MLIR Tutorial:

Building a Compiler with MLIR

MLIR 4 HPC, 2019

Jacques Pienaar
Google

Sana Damani
Georgia Tech

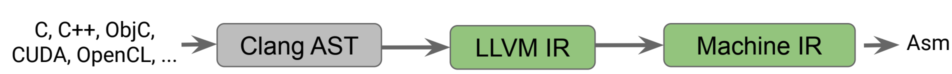
Presenting the work of many people!

Introduction

- ML != Machine Learning in MLIR
- … but Machine Learning is one of first application domains
- And where MLIR started
- … but not what MLIR is limited to :)

Why a new compiler infrastructure?

The LLVM Ecosystem: Clang Compiler



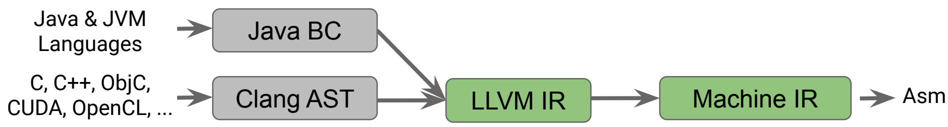
Green boxes are SSA IRs:

- Different levels of abstraction - operations and types are different
- Abstraction-specific optimization at both levels

Progressive lowering:

- Simpler lowering, reuse across other front/back ends

Azul Falcon JVM

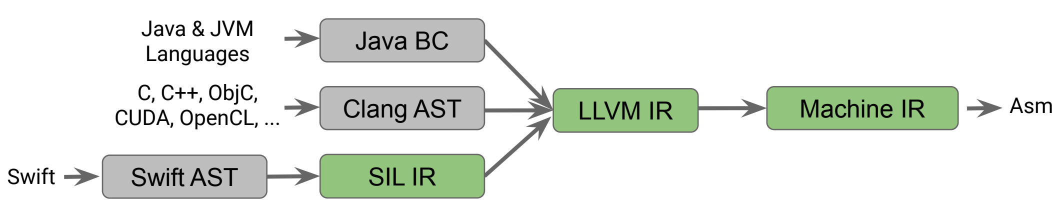


Uses LLVM IR for high level domain specific optimization:

- Encodes information in lots of ways: IR Metadata, well known functions, intrinsics, ...
- Reuses LLVM infrastructure: pass manager, passes like inliner, etc.

"Falcon: An Optimizing Java JIT" - LLVM Developer Meeting Oct'2017

Swift Compiler

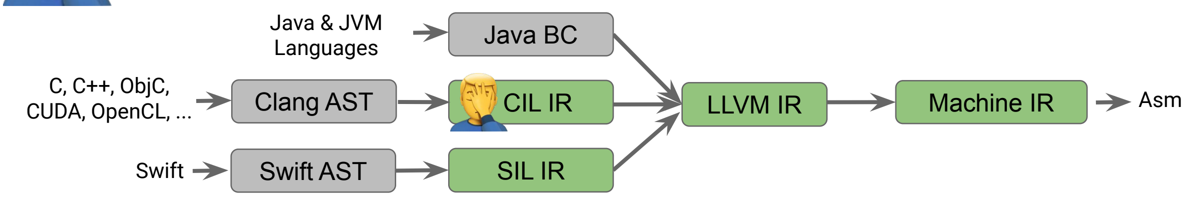


3-address SSA IR with Swift-specific operations and types:

- Domain specific optimizations: generic specialization, devirt, ref count optzns, library-specific optzns, etc
- Dataflow driven type checking passes: e.g. definitive initialization, "static analysis" checks
- Progressive lowering makes each edge simpler!

"Swift's High-Level IR" - LLVM Developer Meeting Oct'2015

A sad aside: Clang should have a CIL!



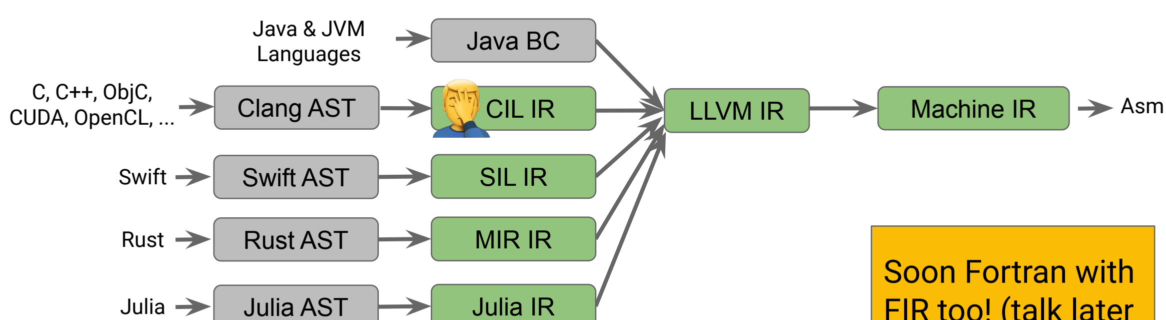
3-address SSA IR with **Clang**-specific operations and types:

- Optimizations for std::vector, std::shared_ptr, std::string, ...
- Better IR for Clang Static Analyzer + Tooling
- Progressive lowering for better reuse

Anyway, back to the talk...

Google

Rust and Julia have things similar to SIL



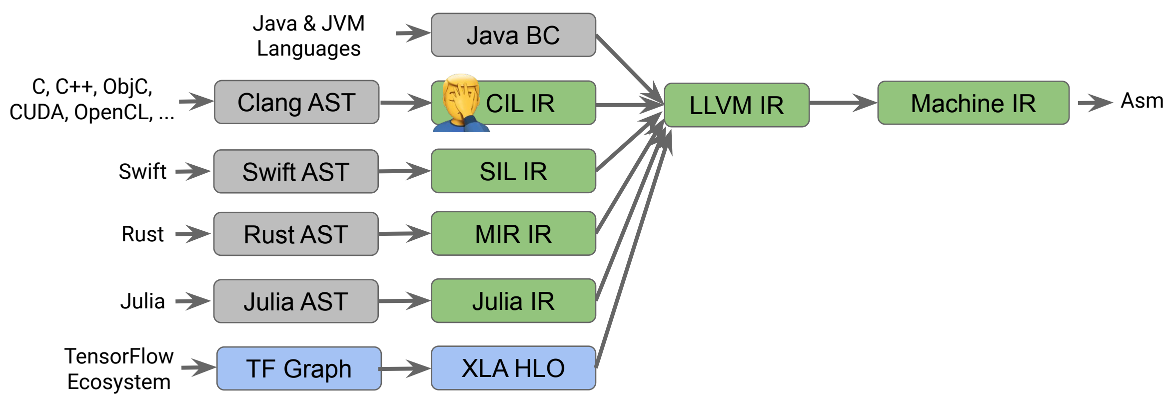
Soon Fortran with FIR too! (talk later today)

- Dataflow driven type checking - e.g. borrow checker
- Domain specific optimizations, progressive lowering

Google

"Introducing MIR": Rust Language Blog, "Julia SSA-form IR": Julia docs

TensorFlow XLA Compiler



- Domain specific optimizations, progressive lowering

Google

"XLA Overview": https://tensorflow.org/xla/overview (video overview)

Domain Specific SSA-based IRs

Great!

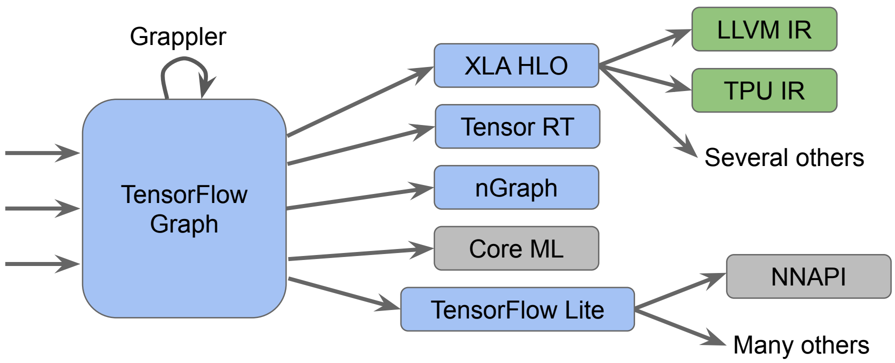
- High-level domain-specific optimizations
- Progressive lowering encourages reuse between levels
- Great location tracking enables flow-sensitive "type checking"

Not great!

- Huge expense to build this infrastructure
- Reimplementation of all the same stuff:
  - pass managers, location tracking, use-def chains, inlining, constant folding, CSE, testing tools, ...
- Innovations in one community don't benefit the others

Google

The TensorFlow compiler ecosystem



Many "Graph" IRs, each with challenges:

- Similar-but-different proprietary technologies: not going away anytime soon
- Fragile, poor UI when failures happen: e.g. poor/no location info, or even crashes
- Duplication of infrastructure at all levels

Google

Goal: Global improvements to TensorFlow infrastructure

But HPC has similar needs so why stop there?

SSA-based designs to generalize and improve ML

- Better side effect modeling and control flow r
- Improve generality of the lowering passes
- Dramatically increase code reuse
- Fix location tracking and other pervasive issues for better user experience

No reasonable existing answers!

- … and we refuse to copy and paste SSA-based optimizers 6 more times!

Google

What is MLIR?

A collection of modular and reusable software components that enables the progressive lowering of operations, to efficiently target hardware in a common way

Google

How is MLIR different?

State of Art Compiler Technology

MLIR is NOT just a common graph serialization format nor is there anything like it

New shared industry abstractions spanning languages ("OMP" dialect?)

Modular & Extensible

From graph representation through optimization to code generation

Mix and match representations to fit problem space

Not opinionated

Choose the level of representation that is right for your device

We want to enable whole new class of compiler research

Google

A toolkit for representing and transforming “code”

Represent and transform IR ⇄⭯⇓

Represent Multiple Levels of

- tree-based IRs (ASTs),
- graph-based IRs (TF Graph, HLO),
- machine instructions (LLVM IR)

IR at the same time

While enabling

Common compiler infrastructure

- location tracking
- richer type system
- common set of conversion passes

And much more

Google

What about HPC?

Could talk about:

- reusing abstractions for parallelism (new parallelism constructs?),
- polyhedral code generations
- stencil abstractions

Instead:

- here to listen what are the problems domain specific abstractions during compilation could lead to much simpler/better world
- Improvements in one community benefiting others

Google

Introduction: a Toy Language

(e.g., enough talking, let's get to code)

Google

Overview

Tour of MLIR by way of implementing basic toy language

- Define a Toy language
- Represent Toy using MLIR
    - Introducing dialect, operations, ODS, verifications
- Attaching semantics to custom operations
- High-level language specific optimizations
    - Pattern rewrite framework
- Writing passes for structure rather than ops
    - Op interfaces for the win
- Lowering to lower-level dialects
    - The road to LLVM IR

Google

[The full tutorial on the MLIRs GitHub]()

Let's Build Our Toy Language

- Mix of scalar and array computations, as well as I/O
- Array shape Inference
- Generic functions
- Very limited set of operators (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;
}
```

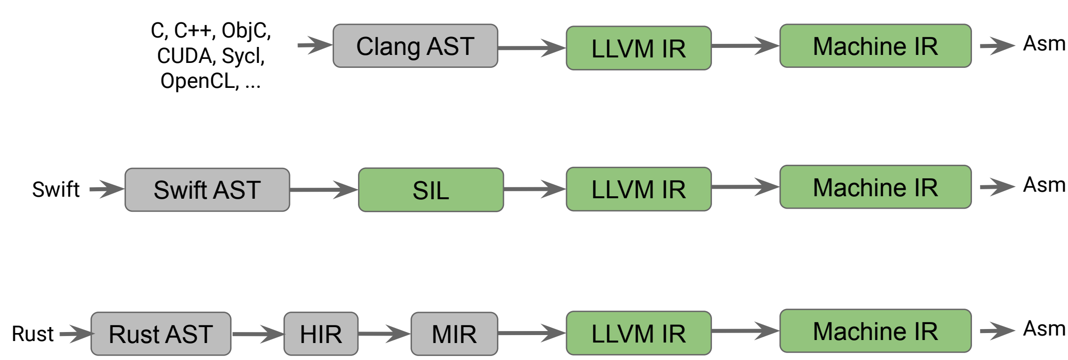
Value-based semantics / SSA

Only 2 builtin functions: print and transpose

Array reshape through explicit variable declaration

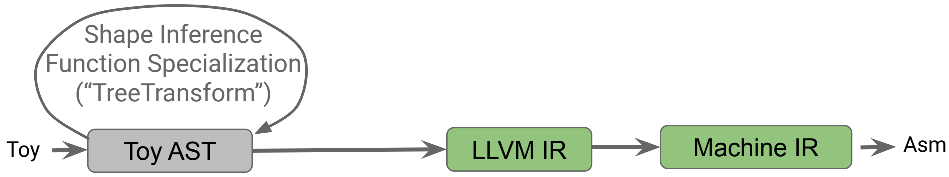
Only float 64s

Existing Successful Model



The Toy Compiler: the “Simpler” Approach of Clang

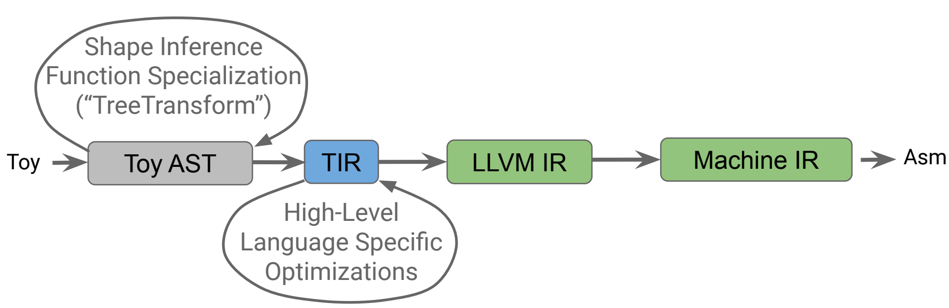
Need to analyze and transform the AST
-> heavy infrastructure!
And is the AST really the most friendly
representation we can get?



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?



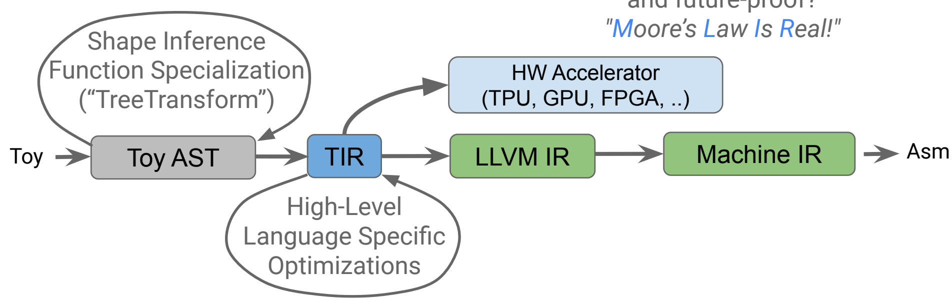
For more optimizations: a custom IR.
Reimplement again all the LLVM infrastructure?

Google

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!"

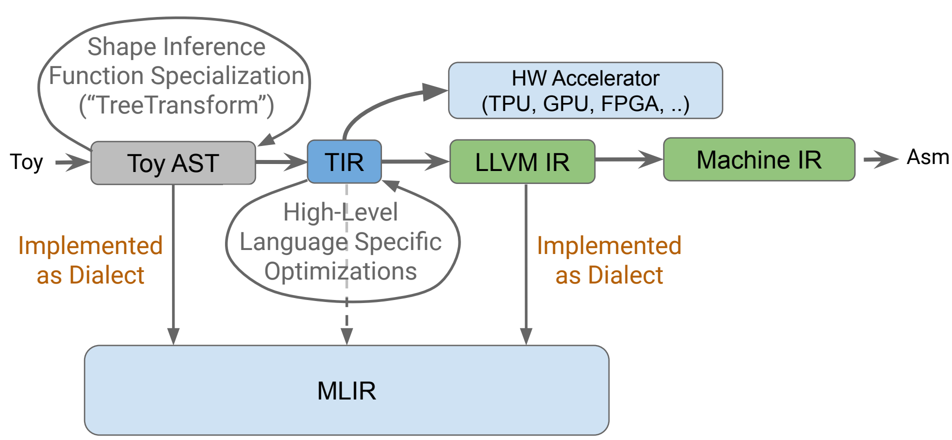


For more optimizations: a custom IR.
Reimplement again all the LLVM infrastructure?

Google

It Is All About The Dialects!

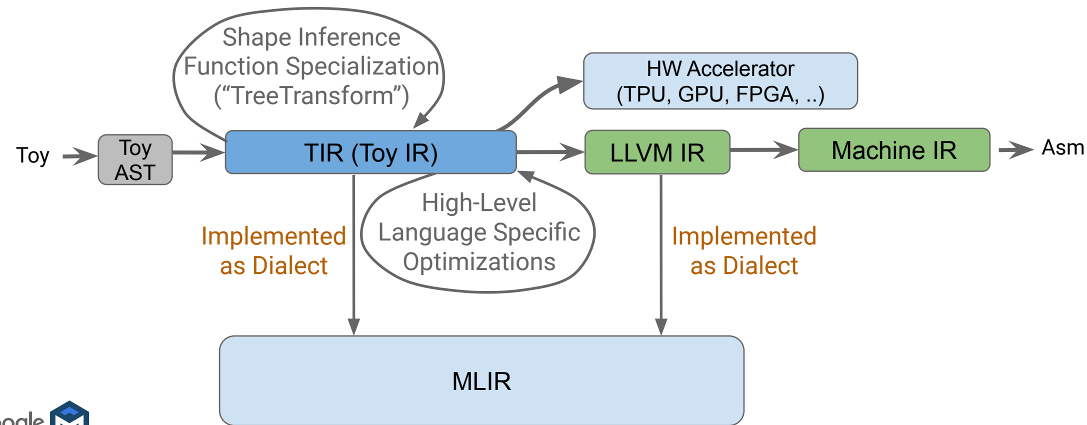
MLIR allows every level to be
represented as a Dialect



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.



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

Google

Example

```
func @some_func(%arg0: !random_dialect<"custom_type">) ->
    !another_dialect<"other_type"> {
  %result = "custom.operation"(%arg0) :
      (!random_dialect<"custom_type">) -> !another_dialect<"other_type">
  return %result : !another_dialect<"other_type">
}
```

Yes: this is a fully valid textual IR module: try round-tripping with *mlir-opt*!

Google

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 return any value

JSON of compiler IR ?!?

Google

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

Google

Look Ma, Something Familiar There...

Dialects are powerful enough that you can wrap LLVM IR within an MLIR Dialect

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

Operations: Regions are Powerful

```
%res:2 = "mydialect.morph"(%input#3) ({ Region A }, { Region B })
  { some.attribute = true, other_attribute = 1.5 } :
    (!mydialect<"custom_type">) -> (!mydialect<"other_type">, !mydialect<"other_type">)
    loc(callsite("foo" at "mysource.cc":10:8))
```

- Regions are list of basic blocks nested alongside an operation.
- Opaque to passes by default, not part of the CFG.
- Similar to a function call but can reference SSA value defined outside.
- SSA value defined inside region don’t escape

Affine Dialect: Simplified Polyhedral Form

- Multidimensional loop nests with affine loops/conditionals/memory references
- Goal: Easier to perform loop transforms (skewing, interchange etc.)
  - See presentation later today!
- Originally baked into the core
  - But not all codegen can use this form, so why not make optional?
- Expanded MLIR core so that it could become "just" a dialect
  - Regions in operations enabled moving it

Google

Region Example: Affine Dialect

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

```
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
}
```

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

Google

https://github.com/tensorflow/mlir/blob/master/g3doc/Dialects/Affine.md

A Toy Dialect

- Dialect & custom types defined in C++
- Dialect can define hooks for
  - type printing and printing
  - constant folding
  - ...
- Custom ops can be defined
  - Programmatically (in C++)
  - Using **O**p **D**efinition **S**pec ->
  - Custom printing, parsing, folding, canonicalization, verification
  - Documentation

```
def TF_LeakyReluOp : TF_UnaryOp<"LeakyRelu",
                     [NoSideEffect, SameValueType]>,
                     Results<(outs TF_Tensor:$output)> {
  let arguments = (ins
    TF_FloatTensor:$value,
    DefaultValuedAttr<F32Attr, "0.2">:$alpha
  );

  let summary = "Leaky ReLU operator";
  let description = [{
    The Leaky ReLU operation takes a tensor and returns
    a new tensor element-wise as follows:
      LeakyRelu(x) = x         if x >= 0
                   = alpha*x   else
  }];

  let constantFolding = ...;
  let canonicalizer = ...;
}
```

A (Robust) Toy Dialect

After registration, operations are now fully checked

```
$ cat test/Examples/Toy/Ch3/invalid.mlir
func @main() {
  "toy.print"()  : () -> ()
}

$ build/bin/toyc-ch3 test/Examples/Toy/Ch3/invalid.mlir -emit=mlir
invalid.mlir:8:8: error: 'toy.print' op requires zero results
  %0 = "toy.print"()  : () -> tensor<2x3xf64>
       ^
```

Toy High-Level Transformations

Interfaces

Motivation

- Decouple transformations from dialect and operation definitions
- Apply transformations across dialects
- Design passes to operate on attributes/structure rather than specific ops
- Prevent code duplication
- Easily extend to new dialects/ops

Google

Interfaces

1. Create an interface
2. Write a pass using the interface
3. Implement interface methods in participating dialects/ops

Google

Types of Interfaces

- Dialect Interfaces: information across operations of a dialect
  - e.g. Inlining

- Operation Interfaces: information specific to operations
  - e.g. Shape Inference

Google

Creating an Inliner Dialect Interface

Create a new Inliner Interface

```
class InlinerInterface
    : public DialectInterfaceCollection<DialectInlinerInterface>
{
public:
  virtual bool isLegalToInline(...) const;
  virtual void handleTerminator(...) const;
}
```

Google

Writing an Opaque Inliner Pass

Create a new Inliner Pass using interface collections

- Use interface collections to obtain a handle to the dialect-specific interface hook to opaquely query interface methods
- Collect all function calls and inline if legal. Also handle block terminators.

```
bool InlinerInterface::isLegalToInline(
    Operation *op, Region *dest, BlockAndValueMapping &valueMapping) const {
  auto *handler = getInterfaceFor(op);
  return handler ? handler->isLegalToInline(op, dest, valueMapping) : false;
}
```

Inlining in Toy

Inherit DialectInlinerInterface within Toy and specialize 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.
    auto returnOp = cast<ReturnOp>(op);

    // Replace the values directly with the return operands.
    assert(returnOp.getNumOperands() == valuesToRepl.size());
    for (const auto &it : llvm::enumerate(returnOp.getOperands()))
      valuesToRepl[it.index()]->replaceAllUsesWith(it.value());
  }
};
```

Inlining in Toy

Add the new interface to Toy Dialect

```
ToyDialect::ToyDialect(mlir::MLIRContext *ctx) : mlir::Dialect("toy", ctx) {
    ...
    addInterfaces<ToyInlinerInterface>();
}
```

Add Inliner Pass to Toy's pass manager

```
mlir::LogicalResult optimize(mlir::ModuleOp module) {
    mlir::PassManager pm(module.getContext());
    ...
    pm.addPass(mlir::createInlinerPass());
    ...
}
```

Google

Operation Interfaces: Shape Inference

- We'll use Shape Inference as an example application of operation interfaces
- We define the following rules for shape inference in Toy
  - A = B + C // A.shape = B.shape = C.shape
  - A = B * C // A.shape = B.rows, C.cols
  - A = transpose(B) // A.shape = B.cols, B.rows

Google

Creating a Shape Inference Interface

Create a ShapeInference OpInterface:

```
def ShapeInferenceOpInterface : OpInterface<"ShapeInference"> {
  let methods = [
    InterfaceMethod<"Infer output shape for the current operation.",
                    "void", "inferShapes", (ins), []>
  ];
}
```

Google

Writing an Opaque Shape Inference Pass

Thanks to operation interfaces, we can write an opaque ShapeInference Pass:

```
while (!opWorklist.empty()) {
    ...
    op = ...
    // Use inferShape if `op` implements the Shape Inference interface
    if (auto shapeOp = dyn_cast<ShapeInference>(op)) {
        shapeOp.inferShapes();
    }
    ...
}
```

Google

Shape Inference in Toy

Specialize interface methods in Toy's op definitions:

```
def AddOp : Toy_Op<"add", [NoSideEffect]> {
  ...
  void inferShapes() {
    getResult()->setType(getOperand(0)->getType());
    return;
  }
}
```

And then add ShapeInference pass to Toy's pass manager.

Pattern-Match and Rewrite

Language Specific Optimizations

```
def no_op(b) {
  return transpose(transpose(b));
}
```

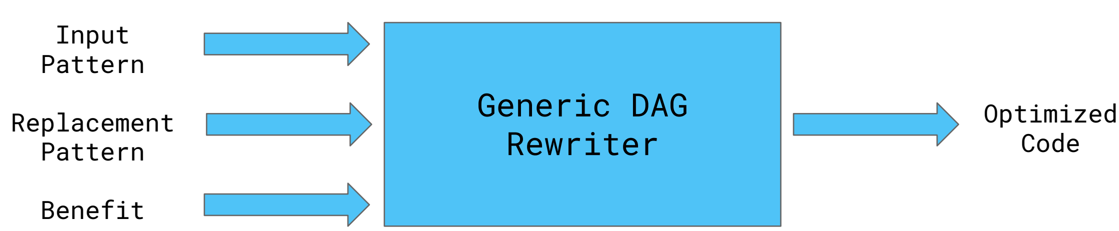
Clang can't optimize away these loops:

```
#define N 100
#define M 100

void sink(void *);
void double_transpose(int A[N][M]) {
  int B[M][N];
  for(int i = 0; i < N; ++i) {
    for(int j = 0; j < M; ++j) {
       B[j][i] = A[i][j];
    }
  }
  for(int i = 0; i < N; ++i) {
    for(int j = 0; j < M; ++j) {
       A[i][j] = B[j][i];
    }
  }
  sink(A);
}
```

Generic DAG Rewriter

- Graph-to-graph rewrites
- Decouple pattern definition and transformation
- Greedy worklist combiner



Pattern Match and Rewrite

`transpose(transpose(x)) => x`

Two ways:

- C++ style using RewritePattern
- Table-driven using DRR

Google

C++ Style using RewritePattern

`transpose(transpose(x)) => x`

Override matchAndRewrite(op):

```
input = op.getOperand();
if (input->getDefiningOp() == TransposeOp)
    x = op->getOperand();
rewriter.replaceOp(op, {x});
```

Register Pattern with Canonicalization Framework

```
void TransposeOp::getCanonicalizationPatterns(...) {
    results.insert<SimplifyRedundantTranspose>(context);
}
```

https://github.com/tensorflow/mlir/blob/master/examples/toy/Ch4/mlir/ToyCombine.cpp#L40-L65
https://github.com/tensorflow/mlir/blob/master/examples/toy/Ch4/mlir/ToyCombine.cpp#L74

Google

Declarative, rule-based pattern-match and rewrite

```
transpose(transpose(x)) => x

// 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)
>;
```

See another example in the repo with Reshape(Reshape(x)):
https://github.com/tensorflow/mlir/blob/master/examples/toy/Ch4/mlir/ToyCombine.td#L43-L44

Declarative, rule-based pattern-match and rewrite

Conditional pattern match:

```
Reshape(x) = x, if input and output shapes are identical
```

Adding Constraints:

```
def TypesAreIdentical : Constraint<CPred<"$0->getType() == $1->getType()">>;
```

Transformation:

```
def ReshapeOptPattern : Pat<(ReshapeOp:$res $arg), (replaceWithValue $arg),\
                            [(TypesAreIdentical $res, $arg)]>;
```

https://github.com/tensorflow/mlir/blob/master/examples/toy/Ch4/mlir/ToyCombine.td#L67-L71

Declarative, rule-based pattern-match and rewrite

Complex Transformation:

```
Reshape(Constant(x)) = x', where x’ is Reshape(x)
```

Native Code Call:

```
def ReshapeConstant :
NativeCodeCall<"$0.reshape(($1->getType()).cast<ShapedType>())">;
```

Transformation:

```
def ConstantReshapeOptPattern : Pat<(ReshapeOp:$res (ConstantOp $arg)), \
                                     (ConstantOp (ReshapeConstant $arg, $res))>;
```

Dialect Lowering

All the way to LLVM!

Lowering

- Goal: Translating source dialect into one or more target dialects
- Full or Partial
- Procedure:
  - Provide target dialects
  - Operation Conversion
  - Type Conversion

Google

DialectConversion framework

Goal: Transform illegal operations to legal ones

Components of the Framework:

- Conversion Target: Which dialects/ops are legal after lowering?
- Rewrite Patterns: Convert illegal ops to legal ops
- Type Convertor: Convert types

Google

Conversion Targets

- Legal Dialects (target dialects)
```
target.addLegalDialect<mlir::AffineOpsDialect, mlir::StandardOpsDialect>();
```
- Illegal Dialects (fail if not converted)
```
target.addIllegalDialect<ToyDialect>();
```
- Legal and Illegal Ops
```
target.addLegalOp<PrintOp>(); // preserve toy.print
target.addIllegalOp<BranchOp>(); // must convert
```
- Dynamically Legal Ops/Dialects (legality constraints such as operand type)
```
target.addDynamicallyLegalOp<ReturnOp>();
```

Google

Operation Conversion using ConversionPattern Rewriter

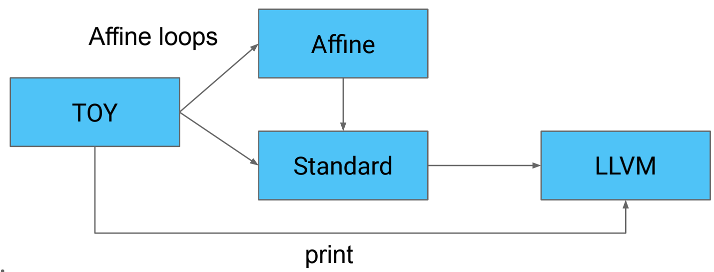
Convert illegal ops into legal ops using a pattern match and rewrite

Transitive conversion: `[bar.add -> baz.add, baz.add -> foo.add]`

ConversionPattern rewriter vs PatternMatch rewriter:
- Additional operands parameter to specify newly rewritten values
- No N->1 or N->M conversions
- Roll-back on failure

Google

Conceptually: Graph Of Lowering



A->B->C lowering

Lowering Graph:

- Nodes: Dialects/Ops
- Edges: Conversion
- Open Problem: Finding optimal* route

Affine to LLVM

- Now let’s generate some executable code
- Same conversion as before but with Type conversion
- Full Lowering

Populate the Lowering Graph:

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

MLIR LLVM dialect to LLVM IR

Mapping from LLVM Dialect ops 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

MLIR : Reusable Compiler Abstraction Toolbox

IR design involves multiple tradeoffs

- Iterative process, constant learning experience

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

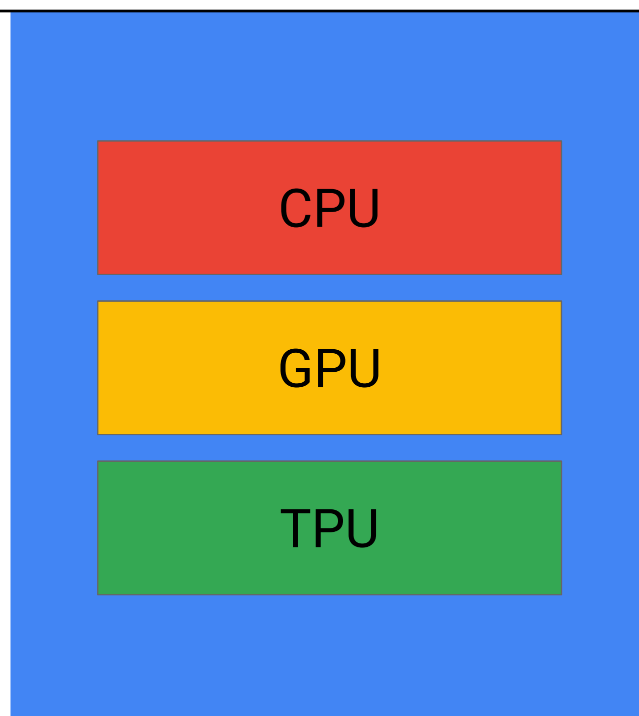
- Dialect-to-dialect lowering is easy
- Ops from different dialects can mix in same IR
    - Lowering from “A” to “D” may skip “B” and “C”
- Avoid lowering too early and losing information
    - Help define hard analyses away

No forced IR impedance mismatch

Fresh look at problems

Not shown today

- Heterogeneous compilation
- MLIR also includes GPU dialect to target
    - CUDA,
    - RocM, and
    - SPIR-V/Vulkan
- New converters to
    - TFLite
    - XLA



Recap

MLIR is a great infrastructure for higher-level compilation

- Gradual and partial lowerings to mixed dialects
  - All the way to LLVMIR and execution
- Reduce impedance mismatch at each level

MLIR provides all the infrastructure to build dialects and transformations

- At each level it is the <u>same</u> infrastructure

Demonstrated this on a Toy language

- Tutorial available on github

Google

Getting Involved

Google

MLIR is Open Source!

Visit us at github.com/tensorflow/mlir:

- Code, documentation, examples
  - Core moving to LLVM repo soon
- Developer mailing list at: mlir@tensorflow.org
- Open design meetings every Thursday
- Contributions welcome!

Google

MLIR

Thank you to the team!

Questions?

We are hiring!
mlir-hiring@google.com