Google

MLIR: Multi-Level Intermediate Representation

Building a Compiler with MLIR

LLVM Dev Mtg, 2020

Mehdi Amini and River Riddle
Google

MLIR

https://mlir.llvm.org/

Overview

Tour of MLIR (with many simplification) by way of implementing a basic toy language

- Defining a Toy language
- Core MLIR Concepts: operations, regions, dialects
- Representing Toy using MLIR
  - Introducing dialect, operations, ODS, verifications
  - Attaching semantics to custom operations
- High-level language specific optimizations
- Writing passes for structure rather than ops
  - Op interfaces for the win
- Lowering to lower-level dialects
  - The road to LLVM IR

What is MLIR?

- Framework to build a compiler IR: define your type system, operations, etc.
- Toolbox covering your compiler infrastructure needs
  - Diagnostics, pass infrastructure, multi-threading, testing tools, etc.
- Batteries-included:
  - Various code-generation / lowering strategies
  - Accelerator support (GPUs)
- Allow different levels of abstraction to freely co-exist
  - Abstractions can better target specific areas with less high-level information lost
  - Progressive lowering simplifies and enhances transformation pipelines
  - No arbitrary boundary of abstraction, e.g. host and device code in the same IR at the same time

Examples:

- High-Level IR for general purpose languages: FIR (Flang IR)
- “ML Graphs”: TensorFlow/ONNX/XLA/….
- HW design: CIRCT project
- Runtimes: TFRT, IREE
- Research projects: Verona (concurrency), RISE (functional), ...

https://mlir.llvm.org/users/

Google

Introducing MLIR by creating: a Toy Language

Google

Let’s Build a Toy Language

- 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!)

```
"template<typename A, typename B, typename C>
auto foo(A a, B b, C c) { ... }"
```

```
def foo(a, b, c) {
  var c = a + b;
  print(transpose(c));
  var d<2, 4> = c * foo(c);
  return d;
}
```

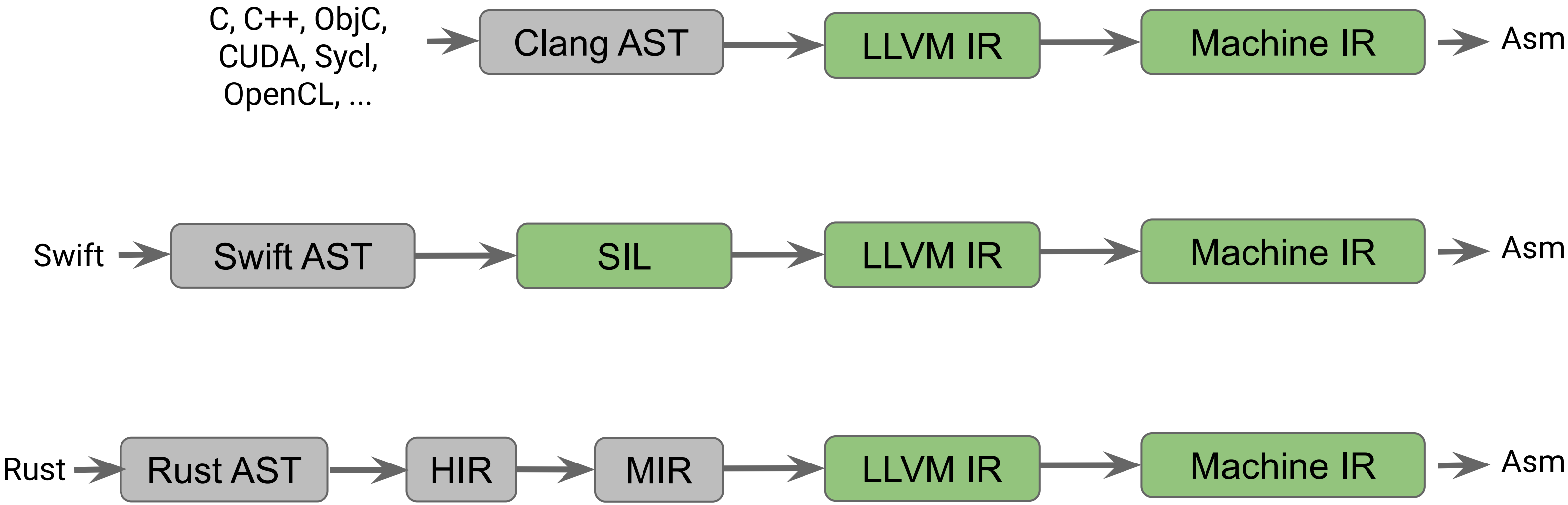
Value-based semantics / SSA

Limited set of builtin functions

Array reshape through explicit variable declaration

Only float 64s

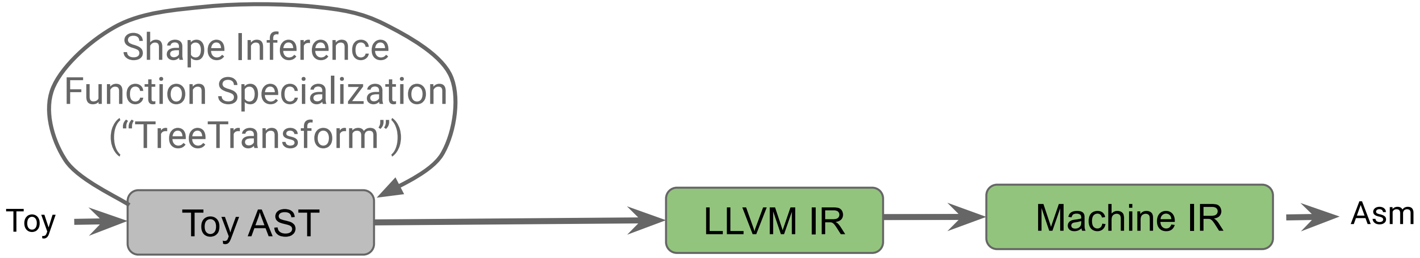
Existing Successful Compilation Models



Google

The Toy Compiler: the “Simpler” Approach of Clang

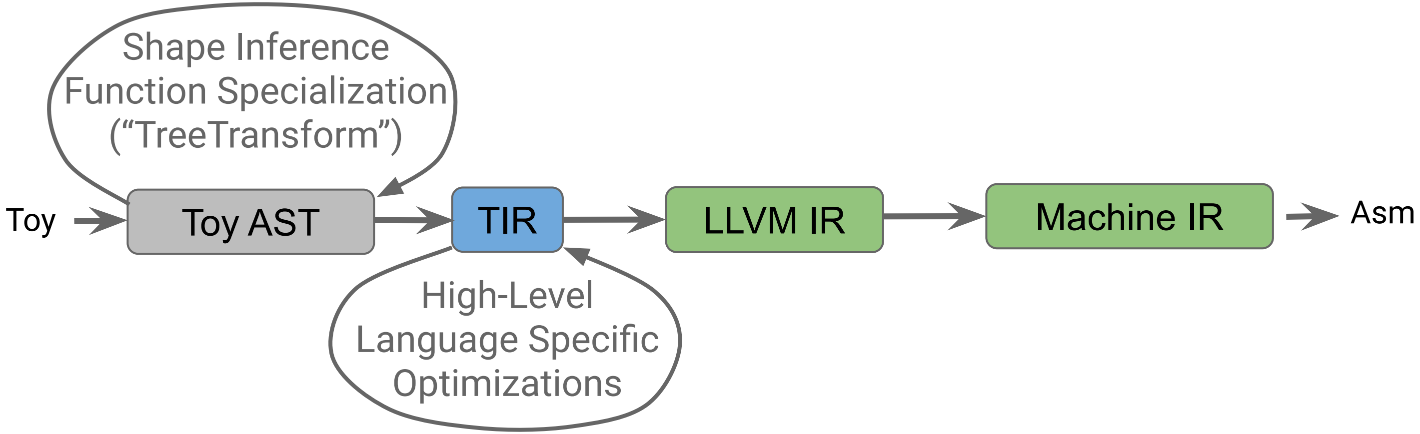
Need to analyze and transform the AST
-> heavy infrastructure!
And is the AST really the most friendly
representation we can get?



Google

The Toy Compiler: With Language Specific Optimizations

Need to analyze and transform the AST
-> heavy infrastructure!
And is the AST really the most friendly
representation we can get?

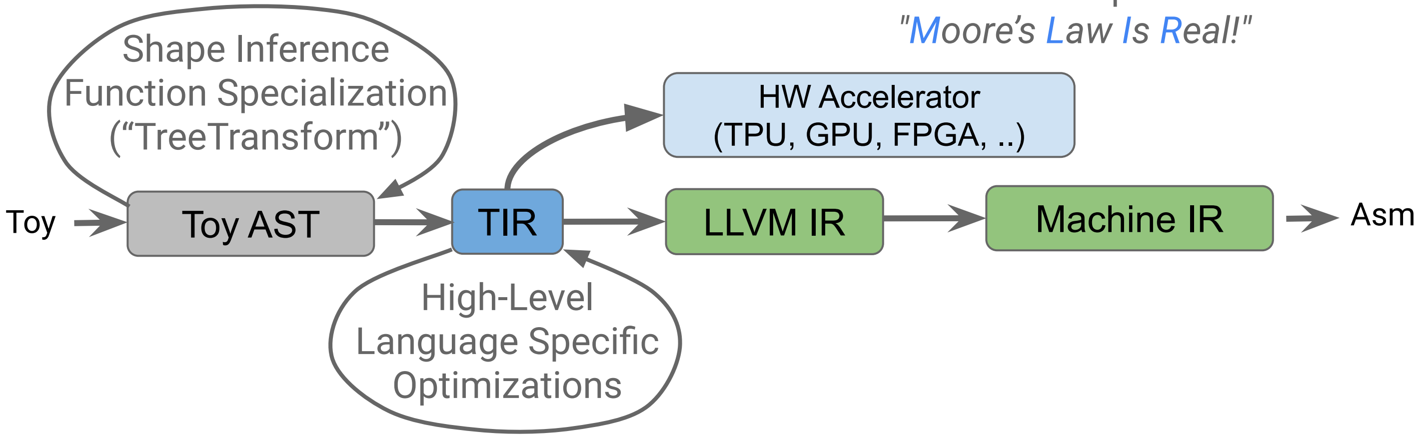


For more optimizations: we need a custom IR
Reimplement again all of LLVM's infrastructure?

Compilers in a Heterogenous World

Need to analyze and transform the AST
-> heavy infrastructure!
And is the AST really the most friendly
representation we can get?

New HW: are we extensible
and future-proof?
"Moore's Law Is Real!"

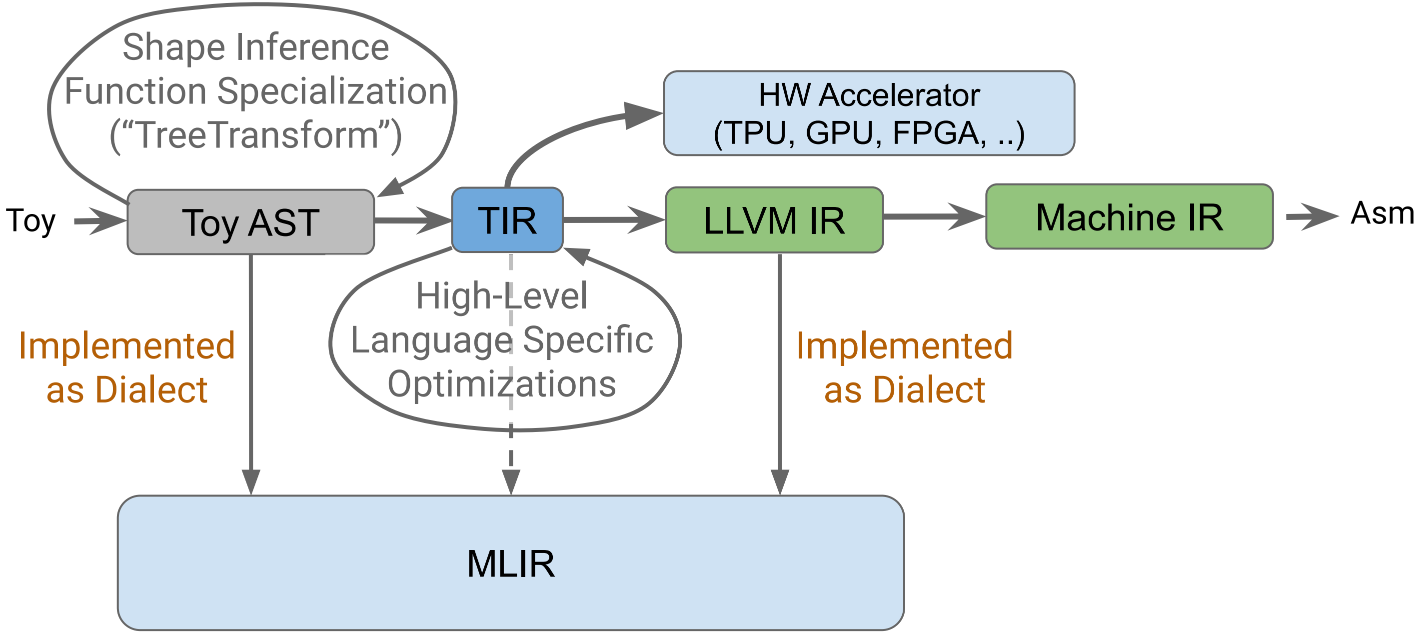


For more optimizations: a custom IR.
Reimplement again all the LLVM infrastructure?

Google

It Is All About The Dialects!

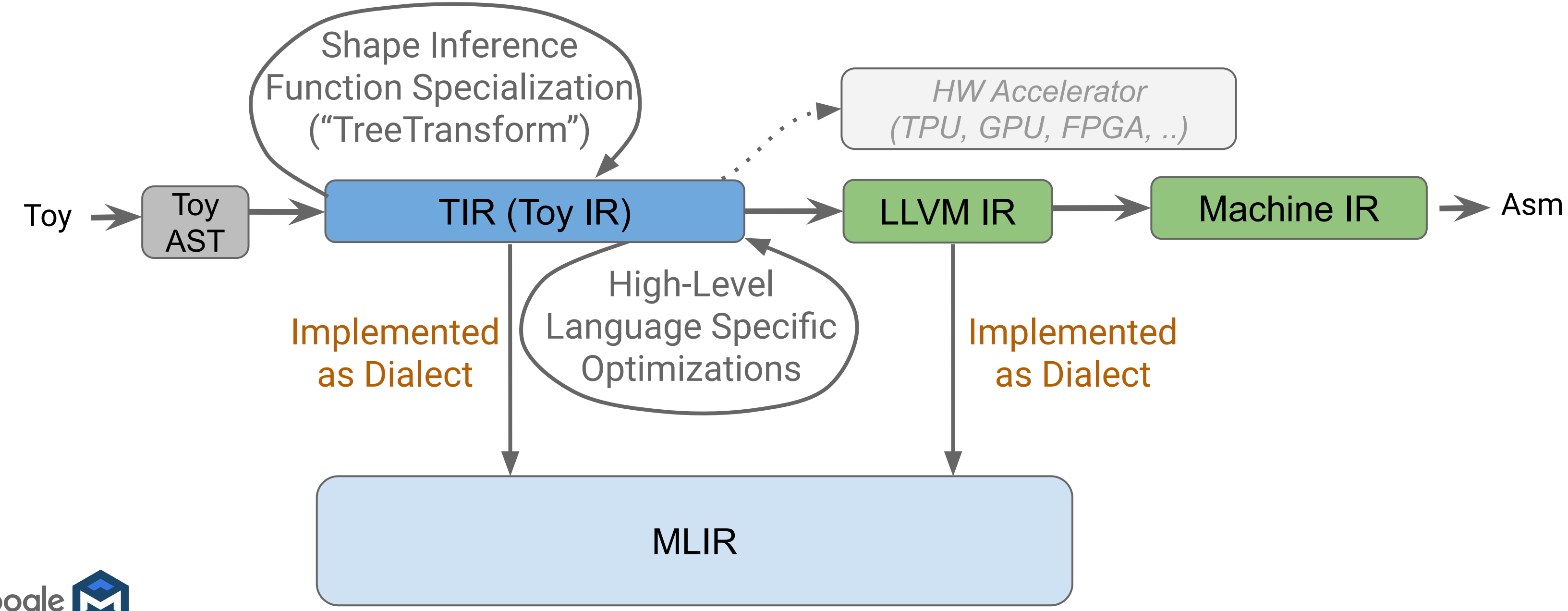
MLIR allows every level of abstraction
to be modeled as a Dialect



Google

Adjust Ambition to Our Budget (let’s fit the talk)

Limit ourselves to a single dialect for Toy IR: still flexible enough to perform shape inference and some high-level optimizations.



Google

MLIR Primer

Operations, Not Instructions

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

Number of value returned

Dialect prefix

Op Id

Argument

Index in the producer’s results

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">)
                                  loc(callsite("foo" at "mysource.cc":10:8))
```

Name of the results

Dialect prefix for the type

Opaque string / Dialect specific type

Mandatory and Rich Location

https://mlir.llvm.org/docs/LangRef/#operations
https://github.com/llvm/llvm-project/blob/master/mlir/include/mlir/IR/Operation.h#L27

Recursive nesting: Operations -> Regions -> Blocks

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

- Regions are list of basic blocks nested inside of an operation.
  - Basic blocks are a list of operations: **the IR structure is recursively nested**!
- Conceptually similar to function call, but can reference SSA values defined outside.
- SSA values defined inside don’t escape.

Google

https://mlir.llvm.org/docs/Tutorials/UnderstandingTheIRStructure/
https://mlir.llvm.org/docs/LangRef/#high-level-structure

The “Catch”

```
func @main() {
  %0 = "toy.print"() : () -> tensor<10xi1>
}
```

Yes: this is also fully valid textual IR module!

It is not valid though! Broken on many aspects:

- The *toy.print* builtin is not a terminator,
- It should take an operand,
- It shouldn’t produce any results

JSON of compiler IR ?!?

Dialects: Defining Rules and Semantics for the IR

A MLIR dialect is a logical grouping including:

- 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, ….)
- Passes: analysis, transformations, and dialect conversions.
- Possibly custom parser and assembly printer

https://mlir.llvm.org/docs/LangRef/#dialects
https://github.com/llvm/llvm-project/blob/master/mlir/include/mlir/IR/Dialect.h#L37
https://mlir.llvm.org/docs/Tutorials/CreatingADialect/

Google

Example: Affine Dialect

```
func @test() {
  affine.for %k = 0 to 10 {
    affine.for %l = 0 to 10 {
      affine.if (d0) : (8*d0 - 4 >= 0, -8*d0 + 7 >= 0)(%k) {
        // Dead code, because no multiple of 8 lies between 4 and 7.
        "foo"(%k) : (index) -> ()
      }
    }
  }
  return
}
```

With custom parsing/printing: affine.for operations with an attached region feels like a regular for!

Extra semantics constraints in this dialect: the if condition is an affine relationship on the enclosing loop indices.

```
#set0 = (d0) : (d0 * 8 - 4 >= 0, d0 * -8 + 7 >= 0)
func @test() {
  "affine.for"() {lower_bound: #map0, step: 1 : index, upper_bound: #map1} : () -> () {
  ^bb1(%i0: index):
    "affine.for"() {lower_bound: #map0, step: 1 : index, upper_bound: #map1} : () -> ()
{
    ^bb2(%i1: index):
      "affine.if"(%i0) {condition: #set0} : (index) -> () {
        "foo"(%i0) : (index) -> ()
        "affine.terminator"() : () -> ()
      } { // else block
      }
      "affine.terminator"() : () -> ()
    }
    ...
```

Same code without custom parsing/printing: **isomorphic to the internal in-memory representation.**

https://mlir.llvm.org/docs/Dialects/Affine/

LLVM as a dialect

```
%13 = llvm.alloca %arg0 x !llvm.double : (!llvm.i32) -> !llvm.ptr<double>

%14 = llvm.getelementptr %13[%arg0, %arg0]
          : (!llvm.ptr<double>, !llvm.i32, !llvm.i32) -> !llvm.ptr<double>
%15 = llvm.load %14 : !llvm.ptr<double>
llvm.store %15, %13 : !llvm.ptr<double>
%16 = llvm.bitcast %13 : !llvm.ptr<double> to !llvm.ptr<i64>
%17 = llvm.call @foo(%arg0) : (!llvm.i32) -> !llvm.struct<(i32, double, i32)>
%18 = llvm.extractvalue %17[0] : !llvm.struct<(i32, double, i32)>
%19 = llvm.insertvalue %18, %17[2] : !llvm.struct<(i32, double, i32)>
%20 = llvm.constant(@foo : (!llvm.i32) -> !llvm.struct<(i32, double, i32)>) :
        !llvm.ptr<func<struct<i32, double, i32> (i32)>>
%21 = llvm.call %20(%arg0) : (!llvm.i32) -> !llvm.struct<(i32, double, i32)>
```

Google

The Toy IR Dialect

A Toy Dialect: The Dialect

Declaratively specified in TableGen

```
def Toy_Dialect : Dialect {
  let summary = "Toy IR Dialect";
  let description = [{
    This is a much longer description of the
    Toy dialect.
    ...
  }];

  // The namespace of our dialect.
  let name = "toy";

  // The C++ namespace that the dialect class
  // definition resides in.
  let cppNamespace = "toy";
}
```

Google

A Toy Dialect: The Dialect

Auto-generated C++ class

```
class ToyDialect : public mlir::Dialect {
public:
  ToyDialect(mlir::MLIRContext *context)
    : mlir::Dialect("toy", context,
        mlir::TypeID::get<ToyDialect>()) {
    initialize();
  }

  static llvm::StringRef getDialectNamespace() {
    return "toy";
  }

  void initialize();
};
```

Declaratively specified in TableGen

```
def Toy_Dialect : Dialect {
  let summary = "Toy IR Dialect";
  let description = [{
    This is a much longer description of the
    Toy dialect.
    ...
  }];

  // The namespace of our dialect.
  let name = "toy";

  // The C++ namespace that the dialect class
  // definition resides in.
  let cppNamespace = "toy";
}
```

Google

A Toy Dialect: The Operations

```
# 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);
}
```

A Toy Dialect: The Operations

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

```
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>) -> ()
}
```

```
$ bin/toy-ch5 -emit=mlir example.toy
```

A Toy Dialect: The Operations

```
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);
}
```

```
func @main() {
  %0 = "toy.constant"() { value: dense<[[1., 2.], [3., 4.]]> : tensor<2x2xf64> }
                       : () -> tensor<2x2xf64>
  %1 = "toy.reshape"(%0) : (tensor<2x2xf64>) -> tensor<2x2xf64>
  %2 = "toy.constant"() { value: dense<tensor<4xf64>, [1., 2., 3., 4.]> }
                       : () -> tensor<4xf64>
  %3 = "toy.reshape"(%2) : (tensor<4xf64>) -> tensor<2x2xf64>
  %4 = "toy.generic_call"(%1, %3) {callee: @multiply_transpose}
                       : (tensor<2x2xf64>, tensor<2x2xf64>) -> tensor<*xf64>
  "toy.print"(%4) : (tensor<*xf64>) -> ()
  "toy.return"() : () -> ()
}
```

```
$ bin/toy-ch5 -emit=mlir example.toy
```

A Toy Dialect: Constant Operation

- Provide a summary and description for this operation.
    - This can be used to auto-generate documentation of the operations within our dialect.

- Arguments and results specified with “constraints” on the type
    - Argument is attribute/operand

- Additional verification not covered by constraints/traits/etc.

```
def ConstantOp : Toy_Op<"constant"> {
  // Provide a summary and description for this operation.
  let summary = "constant operation";
  let description = [{
    Constant operation turns a literal into an SSA value.
    The data is attached to the operation as an attribute.
    %0 = "toy.constant"() {
      value = dense<[1.0, 2.0]> : tensor<2xf64>
    } : () -> tensor<2x3xf64>
  }];

  // The constant operation takes an attribute as the only
  // input. `F64ElementsAttr` corresponds to a 64-bit
  // floating-point ElementsAttr.
  let arguments = (ins F64ElementsAttr:$value);

  // The constant operation returns a single value of type
  // F64Tensor: it is a 64-bit floating-point TensorType.
  let results = (outs F64Tensor);

  // Additional verification logic: here we invoke a static
  // `verify` method in a C++ source file. This codeblock is
  // executed inside of ConstantOp::verify, so we can use
  // `this` to refer to the current operation instance.
  let verifier = [{ return ::verify(*this); }];
}
```

A Toy Dialect: Constant Operation

- Provide a summary and description for this operation.
  - This can be used to auto-generate documentation of the operations within our dialect.
- Arguments and results specified with “constraints” on the type
  - Argument is attribute/operand
- Additional verification not covered by constraints/traits/etc.

```
def ConstantOp : Toy_Op<"constant"> {
  // Provide a summary and description for this operation.
  let summary = "constant operation";
  let description = [{
    Constant operation turns a literal into an SSA value.
    The data is attached to the operation as an attribute.
    %0 = "toy.constant"() {
      value = dense<[1.0, 2.0]> : tensor<2xf64>
    } : () -> tensor<2x3xf64>
  }];

  // The constant operation takes an attribute as the only
  // input. `F64ElementsAttr` corresponds to a 64-bit
  // floating-point ElementsAttr.
  let arguments = (ins F64ElementsAttr:$value);

  // The constant operation returns a single value of type
  // F64Tensor: it is a 64-bit floating-point TensorType.
  let results = (outs F64Tensor);

  // Additional verification logic: here we invoke a static
  // `verify` method in a C++ source file. This codeblock is
  // executed inside of ConstantOp::verify, so we can use
  // `this` to refer to the current operation instance.
  let verifier = [{ return ::verify(*this); }];
}
```

A Toy Dialect: Constant Operation

- Provide a summary and description for this operation.
  - This can be used to auto-generate documentation of the operations within our dialect.
- Arguments and results specified with “constraints” on the type
  - Argument is attribute/operand
- Additional verification not covered by constraints/traits/etc.

```
def ConstantOp : Toy_Op<"constant"> {
  // Provide a summary and description for this operation.
  let summary = "constant operation";
  let description = [{
    Constant operation turns a literal into an SSA value.
    The data is attached to the operation as an attribute.
    %0 = "toy.constant"() {
      value = dense<[1.0, 2.0]> : tensor<2xf64>
    } : () -> tensor<2x3xf64>
  }];

  // The constant operation takes an attribute as the only
  // input. `F64ElementsAttr` corresponds to a 64-bit
  // floating-point ElementsAttr.
  let arguments = (ins F64ElementsAttr:$value);

  // The constant operation returns a single value of type
  // F64Tensor: it is a 64-bit floating-point TensorType.
  let results = (outs F64Tensor);

  // Additional verification logic: here we invoke a static
  // `verify` method in a C++ source file. This codeblock is
  // executed inside of ConstantOp::verify, so we can use
  // `this` to refer to the current operation instance.
  let verifier = [{ return ::verify(*this); }];
}
```

Google

A Toy Dialect: Constant Operation

C++ Generated Code from TableGen:

```
class ConstantOp
  : public mlir::Op<ConstantOp, mlir::OpTrait::ZeroOperands,
                    mlir::OpTrait::OneResult> {
 public:
  using Op::Op;
  static llvm::StringRef getOperationName() {
    return "toy.constant";
  }
  mlir::DenseElementsAttr value();
  mlir::LogicalResult verify();
  static void build(mlir::OpBuilder &builder,
                    mlir::OperationState &state,
                    mlir::Type result,
                    mlir::DenseElementsAttr value);
};
```

```
def ConstantOp : Toy_Op<"constant"> {
  // Provide a summary and description for this operation.
  let summary = "constant operation";
  let description = [{
    Constant operation turns a literal into an SSA value.
    The data is attached to the operation as an attribute.
    %0 = "toy.constant"() {
      value = dense<[1.0, 2.0]> : tensor<2xf64>
    } : () -> tensor<2x3xf64>
  }];

  // The constant operation takes an attribute as the only
  // input. `F64ElementsAttr` corresponds to a 64-bit
  // floating-point ElementsAttr.
  let arguments = (ins F64ElementsAttr:$value);

  // The constant operation returns a single value of type
  // F64Tensor: it is a 64-bit floating-point TensorType.
  let results = (outs F64Tensor);

  // Additional verification logic: here we invoke a static
  // `verify` method in a C++ source file. This codeblock is
  // executed inside of ConstantOp::verify, so we can use
  // `this` to refer to the current operation instance.
  let verifier = [{ return ::verify(*this); }];
}
```

A (Robust) Toy Dialect

After registration, operations are now fully verified.

```
$ cat test/Examples/Toy/Ch3/invalid.mlir
func @main() {
  "toy.print"()  : () -> ()
}
$ build/bin/toyc-ch3 test/Examples/Toy/Ch3/invalid.mlir -emit=mlir
loc("test/invalid.mlir":2:8): error: 'toy.print' op requires a single operand
```

Google

Toy High-Level Transformations

Traits

- Mixins that define additional functionality, properties, and verification on an Attribute/Operation/Type
- Presence is checked opaquely by analyses/transformations
- Examples (for operations):
    - Commutative
    - Terminator: if the operation terminates a block
    - ZeroOperand/SingleOperand/HasNOperands

https://mlir.llvm.org/docs/Traits/

Interfaces

- Abstract classes to manipulate MLIR entities opaquely
  - Group of methods with an implementation provided by an attribute/dialect/operation/type
  - Do not rely on C++ inheritance, similar to interfaces in C#
- Cornerstone of MLIR extensibility and pass reusability
  - Interfaces frequently initially defined to satisfy the need of transformations
  - Dialects implement interfaces to enable and reuse generic transformations
- Examples (for operations):
  - CallOp/CallableOp (callgraph modeling)
  - LoopLike
  - Side Effects

https://mlir.llvm.org/docs/Interfaces/

Example Problem: Shape Inference

- Ensure all dynamic toy arrays become statically shaped
  - CodeGen/Optimization become a bit easier
  - Tutorial friendly

```
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>) -> ()
}
```

Example Problem: Shape Inference

- Ensure all dynamic toy arrays become statically shaped
  - CodeGen/Optimization become a bit easier
  - Tutorial friendly
- Interprocedural shape propagation analysis?

```
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>) -> ()
}
```

Example Problem: Shape Inference

- Ensure all dynamic toy arrays become statically shaped
  - CodeGen/Optimization become a bit easier
  - Tutorial friendly
- Interprocedural shape propagation analysis?
- Function specialization?

```
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>) -> ()
}
```

Example Problem: Shape Inference

- Ensure all dynamic toy arrays become statically shaped
  - CodeGen/Optimization become a bit easier
  - Tutorial friendly
- Interprocedural shape propagation analysis?
- Function specialization?
- Inline everything!

```
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>) -> ()
}
```

Google

Example Problem: Inlining Literally Everything

MLIR provides an inlining pass which defines an interface, Toy dialect just needs to implement the inliner interface:

- Define the legality of inlining Toy operations
- Expose “`toy.generic_call`” to the callgraph

Google

https://mlir.llvm.org/docs/Tutorials/Toy/Ch-4/#inlining

Example Problem: Inlining Literally Everything

This class defines the interface for handling inlining with Toy operations. We simplify inherit from the base interface class and override the necessary methods.

```
struct ToyInlinerInterface : public DialectInlinerInterface {
  using DialectInlinerInterface::DialectInlinerInterface;

  bool isLegalToInline(Operation *, Region *,
                       BlockAndValueMapping &) const final {
    return true;
  }

  void handleTerminator(
      Operation *op, ArrayRef<Value> valuesToRepl) const final {
    // Only "toy.return" needs to be handled here.
    ReturnOp returnOp = cast<ReturnOp>(op);
    for (auto it : llvm::enumerate(returnOp.getOperands()))
      valuesToRepl[it.index()].replaceAllUsesWith(it.value());
  }

  Operation *materializeCallConversion(
      OpBuilder &builder, Value input, Type resultType,
      Location conversionLoc) const final {
    return builder.create<CastOp>(conversionLoc,
                                  resultType, input);
  }
};
```

https://mlir.llvm.org/docs/Tutorials/Toy/Ch-4/#inlining

Example Problem: Inlining Literally Everything

This class defines the interface for handling inlining with Toy operations. We simplify inherit from the base interface class and override the necessary methods.

This hook checks to see if the given operation is legal to inline into the given region. For Toy this hook can simply return true, as all Toy operations are inlinable.

```
struct ToyInlinerInterface : public DialectInlinerInterface {
  using DialectInlinerInterface::DialectInlinerInterface;

  bool isLegalToInline(Operation *, Region *,
                       BlockAndValueMapping &) const final {
    return true;
  }

  void handleTerminator(
      Operation *op, ArrayRef<Value> valuesToRepl) const final {
    // Only "toy.return" needs to be handled here.
    ReturnOp returnOp = cast<ReturnOp>(op);
    for (auto it : llvm::enumerate(returnOp.getOperands()))
      valuesToRepl[it.index()].replaceAllUsesWith(it.value());
  }

  Operation *materializeCallConversion(
      OpBuilder &builder, Value input, Type resultType,
      Location conversionLoc) const final {
    return builder.create<CastOp>(conversionLoc,
                                  resultType, input);
  }
};
```

https://mlir.llvm.org/docs/Tutorials/Toy/Ch-4/#inlining

Google

Example Problem: Inlining Literally Everything

This class defines the interface for handling inlining with Toy operations. We simplify inherit from the base interface class and override the necessary methods.

This hook checks to see if the given operation is legal to inline into the given region. For Toy this hook can simply return true, as all Toy operations are inlinable.

This hook is called when a terminator operation has been inlined. The only terminator that we have in the Toy dialect is the return operation(toy.return). We handle the return by replacing the values previously returned by the call operation with the operands of the return.

```
struct ToyInlinerInterface : public DialectInlinerInterface {
  using DialectInlinerInterface::DialectInlinerInterface;

  bool isLegalToInline(Operation *, Region *,
                       BlockAndValueMapping &) const final {
    return true;
  }

  void handleTerminator(
      Operation *op, ArrayRef<Value> valuesToRepl) const final {
    // Only "toy.return" needs to be handled here.
    ReturnOp returnOp = cast<ReturnOp>(op);
    for (auto it : llvm::enumerate(returnOp.getOperands()))
      valuesToRepl[it.index()].replaceAllUsesWith(it.value());
  }

  Operation *materializeCallConversion(
      OpBuilder &builder, Value input, Type resultType,
      Location conversionLoc) const final {
    return builder.create<CastOp>(conversionLoc,
                                  resultType, input);
  }
};
```

https://mlir.llvm.org/docs/Tutorials/Toy/Ch-4/#inlining

Google

Example Problem: Inlining Literally Everything

This class defines the interface for handling inlining with Toy operations. We simplify inherit from the base interface class and override the necessary methods.

This hook checks to see if the given operation is legal to inline into the given region. For Toy this hook can simply return true, as all Toy operations are inlinable.

This hook is called when a terminator operation has been inlined. The only terminator that we have in the Toy dialect is the return operation(toy.return). We handle the return by replacing the values previously returned by the call operation with the operands of the return.

Attempts to materialize a conversion for a type mismatch between a call from this dialect, and a callable region. This method should generate an operation that takes 'input' as the only operand, and produces a single result of 'resultType'. If a conversion can not be generated, nullptr should be returned.

```
struct ToyInlinerInterface : public DialectInlinerInterface {
  using DialectInlinerInterface::DialectInlinerInterface;

  bool isLegalToInline(Operation *, Region *,
                       BlockAndValueMapping &) const final {
    return true;
  }

  void handleTerminator(
      Operation *op, ArrayRef<Value> valuesToRepl) const final {
    // Only "toy.return" needs to be handled here.
    ReturnOp returnOp = cast<ReturnOp>(op);
    for (auto it : llvm::enumerate(returnOp.getOperands()))
      valuesToRepl[it.index()].replaceAllUsesWith(it.value());
  }

  Operation *materializeCallConversion(
      OpBuilder &builder, Value input, Type resultType,
      Location conversionLoc) const final {
    return builder.create<CastOp>(conversionLoc,
                                  resultType, input);
  }
};
```

https://mlir.llvm.org/docs/Tutorials/Toy/Ch-4/#inlining

Example Problem: Inlining Literally Everything

- Operation interface for callgraph
  - Traits and Interfaces are added right after the mnemonic
  - `DeclareOpInterfaceMethods` implicitly adds interface method declarations to the op class

```
def GenericCallOp : Toy_Op<"generic_call",
    [DeclareOpInterfaceMethods<CallOpInterface>]> {
  // The generic call operation takes a symbol reference
  // attribute as the callee, and inputs for the call.
  let arguments = (ins
     FlatSymbolRefAttr:$callee,
     Variadic<F64Tensor>:$inputs
  );

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

Google

https://mlir.llvm.org/docs/Tutorials/Toy/Ch-4/#inlining

Example Problem: Inlining Literally Everything

```
/// Return the callee of the generic call operation, this is
/// required by the call interface.
CallInterfaceCallable GenericCallOp::getCallableForCallee()
{
  // `calleeAttr` is an auto-generated method that returns
  // the attribute for `callee` defined in ODS.
  return calleeAttr();
}

/// Get the argument operands to the called function, this
/// is required by the call interface.
Operation::operand_range GenericCallOp::getArgOperands() {
  // `inputs` is an auto-generated method that returns the
  // operands corresponding to the `inputs` argument in ODS.
  return inputs();
}
```

```
def GenericCallOp : Toy_Op<"generic_call",
    [DeclareOpInterfaceMethods<CallOpInterface>]> {
  // The generic call operation takes a symbol reference
  // attribute as the callee, and inputs for the call.
  let arguments = (ins
     FlatSymbolRefAttr:$callee,
     Variadic<F64Tensor>:$inputs
  );

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

https://mlir.llvm.org/docs/Tutorials/Toy/Ch-4/#inlining

Example: Inlining Literally Everything

```
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>) -> ()
}
func @main() {
  %0 = "toy.constant"() { value: dense<[[1., 2.], [3., 4.]]> : tensor<2x2xf64> }
                         : () -> tensor<2x2xf64>
  %1 = "toy.reshape"(%0) : (tensor<2x2xf64>) -> tensor<2x2xf64>
  %2 = "toy.constant"() { value: dense<tensor<4xf64>, [1., 2., 3., 4.]> }
                         : () -> tensor<4xf64>
  %3 = "toy.reshape"(%2) : (tensor<4xf64>) -> tensor<2x2xf64>
  %4 = "toy.generic_call"(%1, %3) {callee: @multiply_transpose}
                         : (tensor<2x2xf64>, tensor<2x2xf64>) -> tensor<*xf64>
  "toy.print"(%4) : (tensor<*xf64>) -> ()
  “toy.return"() : () -> ()
}
```

Example: Inlining Literally Everything

```
func @main() {
  %0 = "toy.constant"() { value: dense<[[1., 2.], [3., 4.]]> : tensor<2x2xf64> }
                         : () -> tensor<2x2xf64>
  %1 = "toy.reshape"(%0) : (tensor<2x2xf64>) -> tensor<2x2xf64>
  %2 = "toy.constant"() { value: dense<tensor<4xf64>, [1., 2., 3., 4.]> }
                         : () -> tensor<4xf64>
  %3 = "toy.reshape"(%2) : (tensor<4xf64>) -> tensor<2x2xf64>

  %4 = "toy.cast"(%3) : (tensor<2x2xf64>) -> tensor<*xf64>
  %5 = "toy.cast"(%1) : (tensor<2x2xf64>) -> tensor<*xf64>
  %6 = "toy.transpose"(%4) : (tensor<*xf64>) -> tensor<*xf64>
  %7 = "toy.transpose"(%5) : (tensor<*xf64>) -> tensor<*xf64>
  %8 = "toy.mul"(%6, %7) : (tensor<*xf64>, tensor<*xf64>) -> tensor<*xf64>

  "toy.print"(%8) : (tensor<*xf64>) -> ()
  “toy.return"() : () -> ()
}
```

Google

Example: Intraprocedural Shape Inference

1. Build a worklist containing all the operations that return a dynamically shaped tensor

2. Iterate on the worklist:
    - Find an operation to process: the next ready operation in the worklist has all of its arguments non-generic
    - If no operation is found, break out of the loop
    - Remove the operation from the worklist
    - **Infer the shape of its output from the argument types**
      => Using an interface to make the pass independent of the dialects and reusable.

3. If the worklist is empty, the algorithm succeeded

https://mlir.llvm.org/docs/Tutorials/Toy/Ch-4/#intraprocedural-shape-inference

Example: Shape Inference Interface

- Operation Interface
  - Description

```
def ShapeInferenceOpInterface : OpInterface<"ShapeInference"> {
  let description = [{
    Interface to access a registered method to infer the
    return types for an operation that can be used during
    type inference.
  }];
}
```

Google

https://mlir.llvm.org/docs/Tutorials/Toy/Ch-4/#intraprocedural-shape-inference

Example: Shape Inference Interface

- Operation Interface
  - Description
  - Methods
    - Summary
    - Return Type
    - Name
    - Arguments
    - Default Implementation

```
def ShapeInferenceOpInterface : OpInterface<"ShapeInference"> {
  let description = [{
    Interface to access a registered method to infer the
    return types for an operation that can be used during
    type inference.
  }];

  let methods = [
    InterfaceMethod<"Infer and set the output shape for the"
                    "current operation.",
                    "void", "inferShapes">
  ];
}
```

https://mlir.llvm.org/docs/Tutorials/Toy/Ch-4/#intraprocedural-shape-inference

Example: Shape Inference Interface

```
def MulOp : Toy_Op<"mul",
    [DeclareOpInterfaceMethods<ShapeInferenceOpInterface>]> {
  let arguments = (ins F64Tensor:$lhs, F64Tensor:$rhs);
  let results = (outs F64Tensor);
}

/// Infer the output shape of the MulOp, this is required by
/// the shape inference interface.
void MulOp::inferShapes() {
  getResult().setType(lhs().getType());
}
```

```
def ShapeInferenceOpInterface : OpInterface<"ShapeInference"> {
  let description = [{
    Interface to access a registered method to infer the
    return types for an operation that can be used during
    type inference.
  }];

  let methods = [
    InterfaceMethod<"Infer and set the output shape for the"
                    "current operation.",
                    "void", "inferShapes">
  ];
}
```

Google

https://mlir.llvm.org/docs/Tutorials/Toy/Ch-4/#intraprocedural-shape-inference

Example: Shape Inference Pass

```
func @main() {
  %0 = "toy.constant"() { value: dense<[[1., 2.], [3., 4.]]> : tensor<2x2xf64> }
                         : () -> tensor<2x2xf64>
  %1 = "toy.reshape"(%0) : (tensor<2x2xf64>) -> tensor<2x2xf64>
  %2 = "toy.constant"() { value: dense<tensor<4xf64>, [1., 2., 3., 4.]> }
                         : () -> tensor<4xf64>
  %3 = "toy.reshape"(%2) : (tensor<4xf64>) -> tensor<2x2xf64>

  %4 = "toy.transpose"(%3) : (tensor<2x2xf64>) -> tensor<2x2xf64>
  %5 = "toy.transpose"(%1) : (tensor<2x2xf64>) -> tensor<2x2xf64>
  %6 = "toy.mul"(%4, %5) : (tensor<2x2xf64>, tensor<2x2xf64>) -> tensor<2x2xf64>

  "toy.print"(%6) : (tensor<2x2xf64>) -> ()
  “toy.return"() : () -> ()
}
```

Google

Dialect Lowering

All the way to LLVM!

Towards CodeGen

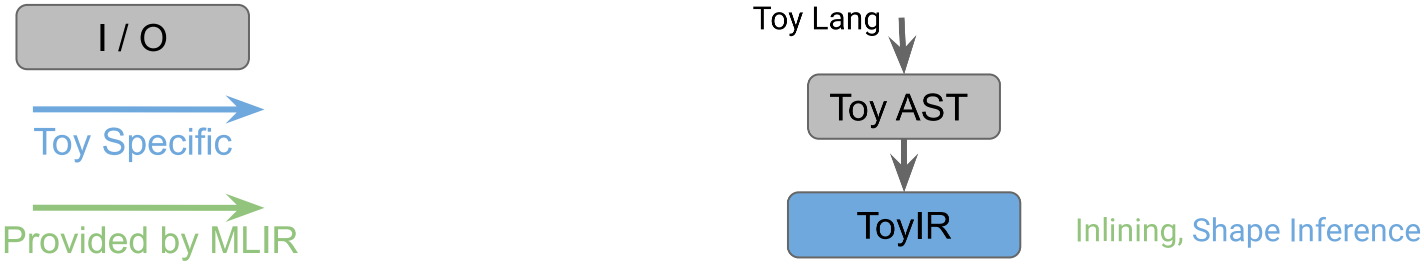
Let’s make Toy executable!

MLIR does not have a code generator for target assembly...

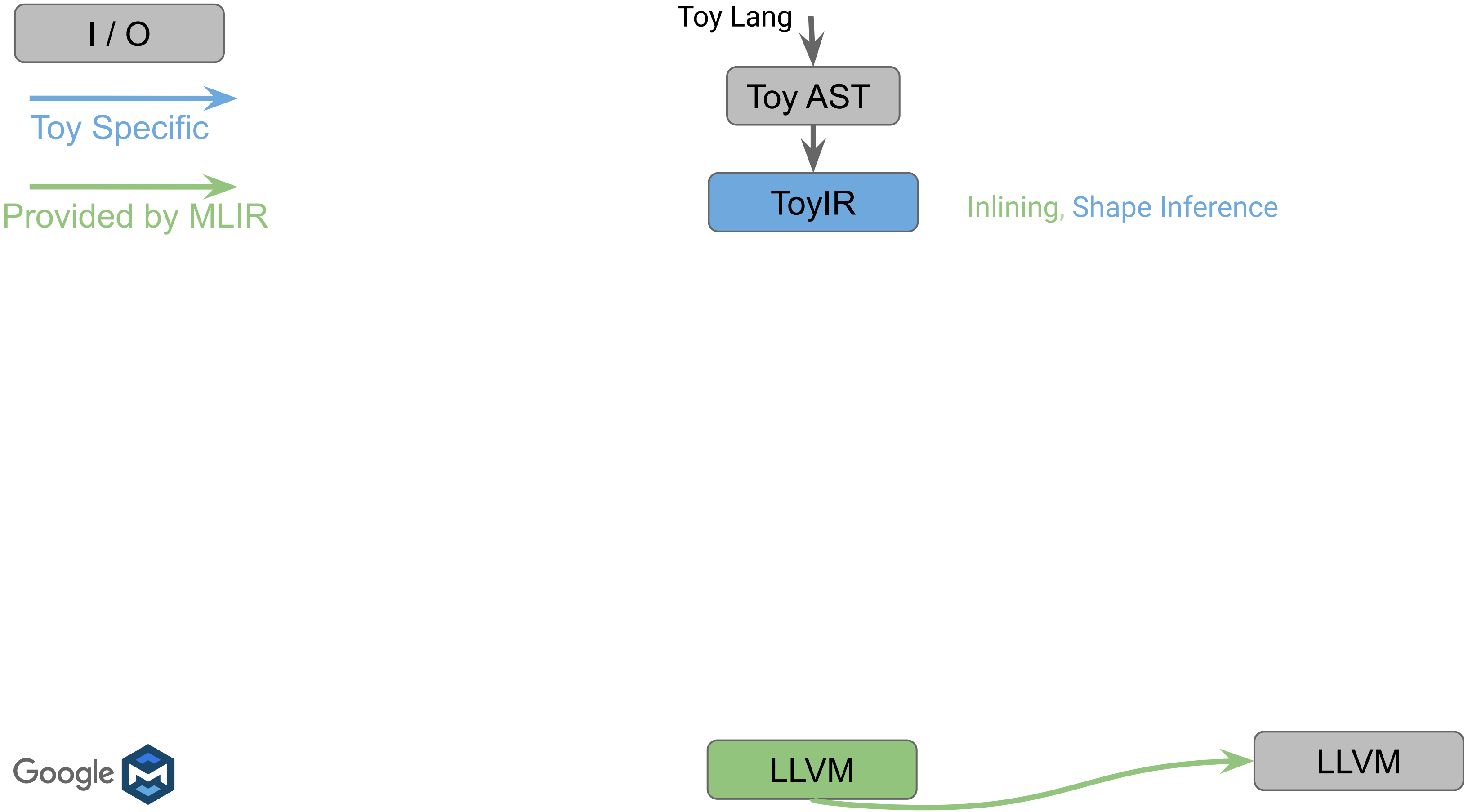
Luckily, LLVM does! And we have an LLVM dialect in MLIR.

General Outline of Dialects, Lowerings, Transformations

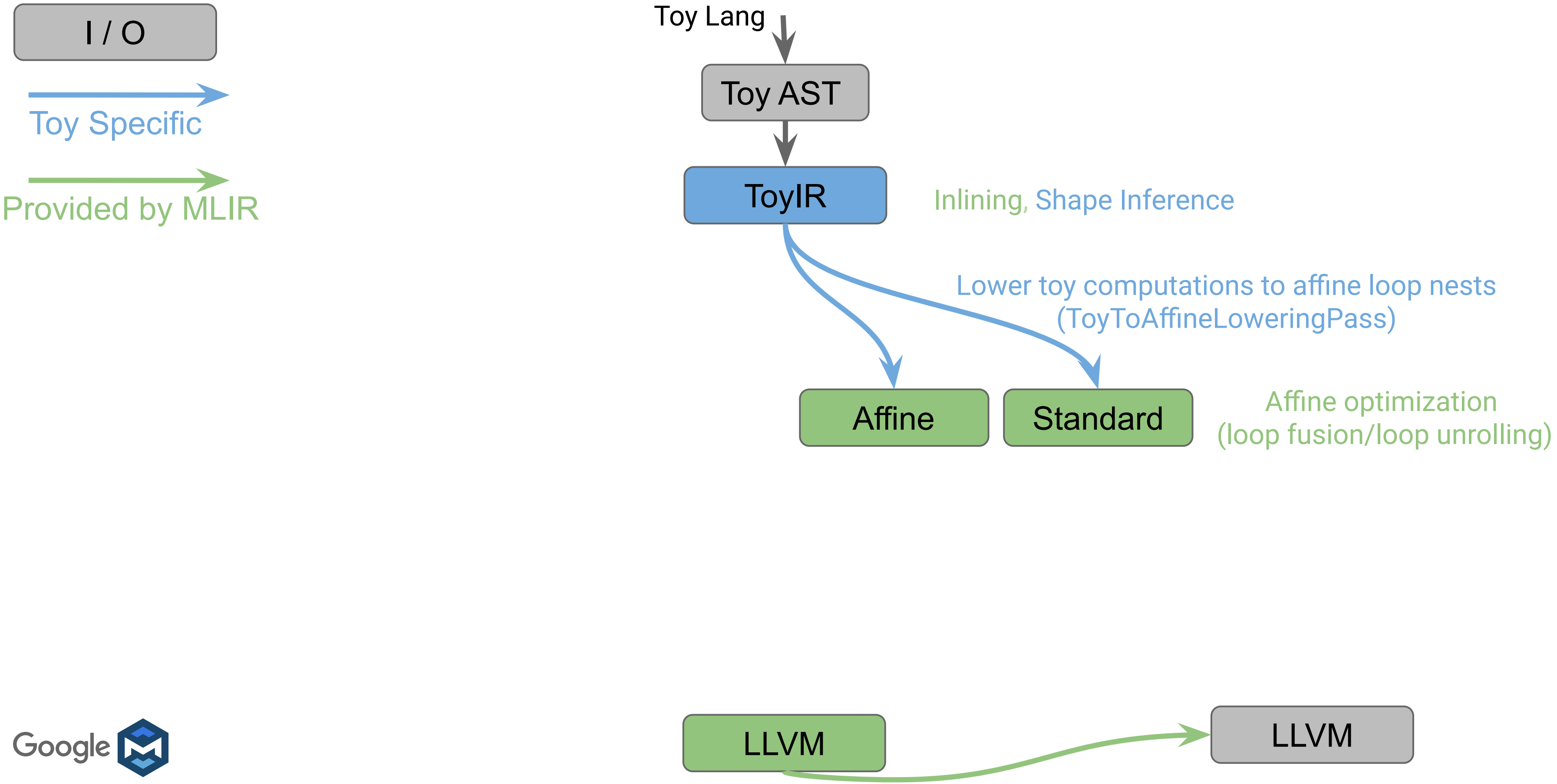
General Outline of Dialects, Lowerings, Transformations



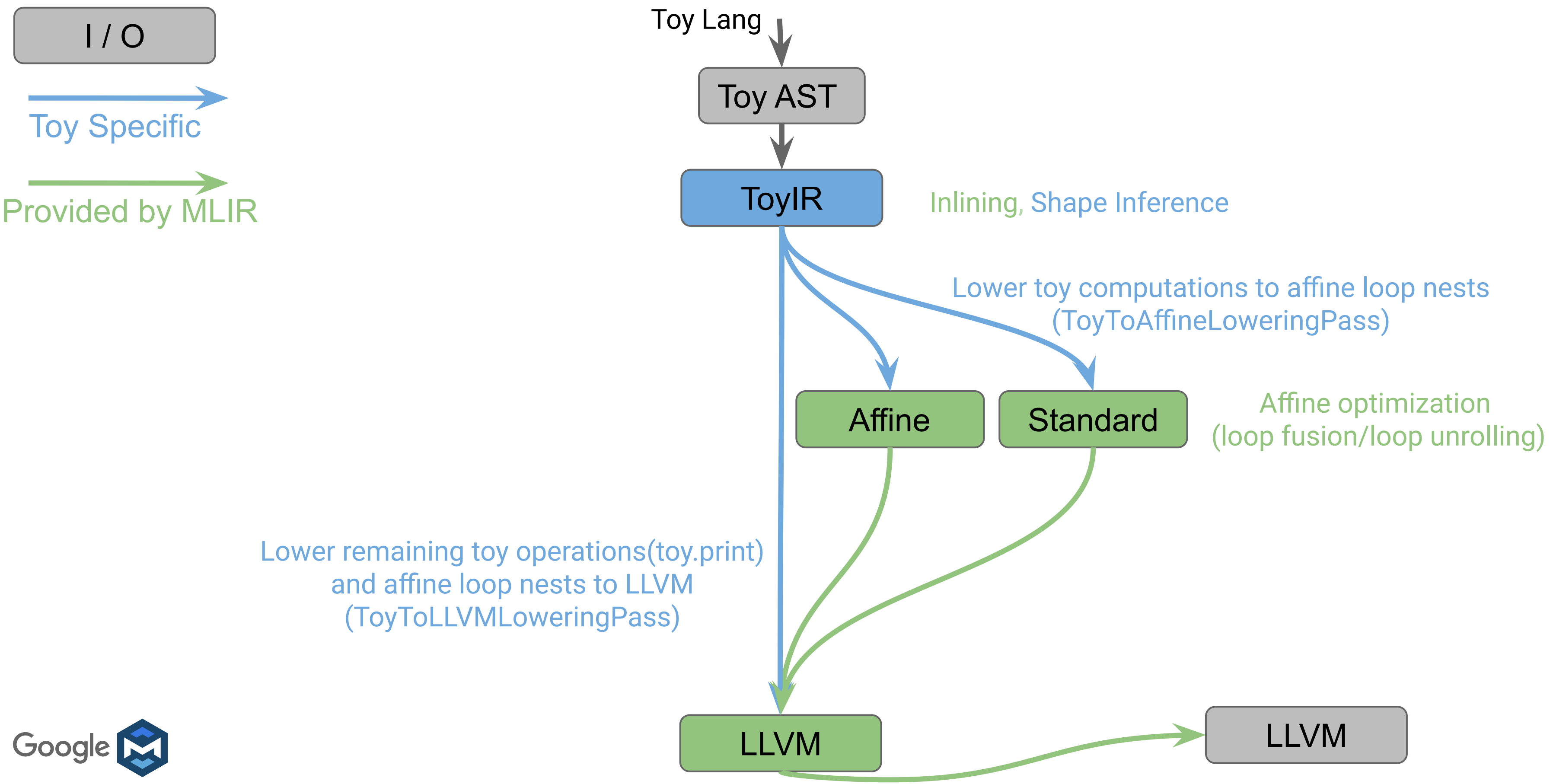
General Outline of Dialects, Lowerings, Transformations



General Outline of Dialects, Lowerings, Transformations

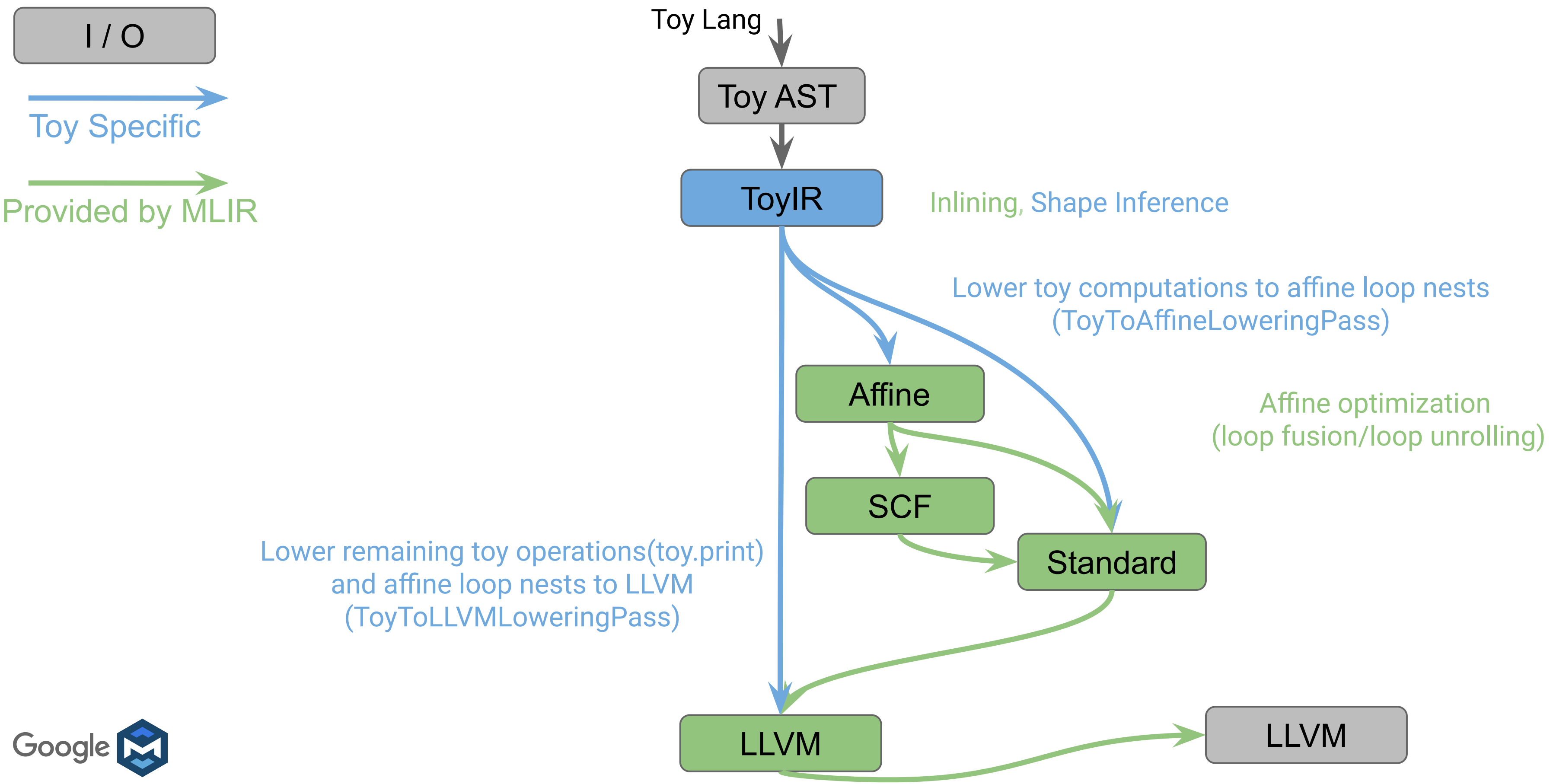


General Outline of Dialects, Lowerings, Transformations



Google

General Outline of Dialects, Lowerings, Transformations



Google

Lowering with 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

https://mlir.llvm.org/docs/DialectConversion/

Dialect Conversion: ConversionTarget

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

Google

https://mlir.llvm.org/docs/Tutorials/Toy/Ch-5/#conversion-target

Dialect Conversion: ConversionTarget

- Define the dialects or operations that are legal for the conversion.
  - Legality can be dynamic

```
// The first thing to define is the conversion target. This
// will define the final target for this lowering.
mlir::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` and `Standard`
// dialects.
target.addLegalDialect<mlir::AffineDialect,
                       mlir::StandardOpsDialect>();
```

Google

https://mlir.llvm.org/docs/Tutorials/Toy/Ch-5/#conversion-target

Dialect Conversion: ConversionTarget

- Define the dialects or operations that are legal for the conversion
    - Legality can be dynamic
- Define dialects or operations that are illegal, i.e. required to be converted

```
// The first thing to define is the conversion target. This
// will define the final target for this lowering.
mlir::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` and `Standard`
// dialects.
target.addLegalDialect<mlir::AffineDialect,
                       mlir::StandardOpsDialect>();

// 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*.
target.addIllegalDialect<ToyDialect>();
target.addLegalOp<PrintOp>();
```


https://mlir.llvm.org/docs/Tutorials/Toy/Ch-5/#conversion-target

Dialect Conversion: Operation Conversion

- Convert illegal ops into legal ops using Dag->Dag rewrite patterns
  - Patterns are highly composable with high reuse between different conversions
- Transitive conversion: [bar.add -> baz.add, baz.add -> foo.add]
  - Patterns don't need to generate strictly legal IR, can rely on other patterns

Dialect Conversion: Operation Conversion

- Specified via ConversionPattern/RewritePattern
  - Root operation type (Optional)
  - Benefit of applying the pattern

```
/// Lower the `toy.transpose` operation to an affine loop nest.
struct TransposeOpLowering : public mlir::ConversionPattern {
  TransposeOpLowering(mlir::MLIRContext *ctx)
      : mlir::ConversionPattern(TransposeOp::getOperationName(),
                                /*benefit=*/1, ctx) {}
};
```

Google

Dialect Conversion: Operation Conversion

- Specified via ConversionPattern, or RewritePattern depending on context
  - Root operation type (Optional)
  - Benefit of applying the pattern
- Provide a method to match and rewrite a given root operation

```
struct TransposeOpLowering : public mlir::ConversionPattern {
  /// Match and rewrite the given `toy.transpose` operation,
  /// with the given operands that have been remapped from
  /// `tensor<...>` to `memref<...>`.
  mlir::LogicalResult
  matchAndRewrite(mlir::Operation *op,
                  llvm::ArrayRef<mlir::Value> operands,
                  mlir::ConversionPatternRewriter &rewriter)
    const final {
    // Returns `mlir::success()` if a match was successful
    // and the pattern was applied, `mlir::failure()`
    // otherwise.
  }
};
```

Dialect Conversion: Operation Conversion

- Specified via ConversionPattern, or RewritePattern depending on context
  - Root operation type (Optional)
  - Benefit of applying the pattern
- Provide a method to match and rewrite a given root operation
  - Rewrite functionality is driven by a PatternRewriter to notify the pattern driver of IR changes

```
// Call to a helper function that will lower the current
// operation to a set of affine loops. We provide a functor
// that operates on the  remapped operands, as well as the loop
// induction variables for the inner most loop body.
lowerOpToLoops(op, operands, rewriter,
  [loc](mlir::PatternRewriter &rewriter,
        llvm::ArrayRef<mlir::Value> memRefOperands,
        llvm::ArrayRef<mlir::Value> loopIvs) {
    // Generate an adaptor for the remapped operands of the
    // TransposeOp. This allows for using the nice named
    // accessors that are generated by the ODS. This adaptor is
    // automatically provided by the ODS framework.
    TransposeOpAdaptor transposeAdaptor(memRefOperands);
    mlir::Value input = transposeAdaptor.input();

    // Transpose the elements by generating a load from the
    // reverse indices.
    SmallVector<mlir::Value, 2> revIVs(llvm::reverse(loopIvs));
    return rewriter.create<mlir::AffineLoadOp>(loc, input,
                                               revIVs);
});
```

Dialect Conversion: Operation Conversion

- Patterns are collected via OwningRewritePatternList

```
// Now that the conversion target has been defined, we just
// need to provide the set of patterns that will lower the
// Toy operations.
mlir::OwningRewritePatternList patterns;
patterns.insert<..., TransposeOpLowering>(&getContext());
```

Dialect Conversion: Partial Conversion

- Existing operations can fail legalization if not explicitly illegal
- Allows for converting a subset of known illegal operations, without knowing the entire IR

```
void ToyToAffineLoweringPass::runOnFunction() {
  ...

  // With the target and rewrite patterns defined, we can now
  // attempt the conversion. The conversion will signal
  // failure if any of our *illegal*  operations were not
  // converted successfully.
  FuncOp function = getFunction();
  if (mlir::failed(mlir::applyPartialConversion(
                                      function, target, patterns)))
    signalPassFailure();
}
```

Google

https://mlir.llvm.org/docs/Tutorials/Toy/Ch-5/#partial-lowering

Dialect Conversion: Partial Conversion

```
func @main() {
  %0 = "toy.constant"() { value: dense<tensor<4xf64>, [1., 2., 3., 4.]> }
                         : () -> tensor<4xf64>
  %1 = "toy.transpose"(%0) : (tensor<2x2xf64>) -> tensor<2x2xf64>
  %2 = "toy.mul"(%1, %1) : (tensor<2x2xf64>, tensor<2x2xf64>) -> tensor<2x2xf64>
  "toy.print"(%0) : (tensor<2x2xf64>) -> ()
  “toy.return"() : () -> ()
}
```

Google

https://mlir.llvm.org/docs/Tutorials/Toy/Ch-5/#partial-lowering

Partial Lowering and Affine Optimizations

```
func @main() {
  %cst = constant 1.000000e+00 : f64
  %cst_0 = constant 2.000000e+00 : f64
  %cst_1 = constant 3.000000e+00 : f64
  %cst_2 = constant 4.000000e+00 : f64
  %0 = alloc() : memref<2x2xf64>
  %1 = alloc() : memref<2x2xf64>
  affine.store %cst, %1[0, 0] : memref<2x2xf64>
  affine.store %cst_0, %1[0, 1] : memref<2x2xf64>
  affine.store %cst_1, %1[1, 0] : memref<2x2xf64>
  affine.store %cst_2, %1[1, 1] : memref<2x2xf64>
  affine.for %arg0 = 0 to 2 {
    affine.for %arg1 = 0 to 2 {
      %2 = affine.load %1[%arg1, %arg0] : memref<2x2xf64>
      %3 = mulf %2, %2 : f64
      affine.store %3, %0[%arg0, %arg1] : memref<2x2xf64>
    }
  }
  toy.print %0 : memref<2x2xf64>
  dealloc %1 : memref<2x2xf64>
  dealloc %0 : memref<2x2xf64>
  return
}
```

Affine / Polyhedral representation to enable relevant optimizations.

Toy dialect operations cohabit with affine and others in the same function

https://mlir.llvm.org/docs/Tutorials/Toy/Ch-5/#taking-advantage-of-affine-optimization

Dialect Conversion: Full Conversion to LLVM

- Only one conversion necessary, the rest are already provided.

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

LLVMTypeConverter typeConverter(&getContext());
mlir::OwningRewritePatternList patterns;
mlir::populateAffineToStdConversionPatterns(patterns, ctx);
mlir::populateLoopToStdConversionPatterns(patterns, ctx);
mlir::populateStdToLLVMConversionPatterns(typeConverter,
                                          patterns);

// The only remaining operation to lower from the `toy`
// dialect is PrintOp.
patterns.insert<PrintOpLowering>(ctx);

mlir::ModuleOp module = getOperation();
if (mlir::failed(mlir::applyFullConversion(module, target,
                                           patterns)))
  signalPassFailure();
```

Google

https://mlir.llvm.org/docs/Tutorials/Toy/Ch-6/

Exporting MLIR LLVM dialect to LLVM IR

Mapping from LLVM Dialect to LLVM IR:

```
auto llvmModule = mlir::translateModuleToLLVMIR(module);
```

LLVM Dialect:

```
%223 = llvm.mlir.constant(2 : index) : !llvm.i64
%224 = llvm.mul %214, %223 : !llvm.i64
```

LLVM IR:

```
%104 = mul i64 %96, 2
```

Conclusion

Not shown today

- Pass Management,
  - MLIR is multi-threaded!
  - Passes, and pass options.
- Diagnostics Infrastructure
- Adding new Attributes/Types
- Declarative assembly format
- Specifying operation canonicalizations
- Symbols and Symbol Tables
- Heterogeneous compilation
  - In particular GPU (CUDA, RocM, and SPIR-V)

MLIR : Reusable Compiler Abstraction Toolbox

MLIR provides all the infrastructure to build IR and transformations:

- Same infra at each abstraction level
- Investment in toolings has compounding effects

IR design involves multiple tradeoffs

- Iterative process, constant learning experience
- MLIR makes compiler design “agile” (and fun!)

MLIR allows mixing levels of abstraction with non-obvious compounding benefits

- Dialect-to-dialect lowering is easy
- Ops from different dialects can mix in same IR
    - Lowering from “A” to “D” may skip “B” and “C”
- Avoid lowering too early and losing information
    - Help define hard analyses away

No forced IR impedance mismatch

Fresh look at problems

MLIR

https://mlir.llvm.org/

Join the community:

Discourse Forums
Discord Chat
Weekly open meeting
Biweekly newsletter

Thank you!

Questions?