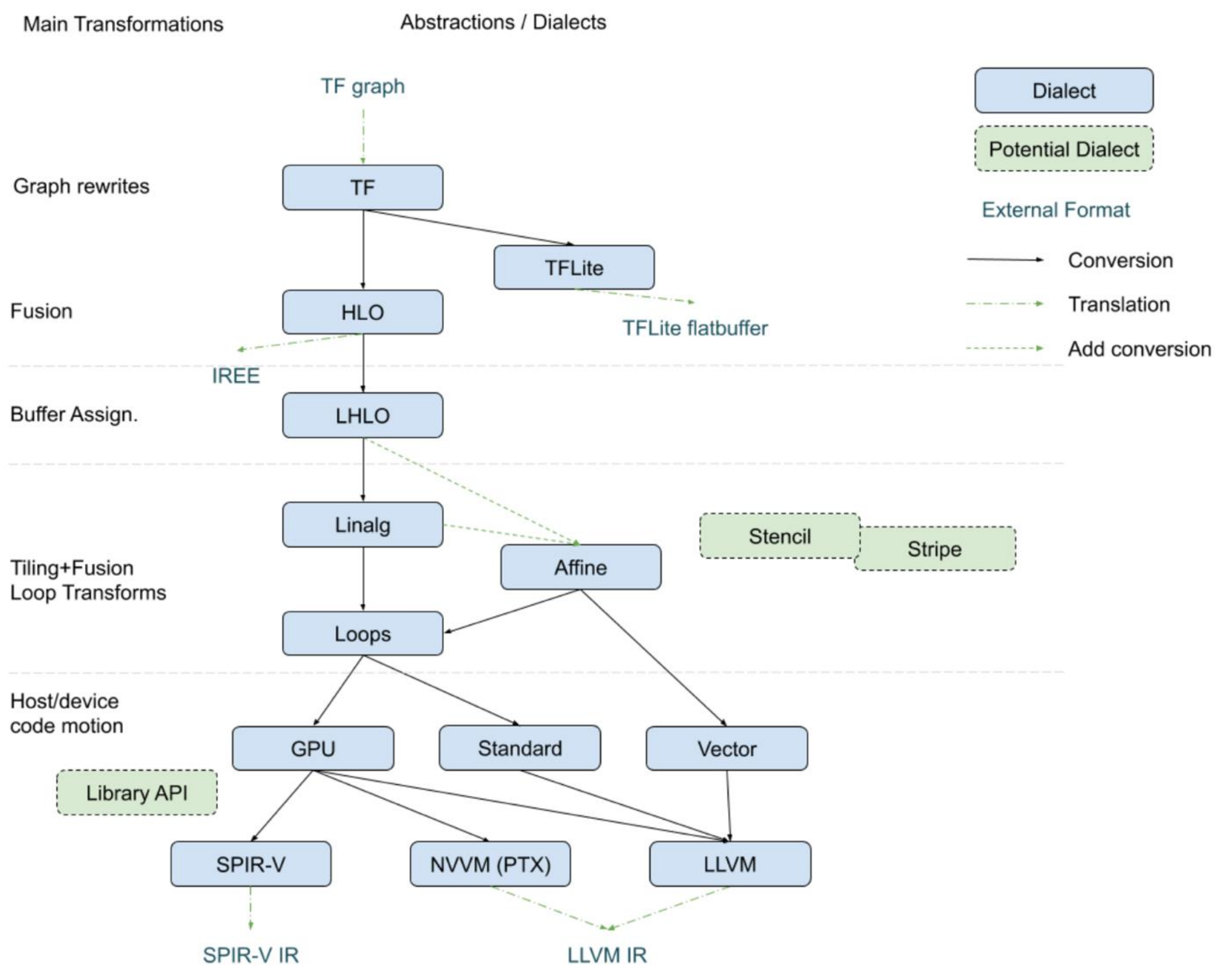
- lowering in small steps along multiple abstraction levels
- using multiple levels of abstractions
  - support variety of platforms and programming models
  - flexible design of compilation pipeline

Key components of MLIR

MLIR Terminology

- Dialect
  - Intermediate representation of specific abstraction level
- Conversion
  - Compile to different dialects
- Transform
  - Optimization inside such dialect
- Translate
  - converts operations of dialect into some other language (specifically LLVM)

Structure of MLIR

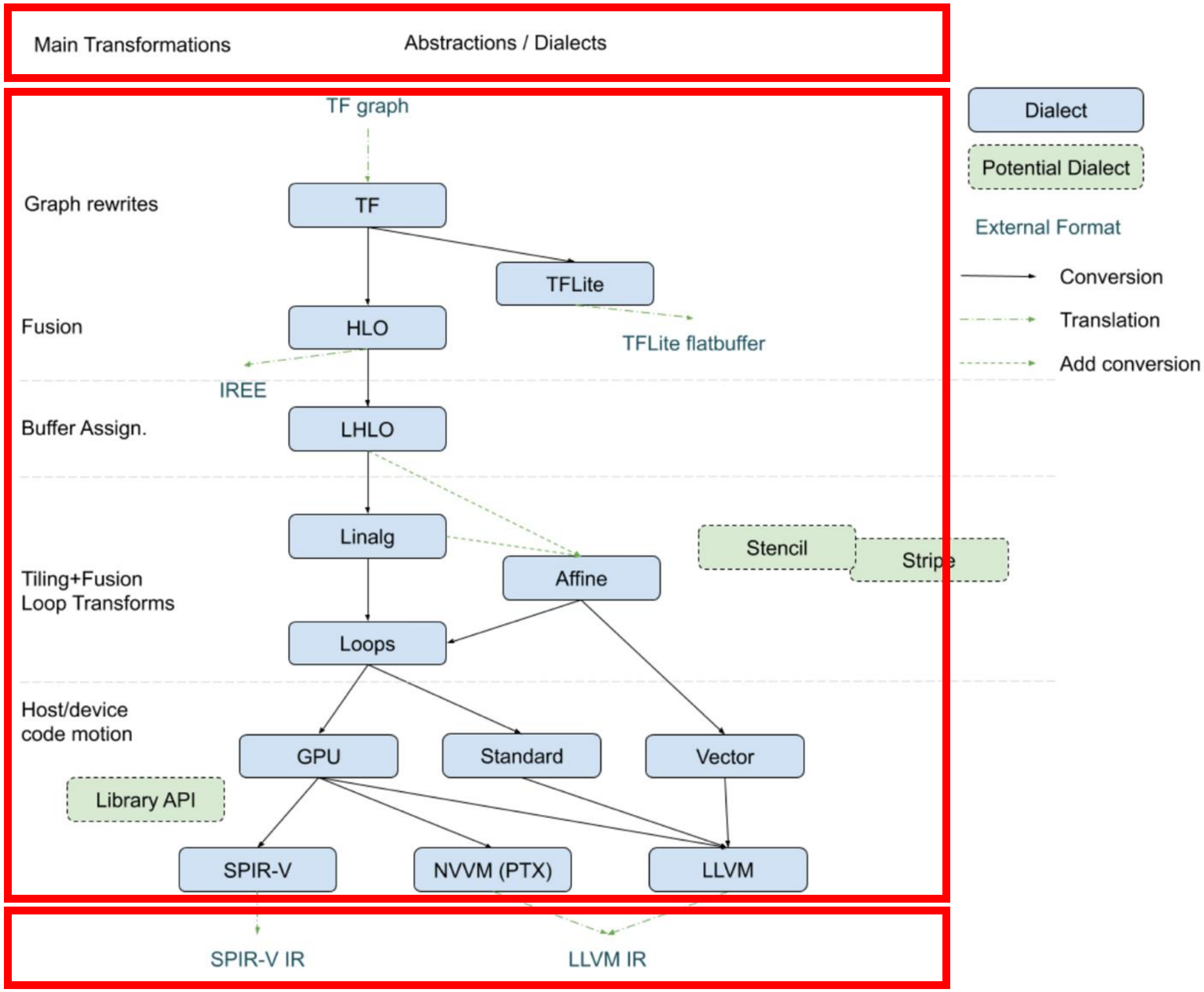


Structure of MLIR

Frontend

Midend

Backend



Let's set Docker environment

- run_docker.sh

```
docker run -td --workdir /root --name 나만의이름 --gpus all corelabyonsei/mlir-tutorial:latest
docker start 나만의이름
```

- exec_docker.sh

```
docker exec -it 나만의이름 bash
```

Practice 1: First Compilation

- Goal
  - Learn how to use MLIR opt

- Steps

1) Write a simple high-level language program (**example.toy**)

```
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);
}
```

Practice 1: First Compilation

- Goal
  - Learn how to use MLIR opt

- Steps
  1) Write a simple high-level language program (**example.toy**)
  2) Generate **AST (ast)** with command **toyc-ch6**
  3) Generate **MLIR (mlir)** with command **toyc-ch6**
  4) Generate **LLVM dialect (mlir-llvm)** with command **toyc-ch6**
  5) Generate **LLVM IR (llvm)** with command **toyc-ch6**

- Command
  - Ex) `toyc-ch6` **`-emit=ast`** `example.toy`

MLIR: Example

- Example code (-emit=milr)

High-level lang

```
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);
}
```

MLIR

```
module {
  toy.func private @multiply_transpose(%arg0: tensor<*xf64>, %arg1: 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<[[1.000000e+00, 2.000000e+00, 3.000000e+00], [4.000000e+0
    %1 = toy.reshape(%0 : tensor<2x3xf64>) to tensor<2x3xf64>
    %2 = toy.constant dense<[1.000000e+00, 2.000000e+00, 3.000000e+00, 4.000000e+00,
    %3 = toy.reshape(%2 : tensor<6xf64>) to tensor<2x3xf64>
    %4 = toy.generic_call @multiply_transpose(%1, %3) : (tensor<2x3xf64>, tensor<2x3x
    %5 = toy.generic_call @multiply_transpose(%3, %1) : (tensor<2x3xf64>, tensor<2x3x
    toy.print %5 : tensor<*xf64>
    toy.return
  }
}
```

MLIR: Example

- Example code (-emit=milr)

High-level lang

```
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);
}
```

Operation

Region

```
module {
  toy.func private @multiply_transpose(%arg0: tensor<*xf64>, %arg1: 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<[[1.000000e+00, 2.000000e+00, 3.000000e+00], [4.000000e+0
    %1 = toy.reshape(%0 : tensor<2x3xf64>) to tensor<2x3xf64>
    %2 = toy.constant dense<[1.000000e+00, 2.000000e+00, 3.000000e+00, 4.000000e+00,
    %3 = toy.reshape(%2 : tensor<6xf64>) to tensor<2x3xf64>
    %4 = toy.generic_call @multiply_transpose(%1, %3) : (tensor<2x3xf64>, tensor<2x3
    %5 = toy.generic_call @multiply_transpose(%3, %1) : (tensor<2x3xf64>, tensor<2x3
    toy.print %5 : tensor<*xf64>
    toy.return
  }
}
```

MLIR: Components

- Operation, Region, Block

```
%results:2 = "d.operation"(%arg0, %arg1) ({
  // Regions belong to Ops and can have multiple blocks.          Region
  ^block(%argument: !d.type):                                    Block
    // Ops have function types (expressing mapping).
    %value = "nested.operation"() ({
      // Ops can contain nested regions.                     Region
      "d.op"() : () -> ()
    }) : () -> (!d.other_type)
    "consume.value"(%value) : (!d.other_type) -> ()
  ^other_block:                                                  Block
    "d.terminator"() [^block(%argument : !d.type)] : () -> ()
  })
// Ops can have a list of attributes.
{attribute="value" : !d.type} : () -> (!d.type, !d.other_type)
```

Operations

- **Operation**! Not instruction (-emit=mlir)
  - No predefined set of instructions
  - Operations are like “opaque functions” to MLIR

Dialect prefix

Op id

Arguments

Name of the result

```
%4 = "toy.generic_call" @multiply_transpose(%1, %3)
  : (tensor<2x3xf64>, tensor<2x3xf64>) -> tensor<*xf64>
```

Dialect prefix for the type

Opaque string / Dialect specific type

Operations

Arguments

Name of the results

Dialect prefix

Op id

Index in the producer's result

List of **attributes**: Constant named arguments

```
%res:2 = "mydialect.morph"(%input#3) { some.attribute = true, other_attribute = 1.5 }
        : (!mydialect<"custom_type">) -> (!mydialect<"other_type">, !mydialect<"other_type">)
```

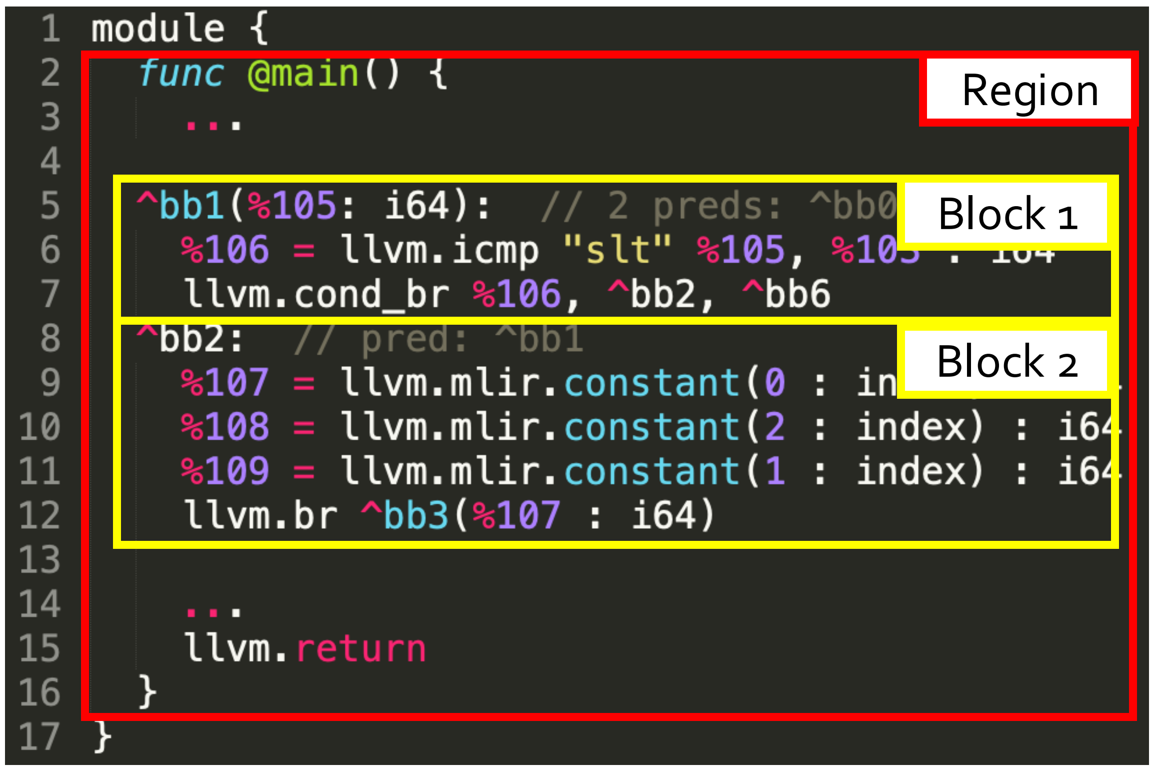
Number of the results

Dialect prefix for the type

Opaque string / **Dialect** specific type

Regions

- Container that holds one or more blocks



Regions: SSACFG regions

- SSACFG (Static Single Assignment Control Flow Graph)
  - Describes control flow between blocks

```
func @example(%arg: i32) {
  ^bb0:
    cond_br %condition, ^bb1, ^bb2

  ^bb1:
    // do something
    br ^bb3

  ^bb2:
    // do something else
    br ^bb3

  ^bb3:
    // merge point
}
```

Regions: Graph Regions

- Do not necessarily require control flow between blocks
- Used in representations like Data Flow Graphs or Computational Graphs

```
// A custom operation that encapsulates a Graph Region
graph_op {
  // Entry Block (No explicit control flow from here)
  ^bb0:
    // Operations without control-flow dependencies
    %result1 = some_op1
    %result2 = some_op2
    graph_terminator
}
```

Blocks

- A Block is a list of operations
- Instructions inside the block are executed in order

Block 1

```
module {
  func @main() {


  ^bb1(%105: i64):  // 2 preds: ^bb0, ^bb5
    %106 = llvm.icmp "slt" %105, %103 : i64
    llvm.cond_br %106, ^bb2, ^bb6
  ^bb2:  // pred: ^bb1
    %107 = llvm.mlir.constant(0 : index) : i64
    %108 = llvm.mlir.constant(2 : index) : i64
    %109 = llvm.mlir.constant(1 : index) : i64
    llvm.br ^bb3(%107 : i64)

    ...
    llvm.return
  }
}
```

Block 2

Blocks: Single Static Assignment (SSA)

- Every value must be defined **only once**
  - In other words, every value has a **single** definition

```
%x = 1 + 2
%x = %x + 3
```

```
%x = 1 + 2
%y = %x + 3
```

- Facilitate program analyses
  - Liveness Analysis: From **DEF** to last **USE**
  - Constant Propagation: If **DEF** is constant, then **USE** is also constant

Blocks: Block argument

- In LLVM, there is phi-node

```c
int x;
if (condition) {
    x = 10;
} else {
    x = 20;
}
// use x
```

```llvm
%entry:
  br i1 %condition, label %true_branch, label %false_branch

%true_branch:
  br label %merge

%false_branch:
  br label %merge

%merge:
  %x = phi i32 [10, %true_branch], [20, %false_branch]
  ; use x
```

Blocks: Block argument

- In LLVM, there is phi-node

```c
int x;
if (condition) {
    x = 10;
} else {
    x = 20;
}
// use x

```

```llvm
%entry:
  br i1 %condition, label %true_branch, label %false_branch

%true_branch:
  br label %merge

%false_branch:
  br label %merge

%merge:
  %x = phi i32 [10, %true_branch], [20, %false_branch]
  ; use x
```

Too low-level

Blocks: Block argument

- But in MLIR, **Block Argument !**

```
int x;
if (condition) {
    x = 10;
} else {
    x = 20;
}
// use x

```

```
func @example(%condition: i1) {
  cond_br %condition, ^bb1, ^bb2

^bb1:
  br ^bb3(10 : i32)

^bb2:
  br ^bb3(20 : i32)

^bb3(%x: i32):
  // use x
}
```

Blocks: Block argument

```
int x;
if (condition) {
    x = 10;
} else {
    x = 20;
}
// use x

```

- But in MLIR, **Block Argument !**

```
func @example(%condition: i1) {
  cond_br %condition, ^bb1, ^bb2

^bb1:
  br ^bb3(10 : i32)

^bb2:
  br ^bb3(20 : i32)

^bb3(%x: i32):
  // use x
}
```

similar to using parameters in real programming languages

Now go back to Operations

- Dialects are the primitive unit of MLIR

Dialect prefix

Op id

Arguments

Name of the result

```
%4 = "toy.generic_call" @multiply_transpose(%1, %3)
    : (tensor<2x3xf64>, tensor<2x3xf64>) -> tensor<*xf64>
```

Dialect prefix for the type

Opaque string / Dialect specific type

Create Toy Dialect: Directory setup

- {PROJECT_ROOT}**/include/Dialect/Toy**
  - for public include files | *.td, *.h, *.hpp
- {PROJECT_ROOT}**/lib/Dialect/Toy**
  - for sources
- {PROJECT_ROOT}**/lib/Dialect /Toy/IR**
  - for operations | *.cpp
- {PROJECT_ROOT}**/lib/Dialect/Toy/Transforms**
  - for transforms
- {PROJECT_ROOT}**/test/Dialect/Toy**
  - for tests

Create Toy Dialect: Cmake configuration

- TableGen Targets
  - {PROJECT_ROOT}/include/Dialect/{Dialect Name}/IR/CMakeLists.txt

```
add_mlir_dialect({Dialect_Name}Ops {tablegen_target_name})
add_mlir_doc({Dialect_Name}Ops {Dialect_Name}Dialect Dialects/ -gen-dialect-doc)
```

- Library Targets

```
add_mlir_dialect_library({dialect-library-target}
        DEPENDS
        <tablegen-targets>

        LINK_COMPONENTS
        Core

        LINK_LIBS PUBLIC
        <some-other-library>
)
```

Create Toy Dialect: Dialect Implementation

- Dialect definition in ODS format

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

  let summary = ”…";

  let description = [{ …}];

  let cppNamespace = ”mlir::toy";
}

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

Create Toy Dialect: Details

- Let’s go to Docker

Practice 2: Create transform pass

- Let’s go to Docker

```
module {
  func @main(%arg0: tensor<2x3xf64>, %arg1: tensor<2x3xf64>) {
    %0 = "toy.add"(%arg0, %arg1) : (tensor<2x3xf64>, tensor<2x3xf64>) -> tensor<2x3xf64>
    %1 = "toy.add"(%arg0, %arg1) : (tensor<2x3xf64>, tensor<2x3xf64>) -> tensor<2x3xf64>
    toy.return %1 : tensor<2x3xf64>
  }
}
```

- %0 is redundant again!
- Let’s remove the “No Use” operations in Module