MLIR Fundamentals

Hot Chips 34, 2022

Jacques Pienaar
Google

Outline

- Brief MLIR introduction
- MLIR philosophy
- What you get in the box
- Questions

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

Multi-Level Intermediate Representation

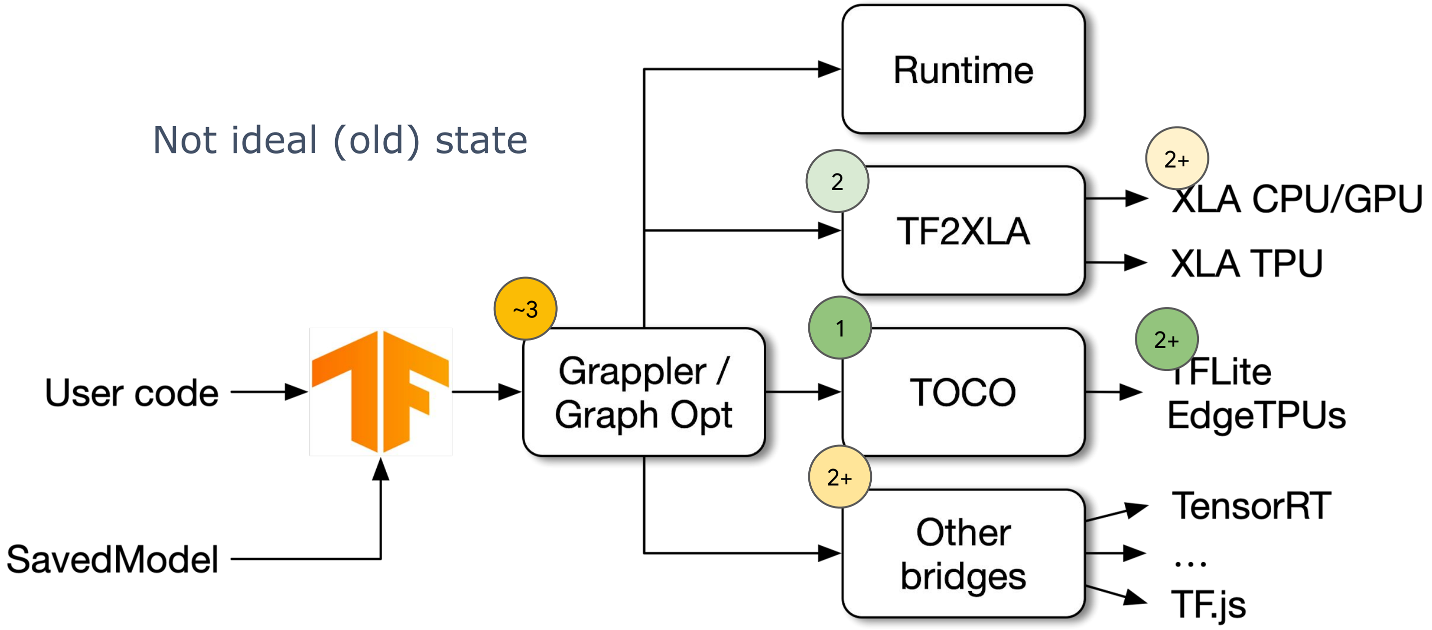
New compiler infrastructure

Originally built by team in TensorFlow ecosystem

Under neutral governance as part of LLVM project

Origin: Many graph compilers

Not ideal (old) state

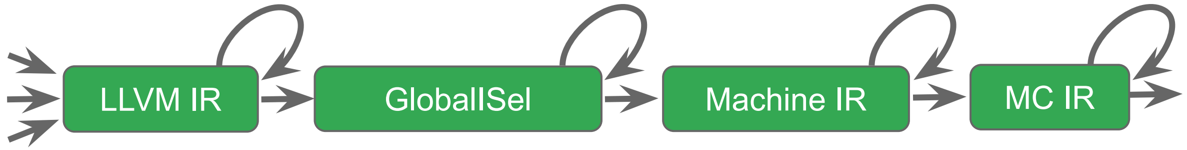


Many broken user journeys

LLVM: Industry Standard for Compiler Infrastructures

LLVM IR has proved itself as a versatile “mid-level” representation similar to C with vectors and SSA

LLVM: Industry Standard for Compiler Infrastructures

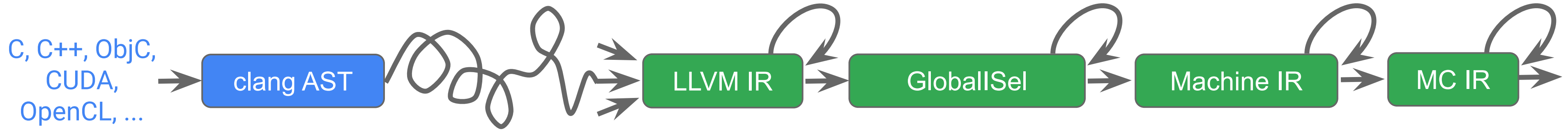


LLVM IR has proved itself as a versatile “mid-level” representation similar to C with vectors and SSA

LLVM IR is not enough for low-level representations

Multiple lower levels of abstraction introduced over time

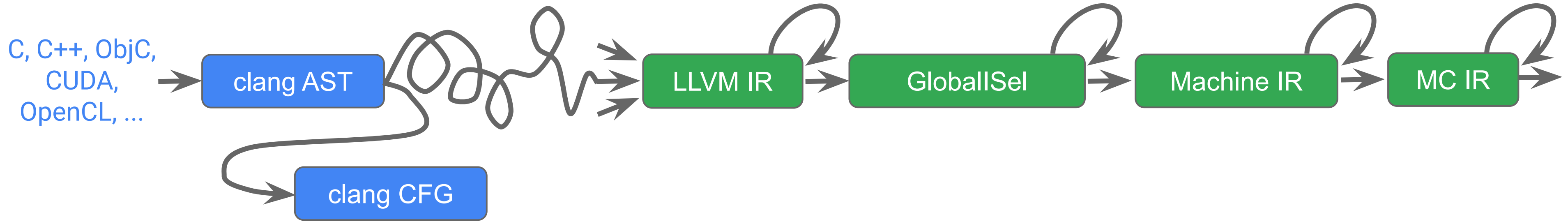
LLVM: Industry Standard for Compiler Infrastructures



LLVM IR is not enough for high-level representations

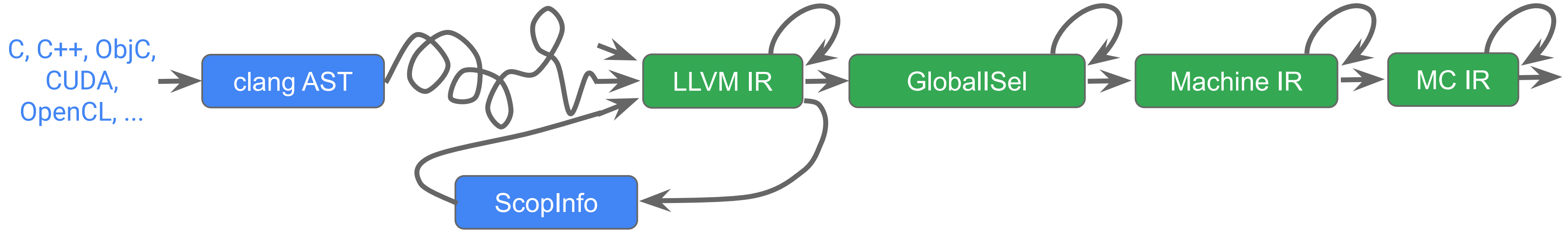
There is a huge abstraction gap between ASTs and LLVM IR, covered in a one-shot conversion in Clang.

LLVM: Industry Standard for Compiler Infrastructures



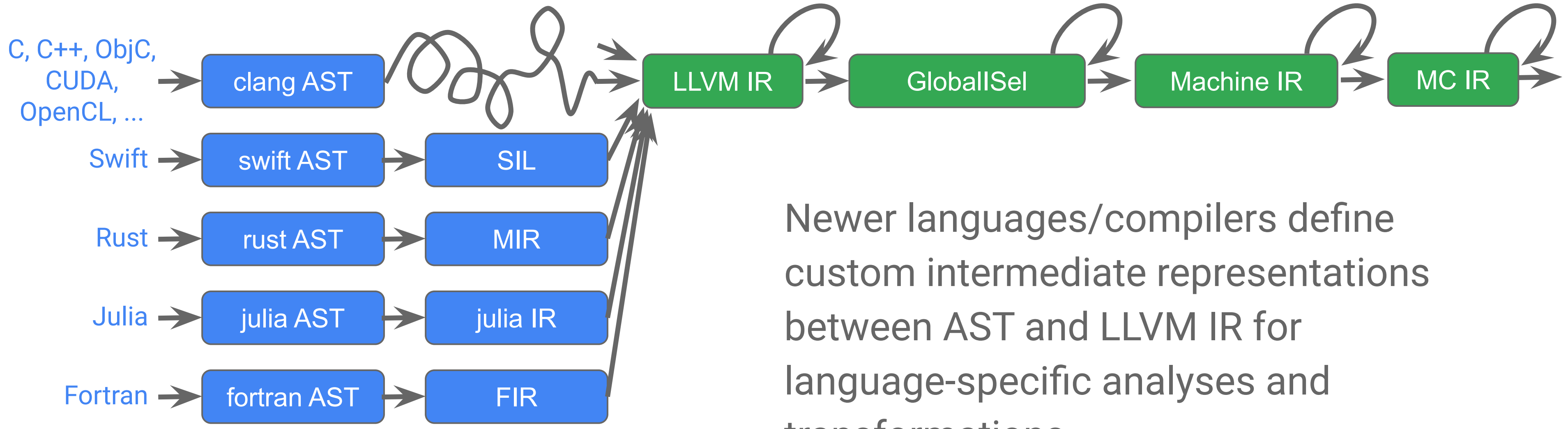
Clang has a representation parallel to AST used in, e.g., static analyzer, various advanced diagnostics.

LLVM: Industry Standard for Compiler Infrastructures



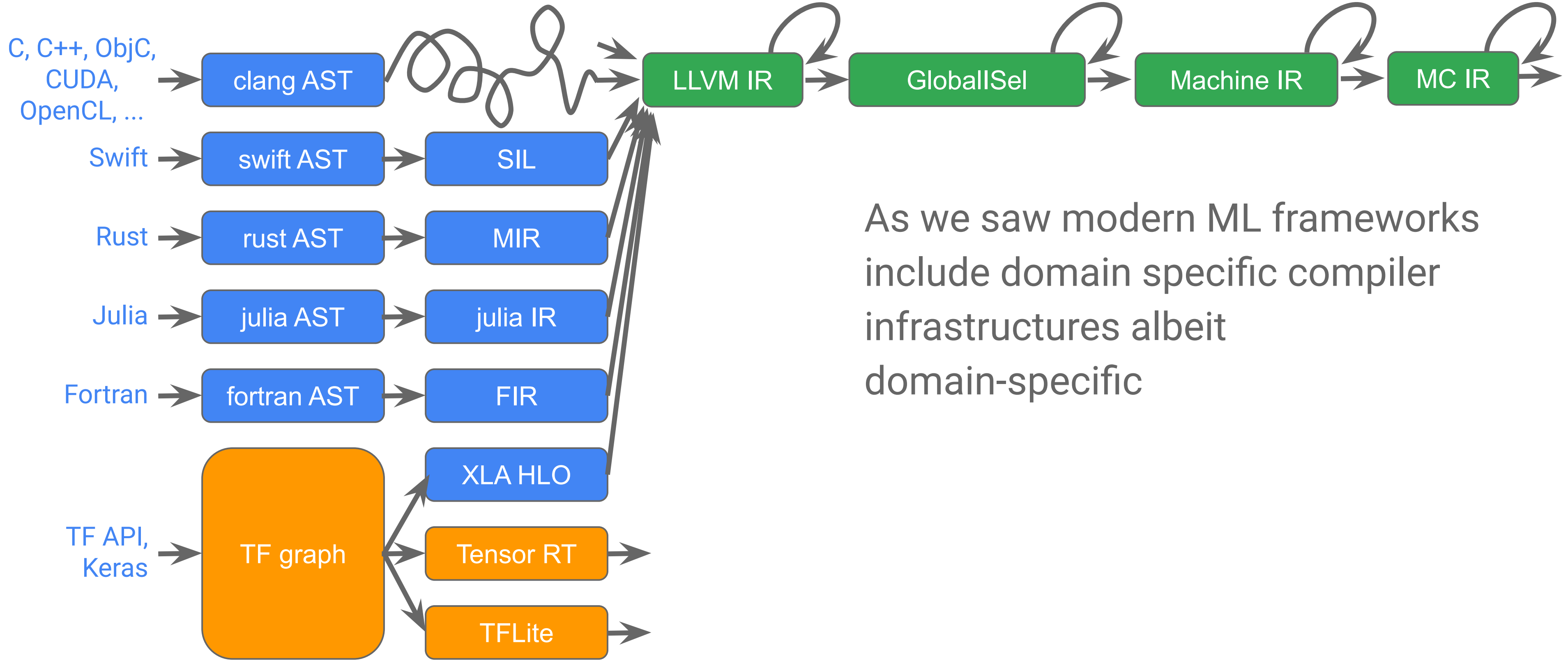
Some tools (e.g. Polly) resort to raising from LLVM IR to represent higher-level constructs such as loops.

LLVM: Industry Standard for Compiler Infrastructures



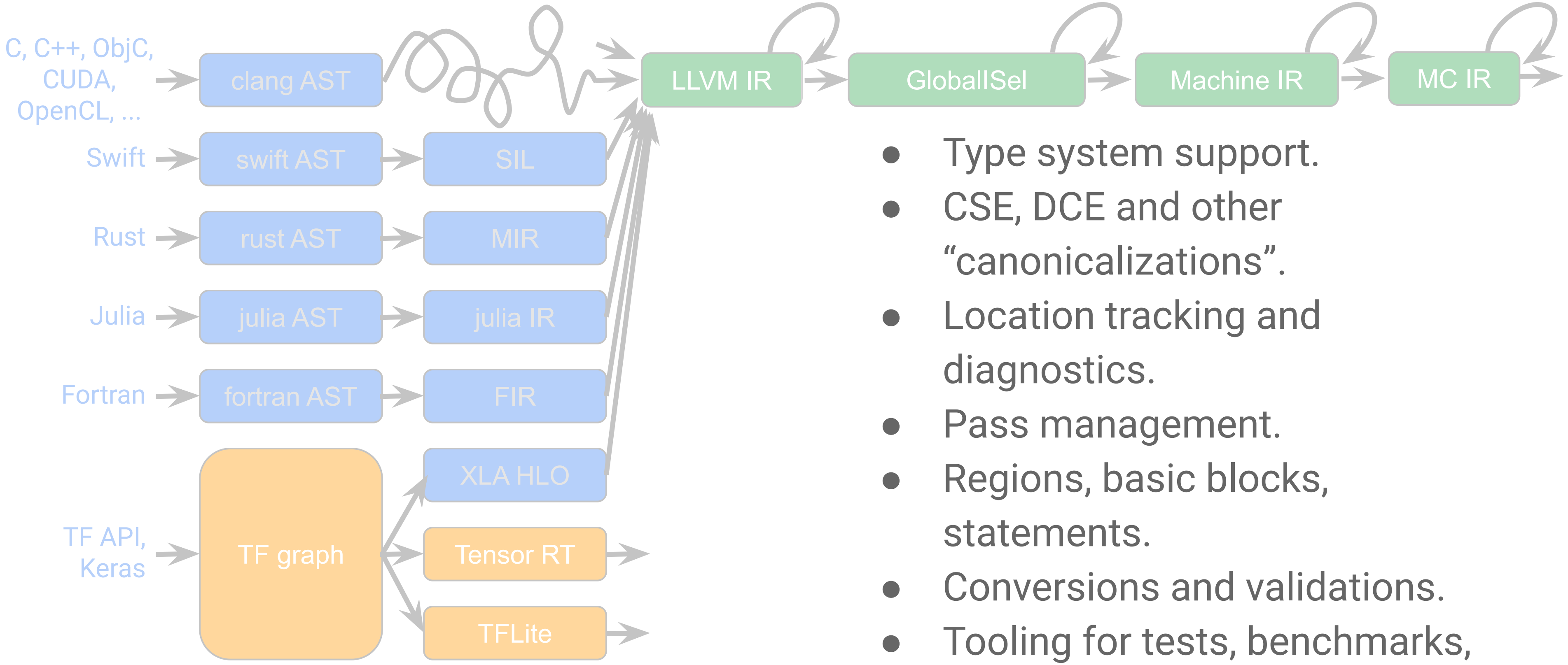
Newer languages/compilers define custom intermediate representations between AST and LLVM IR for language-specific analyses and transformations

It’s not even only source languages!



As we saw modern ML frameworks include domain specific compiler infrastructures albeit domain-specific

How much code in common but reimplemented?



- Type system support.
- CSE, DCE and other "canonicalizations".
- Location tracking and diagnostics.
- Pass management.
- Regions, basic blocks, statements.
- Conversions and validations.
- Tooling for tests, benchmarks, etc.

Domain specific intermediate representation

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**

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 passes (analysis/optimization)

And much more

⥀⇄⇊

Missing direction? Sort of

- Almost always easier to preserve than recover info
- Lifting is fragile
- User-intent impossible to recover

Principle: Don't destroy information/structure you'll need to recover later

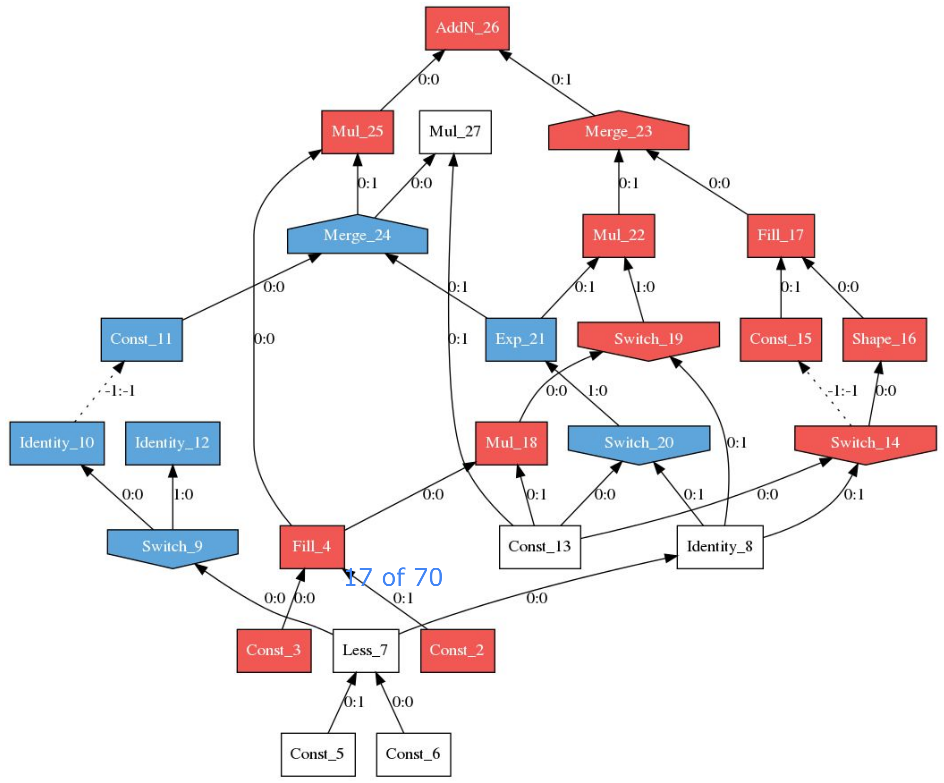
Example: TensorFlow Control Flow v1

User writes:

```
x = constant_op.constant(10.0, name='x')
pred = math_ops.less(1, 2)
fn1 = lambda: math_ops.exp(x, name='fn1')
fn2 = lambda: constant_op.constant(20.0)
r = control_flow_ops.cond(pred, fn1, fn2)
r = r * x
_ = gradients.gradients(r, [x])[0]
```

XLA wants (using MLIR MHLO dialect):

```
%c0 = mhlo.constant dense<1> : tensor<i64>
%c1 = mhlo.constant dense<2> : tensor<i64>
%0 = mhlo.while(%arg1 = %arg0) :
    (tensor<*xf32>) -> tensor<*xf32>
cond {

  %1 = "mhlo.compare"(%c0, %c1)
   {comparison_direction = "LT"} :
      (tensor<i64>, tensor<i64>) -> tensor<i1>
  "mhlo.return"(%1) : (tensor<i1>) -> ()
} do {
  // ...
  "mhlo.return"(%4) : (tensor<*f32>) -> ()
}
```



Dataflow computing

Mix and match in a single IR

I very rarely work with only 1 dialect (even at given time)

Lowering

TensorFlow

```
%x = "tf.Conv2d"(%input, %filter)
          {strides: [1,1,2,1], padding: "SAME", dilations: [2,1,1,1]}
    : (tensor<*xf32>, tensor<*xf32>) -> tensor<*xf32>
```

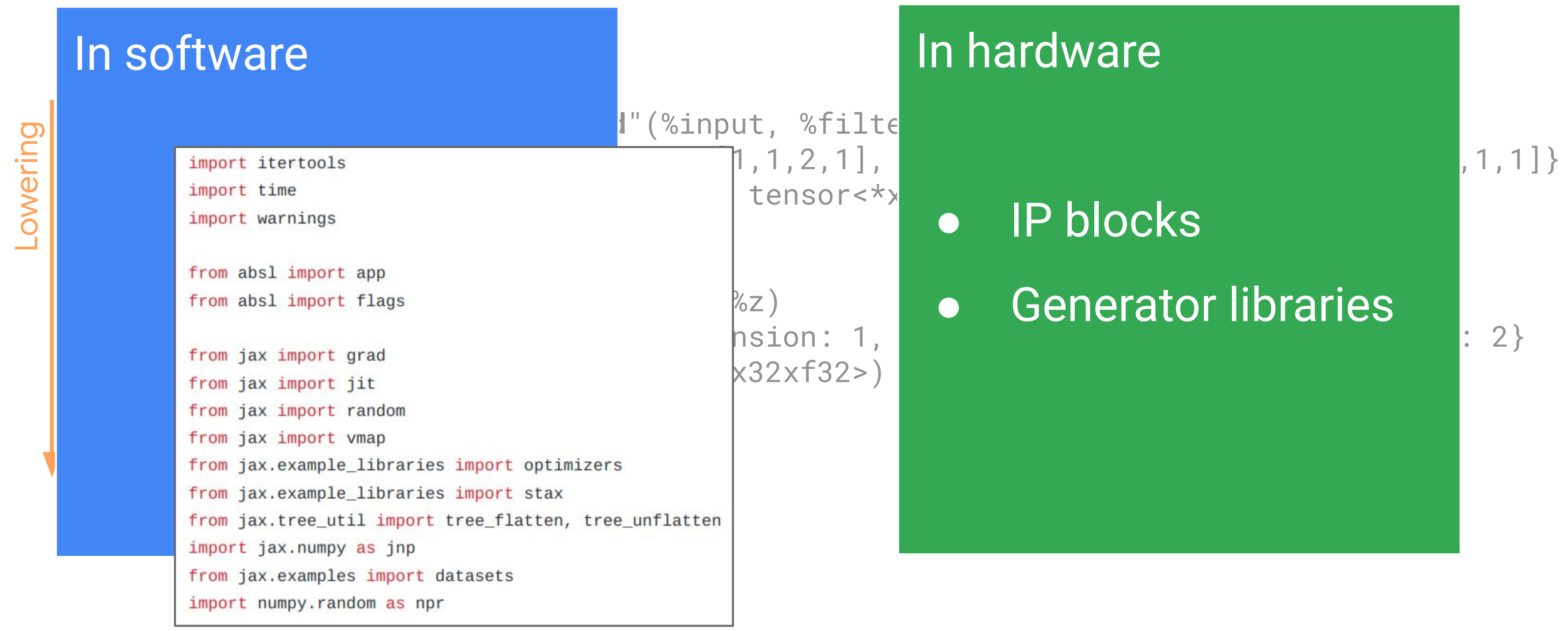
XLA (MHLO)

```
%m = “mhlo.AllToAll"(%z)
          {split_dimension: 1, concat_dimension: 0, split_count: 2}
    : (memref<300x200x32xf32>) -> memref<600x100x32xf32>
```

LLVM IR

```
%f = llvm.add %a, %b
    : f32
```

Mix and match in a single IR



In software

```
import itertools
import time
import warnings

from absl import app
from absl import flags

from jax import grad
from jax import jit
from jax import random
from jax import vmap
from jax.example_libraries import optimizers
from jax.example_libraries import stax
from jax.tree_util import tree_flatten, tree_unflatten
import jax.numpy as jnp
from jax.examples import datasets
import numpy.random as npr
```

```
"(%input, %filte
1,1,2,1],                          ,1,1]}
 tensor<*x

%z)
nsion: 1,                          : 2}
x32xf32>)
```

In hardware

- IP blocks
- Generator libraries

Don't create artificial boundaries

MLIR enables building domain specific IRs and representing problem domains

vs

Force all into one

Without reinventing the wheel

Without forcing abstracting over and dropping semantics until desired

Different mechanisms for abstracting (ops, interfaces, types)

Google

Core design principles

1. Parsimony
2. Traceability
3. Progressivity

See "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation", CGO, March 1, 2021 for further expansion

Design principles

Parsimony

"Entities should not be multiplied without necessity." In compilers, some things are intrinsically complex, **avoid making easy things incidentally complex**. A small set of versatile built-in concepts enables wide extensibility of the system.

Traceability

It is **almost always easier to preserve information than to recover it**. Keep the compiler accountable by making its operation transparent and analyzable. Declarative specification helps unless it becomes more complex than algorithms.

Progressivity

In compilers, **premature lowering is the predecessor of all evil**. Preserve high-level abstractions as long as necessary, lower them consciously. Embrace diverging flows and extensibility. **Intermediate state is important in an IR**.

Design requirements

Parsimony

- Everything extensible
- SSA + graphs + regions

Traceability

- Pervasive source location
- Declarative definitions

Progressivity

- Support high-level abstractions
- Progressive lowering

How is MLIR different?

State of Art Compiler Technology

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

Modular & Extensible

From graph representation through optimization to code generation

Not opinionated

Choose the level of representation that is right for your device

MLIR : Reusable Compiler Abstraction Toolbox

IR/"optimization format" design involves multiple tradeoffs

- Iterative process, constant learning experience

Doesn't think for you, enables iterating

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”

No forced IR impedance mismatch

- Avoid lowering too early and losing information
  - Premature lowering predecessor of all evil
  - Help define hard analyses away

Fresh look at problems

What's in the box

What's in box : Looking at code

- Model operations
- Defining passes/transforms
- Testing

Not shown:

- Defining custom attributes & types
- Dataflow analysis frameworks
  - Sparse & dense, lattice of values, possible to combine multiple together
- Existing sets of optimizations, analysis, dialects

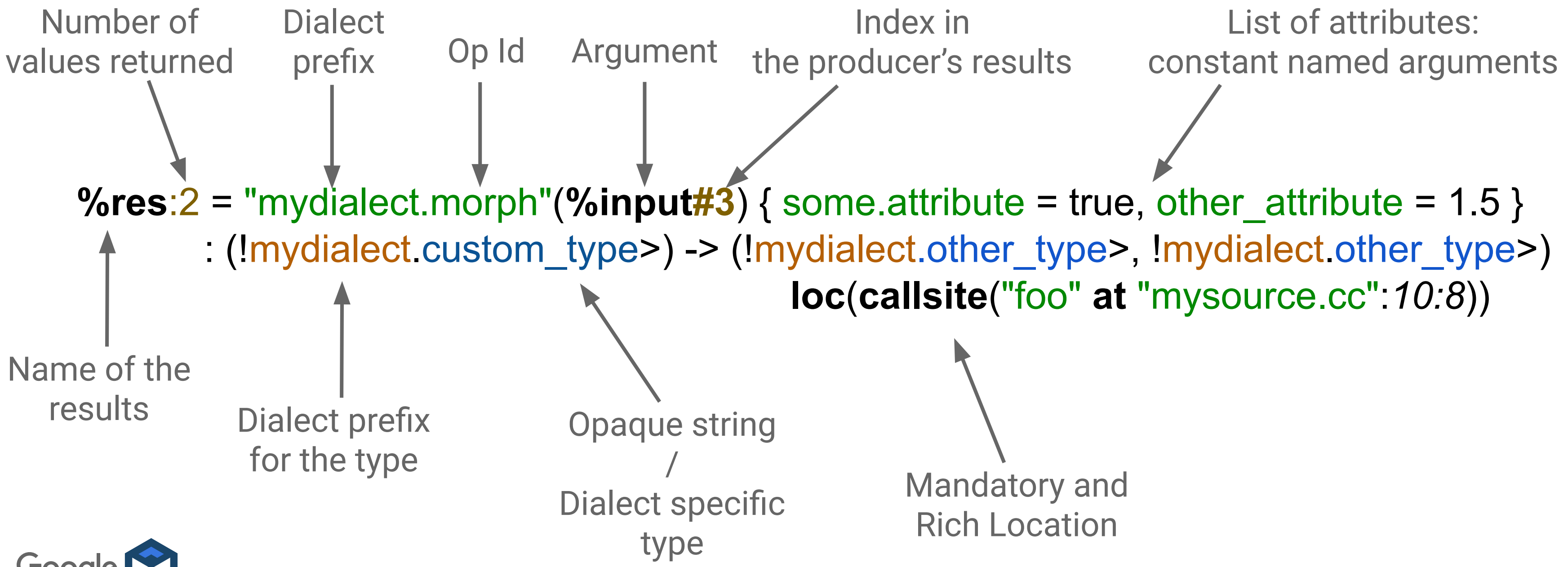
Google

Operations

- MLIR provides general system for creating and modelling operations
- Operations enable defining the level of abstraction/optimization
- Very little built-in concepts in MLIR
  - function, module, for-loop are all just operations
  - a user could have defined them just
- Operations *need not* be defined

```
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">
}
```

Syntax In a Nutshell



```
%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))
```

Number of values returned
Dialect prefix
Op Id
Argument
Index in the producer’s results
List of attributes: constant named arguments
Name of the results
Dialect prefix for the type
Opaque string / Dialect specific type
Mandatory and Rich Location

Dialects

A MLIR dialect is a logical grouping including:

- ~ consistent collection of abstractions/library
- A prefix (“namespace” reservation)
- A list of custom types
- A list of operations, each its name and implementation:
  - Verifier for operation invariants
  - Semantics (has-no-side-effects, constant-folding, CSE-allowed, ….)
- Possibly custom parser and assembly printer
- A list of passes (for analysis, transformations, and dialect conversions)

Defining a Dialect

- Dialect & custom types defined in C++
- Dialect can define hooks for
  - type printing and printing, constant folding
  - ...
- Ops can be defined
  - Programmatically (in C++)
  - Using **O**p **D**efinition **S**pec (TableGen)
    - All (almost all?) ops in TF, TFlite, MLIR core defined using ODS
    - Model hierarchies, multiclasses, ...
  - Custom printing, parsing, folding, canonicalization, verification
  - Documentation

```
def TF_LeakyReluOp : TF_Op<"LeakyRelu",
    [NoSideEffect,
     TF_SameOperandsAndResultTypeResolveRef]> {
  let summary =
    "Computes rectified linear: `max(features, features * alpha)`.";

  let arguments = (ins
    TF_FloatTensor:$features,

    DefaultValuedAttr<F32Attr, "0.2f">:$alpha
  );

  let results = (outs
    TF_FloatTensor:$activations
  );

  // Derived attributes are infrequent outside TF.
  TF_DerivedOperandTypeAttr T =
    TF_DerivedOperandTypeAttr<0>;

  let hasFolder = 1;
}
```

Progressive disclosure: Op modelling is a sliding scale

~~~
def TF_Log1pOp : TF_Op<"Log1p", [NoSideEffect,
    SameOperandsAndResultType, TF_CwiseUnary]> {
  let summary = "Computes natural logarithm of (1 + x) element-wise.";

  let description = [{
    I.e., \\(y = \log_e (1 + x)\\).

    Example:

    ```python
    x = tf.constant([0, 0.5, 1, 5])
    tf.math.log1p(x) ==> [0., 0.4054651, 0.6931472, 1.7917595]
    ```
  }];

  let arguments = (ins
    TF_FpOrComplexTensor:$x
  );

  let results = (outs
    TF_FpOrComplexTensor:$y
  );

  TF_DerivedOperandTypeAttr T = TF_DerivedOperandTypeAttr<0>;
}
~~~

- Start with conservative definition of op, refine over time
- The more modelled, the better
  - Verification -> good invariants results in smaller debugs
  - Side-effects enables greater optimizations: "may change the world" -> has to run before delete [opt-in to performance]
- For data flow names operands, results & basic attributes goes far
- Declarative assembly format

https://www.tensorflow.org/mlir/tf_ops#tflog1p_tflog1pop

Passes/transforms/patterns

- Now you have operations/modelled your problem, what now?
- Optimize the model
  - Mostly computationally (make it go faster)
  - Quantize it?
  - Compress operations?
- Analyze the graph
  - Find maximum memory usage
- Compile to target architecture
  - Lower to loops, target raw libraries [talk later today by Harsh]

Writing a pattern

Two ways:

1. C++ pattern

2. Declarative rewrite specification

```
def : Pat<(TF_SqueezeOp StaticShapeTensor:$arg), ( TFL_ReshapeOp $arg)>;
```

Google

Specify simple patterns simply

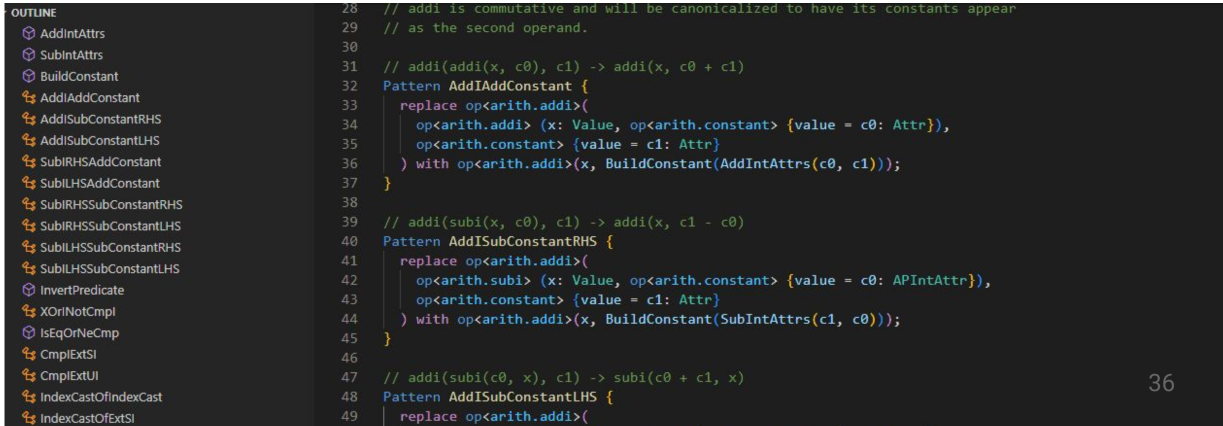
```
def : Pat<(TF_SqueezeOp StaticShapeTensor:$arg), (TFL_ReshapeOp $arg)>;
```

- Support M-N patterns
- Support constraints on Operations, Operands and Attributes
- Support specifying dynamic predicates
- Support native C++ code rewrites
    - Always a long tail, don't make the common case hard for the tail!

Goal: Reduces boilerplate, easy to express for simple cases

Declarative Rewrite Rule frontend

- Currently TableGen DAG (S-expr) format
  - Widely used in LLVM backends
  - Acquired taste still :)
  - It is intended to keep the simple case simple
- Also working on PDL, a lower level transformation bytecode
  - Frontend independent, goal to be targeted by multiple frontends
  - Others are building some Python rewrite specifications on top
- Others generating these from YAML



```
// addi is commutative and will be canonicalized to have its constants appear
// as the second operand.

// addi(addi(x, c0), c1) -> addi(x, c0 + c1)
Pattern AddIAddConstant {
  replace op<arith.addi>(
    op<arith.addi> (x: Value, op<arith.constant> {value = c0: Attr}),
    op<arith.constant> {value = c1: Attr}
  ) with op<arith.addi>(x, BuildConstant(AddIntAttrs(c0, c1)));
}

// addi(subi(x, c0), c1) -> addi(x, c1 - c0)
Pattern AddISubConstantRHS {
  replace op<arith.addi>(
    op<arith.subi> (x: Value, op<arith.constant> {value = c0: APIntAttr}),
    op<arith.constant> {value = c1: Attr}
  ) with op<arith.addi>(x, BuildConstant(SubIntAttrs(c1, c0)));
}

// addi(subi(c0, x), c1) -> subi(c0 + c1, x)
Pattern AddISubConstantLHS {
  replace op<arith.addi>(
```

Define a pass

```
include "mlir/Pass/PassBase.td"

// This defines the structure for a pass. Normally one would have multiple
// patterns or transformations per pass. And so defining a new pass isn't that
// frequent.
//
// The format below is used to both dictate on what the pass operates and to
// add description from which documentation could be generated. It can also
// have additional options as well specify dependent dialects.
def AddRewritePass : Pass<"add-rewrite", "FuncOp"> {
  // name of pass on CLI   ~~~~~~~~~~~~
  // type of op it operates on           ~~~~~~~
  let summary = "Example addition rewrite pass";
  let description = [{
    Does cool stuff.
  }];

  // Constructor that will return an instance of the AddRewrite pass.
  let constructor = "::mlir::TF::CreateAddRewritePass()";

  let options = [
    // This pass doesn't have any options (which is default), but adding to
    // make aware of as this allows passing options to the pass.
  ];

  let statistics = [
    Statistic<"num_addns_", "num-adds", "Number of AddNs after pass ran">
  ];
}
```

Pass driver (opt-tool)

- Normally per project level
- Pretty easy to add new driver:

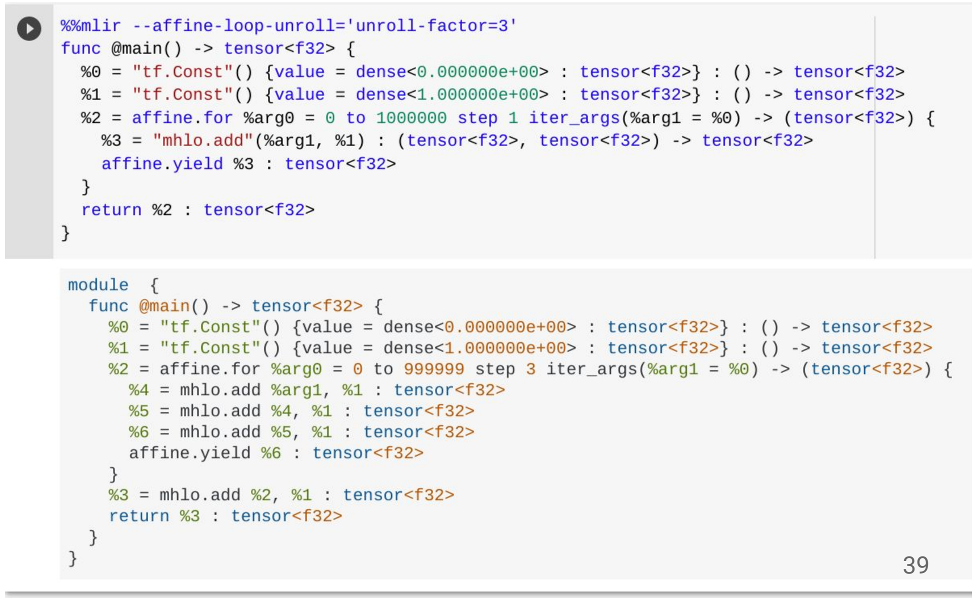
```
#include "add_rewrite_pass.h"
#include "tensorflow/compiler/mlir/init_mlir.h"
#include "tensorflow/compiler/mlir/tensorflow/dialect_registration.h"
#include "third_party/llvm/mlir/include/mlir/InitAllDialects.h"
#include "third_party/llvm/mlir/include/mlir/InitAllPasses.h"
#include "third_party/llvm/mlir/include/mlir/Support/MlirOptMain.h"

int main(int argc, char **argv) {
  tensorflow::InitMlir y(&argc, &argv);

  mlir::DialectRegistry registry;
  mlir::registerAllDialects(registry);
  mlir::RegisterAllTensorFlowDialects(registry);

  mlir::registerAllPasses();
  // New pass being tested.
  mlir::TF::registerTensorFlowAddRewritePasses();

  return failed(
      mlir::MlirOptMain(argc, argv, "Rewrite test pass driver\n", registry));
}
```

Hint: develop your passes iteratively like your tests

unit test

```
// RUN: mlir-opt %s -affine-loop-unroll="unroll-full" | FileCheck %s
func @loop_nest_simplest() {
 // UNROLL-FULL: affine.for %arg0 = 0 to 100 step 2 {
 affine.for %i = 0 to 100 step 2 {
  // UNROLL-FULL: %c1_i32 = constant 1 : i32
  // UNROLL-FULL-NEXT: %c1_i32_0 = constant 1 : i32
  // UNROLL-FULL-NEXT: %c1_i32_1 = constant 1 : i32
  // UNROLL-FULL-NEXT: %c1_i32_2 = constant 1 : i32
  affine.for %j = 0 to 4 {
   %x = constant 1 : i32
  }
 }      // UNROLL-FULL:  }
 return  // UNROLL-FULL:  return
}
```

colab



```
%%mlir --affine-loop-unroll='unroll-factor=3'
func @main() -> tensor<f32> {
  %0 = "tf.Const"() {value = dense<0.000000e+00> : tensor<f32>} : () -> tensor<f32>
  %1 = "tf.Const"() {value = dense<1.000000e+00> : tensor<f32>} : () -> tensor<f32>
  %2 = affine.for %arg0 = 0 to 1000000 step 1 iter_args(%arg1 = %0) -> (tensor<f32>) {
    %3 = "mhlo.add"(%arg1, %1) : (tensor<f32>, tensor<f32>) -> tensor<f32>
    affine.yield %3 : tensor<f32>
  }
  return %2 : tensor<f32>
}
```

```
module  {
  func @main() -> tensor<f32> {
    %0 = "tf.Const"() {value = dense<0.000000e+00> : tensor<f32>} : () -> tensor<f32>
    %1 = "tf.Const"() {value = dense<1.000000e+00> : tensor<f32>} : () -> tensor<f32>
    %2 = affine.for %arg0 = 0 to 999999 step 3 iter_args(%arg1 = %0) -> (tensor<f32>) {
      %4 = mhlo.add %arg1, %1 : tensor<f32>
      %5 = mhlo.add %4, %1 : tensor<f32>
      %6 = mhlo.add %5, %1 : tensor<f32>
      affine.yield %6 : tensor<f32>
    }
    %3 = mhlo.add %2, %1 : tensor<f32>
    return %3 : tensor<f32>
  }
}
```

Input may be written by hand or result of tools such as tf-translate or dumped reproducer module

Getting involved

MLIR is a community project

- Important takeaway from looking around internally and externally, from Compilers for Machine Learning (C4ML) & HPC community (SC) to HW folks (ISSCC)
  - All solving similar problems over and over
  - Effort on common (but very important and not really common) parts take away from value add
- MLIR is OSS with active community
  - **mlir.dev/forum** for Discourse forum (RFCs and longer discussions happen here)
  - **mlir.dev/chat** for Discord chat (quick convos, across time zones often here)

MLIR

Thank you to the team!

Questions?