IRDL: Dialects Definitions as an IR

An ODS-like dialect to define dialects at runtime

Mathieu Fehr, Théo Degioanni, and others

THE UNIVERSITY *of* EDINBURGH

Motivation

Motivation

In some use cases, we want to:

- define dialects using meta-programming (e.g. Linalg)
- define dialects from another language, like Python

Motivation

In some use cases, we want to:

- define dialects using meta-programming (e.g. Linalg)
- define dialects from another language, like Python

without the need to generate C++, or recompile MLIR

Motivation

We can already define new operations, attributes and types at runtime:

```
auto OpDef = DynamicOpDefinition::get(name, dialect, verifyFn, parseFn, printFn);
auto attrDef = DynamicAttrDefinition::get(name, dialect, verifyFn, printer, parser);
```

Motivation

We can already define new operations, attributes and types at runtime:

```
auto OpDef = DynamicOpDefinition::get(name, dialect, verifyFn, parseFn, printFn);
auto attrDef = DynamicAttrDefinition::get(name, dialect, verifyFn, printer, parser);
```

However, this requires a lot of boilerplate code.

Motivation

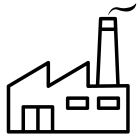
```
class XOrIOpAdaptor {
public:
  XOrIOpAdaptor(::mlir::ValueRange values, ::mlir::DictionaryAttr attrs = nullptr, ::mlir::RegionRange regions = {});
  XOrIOpAdaptor(XOrIOp&op);
  ::mlir::ValueRange getOperands();
  std::pair<unsigned, unsigned> getODSOperandIndexAndLength(unsigned index);
  ::mlir::ValueRange getODSOperands(unsigned index);
  ::mlir::Value lhs();
  ::mlir::Value rhs();
  ::mlir::DictionaryAttr getAttributes();
  ::mlir::LogicalResult verify(::mlir::Location loc);

private:
  ::mlir::ValueRange odsOperands;
  ::mlir::DictionaryAttr odsAttrs;
  ::mlir::RegionRange odsRegions;
};
class XOrIOp : public ::mlir::Op<XOrIOp, ::mlir::OpTrait::ZeroRegion, ::mlir::OpTrait::OneResult, ::mlir::OpTrait::OneTypedResult<::mlir::Type>::Impl, ::mlir::OpTrait::ZeroSuccessor, ::mlir::OpTrait::NOperands<2>::Impl, ::
::mlir::OpTrait::SameOperandsAndResultType, ::mlir::MemoryEffectOpInterface::Trait, ::mlir::VectorUnrollOpInterface::Trait, ::mlir::OpTrait::Elementwise, ::mlir::OpTrait::Scalarizable, ::mlir::OpTrait::Vectorizable, ::mli
public:
  using Op::Op;
  using Op::print;
  using Adaptor = XOrIOpAdaptor;
  static ::llvm::ArrayRef<::llvm::StringRef> getAttributeNames() {
              return {};
  }
  static constexpr ::llvm::StringLiteral getOperationName() {
              return ::llvm::StringLiteral("arith.xori");
  }
  std::pair<unsigned, unsigned> getODSOperandIndexAndLength(unsigned index);
  ::mlir::Operation::operand_range getODSOperands(unsigned index);
  ::mlir::Value lhs();
  ::mlir::Value rhs();
  ::mlir::MutableOperandRange lhsMutable();
  ::mlir::MutableOperandRange rhsMutable();
  std::pair<unsigned, unsigned> getODSResultIndexAndLength(unsigned index);
  ::mlir::Operation::result_range getODSResults(unsigned index);
  ::mlir::Value result();
  static void build(::mlir::OpBuilder &odsBuilder, ::mlir::OperationState &odsState, ::mlir::Type result, ::mlir::Value lhs, ::mlir::Value rhs);
  static void build(::mlir::OpBuilder &odsBuilder, ::mlir::OperationState &odsState, ::mlir::TypeRange resultTypes, ::mlir::Value lhs, ::mlir::Value rhs);
  static void build(::mlir::OpBuilder &, ::mlir::OperationState &odsState, ::mlir::TypeRange resultTypes, ::mlir::ValueRange operands, ::llvm::ArrayRef<::mlir::NamedAttribute> attributes = {});
  static void build(::mlir::OpBuilder &odsBuilder, ::mlir::OperationState &odsState, ::mlir::Value lhs, ::mlir::Value rhs);
  static void build(::mlir::OpBuilder &odsBuilder, ::mlir::OperationState &odsState, ::mlir::ValueRange operands, ::llvm::ArrayRef<::mlir::NamedAttribute> attributes = {});
  ::mlir::LogicalResult verify();
  static void getCanonicalizationPatterns(::mlir::RewritePatternSet &results, ::mlir::MLIRContext *context);
  ::mlir::OpFoldResult fold(::llvm::ArrayRef<::mlir::Attribute> operands);
  static ::mlir::ParseResult parse(::mlir::OpAsmParser &parser, ::mlir::OperationState &result);
  void print(::mlir::OpAsmPrinter &p);
  void getEffects(::mlir::SmallVectorImpl<::mlir::SideEffects::EffectInstance<::mlir::MemoryEffects::Effect>> &effects);
};
```

IRDL Goals

Concise

Introspectable

Dynamic



Generable

Main challenge: We should rely on a declarative specification, not C++

IRDL Goals

Concise

Generable

A dialect is exactly what we need!

Main challenge: We should rely on a declarative specification, not C++

THE UNIVERSITY *of* EDINBURGH

Usage

IRDL: IR Definition Language

```
irdl.Dialect cmath {




}
```

A ***Dialect*** definition is a single operation.

IRDL: IR Definition Language

```
irdl.Dialect cmath {
  irdl.Type complex {
    irdl.Parameters (elementType: !FloatType)
  }




}
```

A ***Dialect*** definition is a single operation.

A ***Type*** can be parametric.

IRDL: IR Definition Language

```
irdl.Dialect cmath {
  irdl.Type complex {
    irdl.Parameters (elementType: !FloatType)
  }

  irdl.Alias AnyComplex = !complex<!FloatType>




}
```

A ***Dialect*** definition is a single operation.

A ***Type*** can be parametric.

An ***Alias*** abbreviates lengthy types.

IRDL: IR Definition Language

```
irdl.Dialect cmath {
  irdl.Type complex {
    irdl.Parameters (elementType: !FloatType)
  }

  irdl.Alias AnyComplex = !complex<!FloatType>

  irdl.Operation mul {
    irdl.ConstraintVar (T: !AnyComplex)
    irdl.Operands (lhs: !T, rhs: !T)
    irdl.Results (res: !T)

    irdl.Format "$lhs, $rhs : $T.elementType"
  }
}
```

A ***Dialect*** definition is a single operation.

A ***Type*** can be parametric.

An ***Alias*** abbreviates lengthy types.

An ***Operation*** is defined similarly to ODS.

Registering IRDL at runtime

IRDL

```
irdl.Dialect cmath {
  irdl.Type complex {
    irdl.Parameters (elementType: !FloatType)
  }

  irdl.Alias AnyComplex = !complex<!FloatType>

  irdl.Operation mul {
    irdl.ConstraintVar (T: !AnyComplex)
    irdl.Operands (lhs: !T, rhs: !T)
    irdl.Results (res: !T)

    irdl.Format "$lhs, $rhs : $T.elementType"
  }
}
```

MLIR

```
func @conorm(%p: !cmath.complex<std.f32>,
             %q: !cmath.complex<std.f32>) -> !std.f32 {
       %pq = cmath.mul(%p, %q) : !std.f32
       return %pq : !cmath.complex<std.f32>
}
```

THE UNIVERSITY *of* EDINBURGH

Language Specification

IRDL Local Constraints

```
irdl.Type complex {
  irdl.Parameters (elementType: FloatType)
}
```

IRDL Local Constraints

```
irdl.Type complex {
  irdl.Parameters (elementType: FloatType)
}
```

Type and Attribute constraints
represent local structural invariants:

```
irdl.Alias myI32             = !i32
irdl.Alias AnyComplex        = complex<Any>
irdl.Alias AnyComplex2       = complex
irdl.Alias ComplexF32        = complex<!f32>
irdl.Alias ComplexF32OrF64   = complex<AnyOf<!f32, !f64>>
irdl.Alias ComplexNotF32     = complex<And<FloatType, Not<!f32>>>
```

IRDL Custom Local Constraints (WIP)

```
irdl.Alias custom<T> = custom<myConstr>(T)
```

Custom constraints are defined outside IRDL.

C++

```
auto myConstraint = [](auto emitError, SmallVector<Attribute> attrs) {
    ...
};
irdlDialect->registerCustomConstraint(myConstraint, "myConstr");
```

PYTHON

```
def my_constraint(emitError, args):
  ...

irdlDialect.register_custom_constraint(my_constraint, "myConstr");
```

IRDL Custom Local Constraints (WIP)

What should custom constraints take as arguments?

IRDL Custom Local Constraints (WIP)

What should custom constraints take as arguments?

```
irdl.Alias custom1 = custom<myConstr1>(2)
```

Using a ***custom parser*** for constraints?

IRDL Custom Local Constraints (WIP)

What should custom constraints take as arguments?

```
irdl.Alias custom1 = custom<myConstr1>(2)
irdl.Alias custom2 = custom<myConstr2>(custom1)
```

Using a ***custom parser*** for constraints?

Using **constraints parameters**?

IRDL Constraint over C++ types/attributes

```
irdl.Alias myComplex<T> = builtin.complex<T>
```


C++ types/attributes can be used in IRDL.

C++

```
class ComplexTypeWrapper : public ConcreteTypeWrapper<ComplexType> {
  StringRef getName() { return "builtin.complex"; }

  SmallVector<Attribute> getParameters(ComplexType type) {
    return {TypeAttr::get(type.getElementType())};
  }
};
```

C++

```
irdlDialect->addTypeWrapper<ComplexTypeWrapper>();
```

IRDL Constraint over C++ types/attributes

```
irdl.Alias myComplex<T> = builtin.complex<T>
```

C++ types/attributes can be used in IRDL.

```
class Comple
  StringRef

  SmallVecto
    return {
  }
};
```

Could MLIR provide a way to get this information?
(e.g. extending SubElementTypeInterface)

```
irdlDialect->addTypeWrapper<ComplexTypeWrapper>();
```

C++

Type and Attribute definitions

```
irdl.Type complex {
  irdl.Parameters (elementType: !FloatType)
  irdl.Summary "A complex type"
  irdl.Format "`<` $elementType `>`"
}
```

Parameters with local constraints

Integrated ***documentation***

Assembly format defines a textual representation

Constraint variables

```
irdl.Attribute complex {
  irdl.ConstraintVar (T: !FloatType)
  irdl.Parameters (re: #FloatAttr<Any, T>,
                   im: #FloatAttr<Any, T>)
}
```

Constraint variables specify global equality constraints

```
#complex<4.2 : !std.f32, 2.4 :!std.f32>
#complex<4.2 : !std.f64, 2.4 : !std.f64>

#complex<4.2 : !std.f64, 2.4 : !std.f32>
// Error: non-equal parameters

#complex<42 : !std.i32, 24 : !std.i32>
// Error: parameters not satisfying local constraint
```

Operation definitions

```
irdl.Operation norm {
  irdl.ConstraintVar (T: !Any)
  irdl.Operands (c: !AnyComplex<T>)
  irdl.Results (res: !T)

  irdl.Format "$c : $T.elementType"
  irdl.Summary "The euclidean norm of a
                complex number"
}
```

Operands and **results** use type constraints

Assembly format can use type parameters

```
%norm1 = "cmath.norm"(%q) : (!cmath.complex<std.f32>) -> !std.f32
%norm2 = cmath.norm(%q) : !std.f32
```

Traits and Interfaces Implementation (WIP)

```
irdl.Operation norm {
  irdl.ConstraintVar (T: !Any)
  irdl.Operands (c: !AnyComplex<T>)
  irdl.Results (res: !T)

  irdl.Traits (Trait1, trait2, ...)
  irdl.Interfaces (Interface1<args...>, ...)
}
```

Traits and **Interfaces** need to be registered in IRDL beforehand.

C++

```
auto irdlDialect = ctx->getOrLoadDialect<IRDLDialect>();

irdlDialect->registerTrait<Trait1>("Trait1");
irdlDialect->registerInterface<Interface1>("Interface1", parseFn)
```

Traits and Interfaces Implementation (WIP)

```
irdl.Operation norm {
  irdl.ConstraintVar (T: !Any)
  irdl.Operands (c: !AnyComplex<T>)
  irdl.Results (res: !T)

  irdl.Tra
  irdl.Int
}
```

d **Interfaces** need
stered in IRDL
nd.

Challenges:

- How to handle templated traits/interfaces?
- How to handle dependent traits/interfaces?

C++

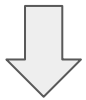
```
auto irdlDialect = ctx->getOrLoadDialect<IRDLDialect>();

irdlDialect->registerTrait<Trait1>("Trait1");
irdlDialect->registerInterface<Interface1>("Interface1", parseFn)
```

THE UNIVERSITY *of* EDINBURGH

IRDL-SSA, a lower abstraction

IRDL-SSA: an SSA-based IRDL

```
irdl.Operation mul {
  irdl.ConstraintVar (T: complex<AnyOf<f32, f64>>)
  irdl.Operands (lhs: !T, rhs: !T)
  irdl.Results (res: !T)
}
```



```
irdlssa.operation mul {
  %0 = irdlssa.is_type : f32
  %1 = irdlssa.is_type : f64
  %2 = irdlssa.any_of(%0, %1)
  %3 = irdlssa.parametric_type : "cmath.complex"<%2>
  irdlssa.operands(lhs: %3, rhs: %3)
  irdlssa.results(res: %3)
}
```

Each SSA value correspond to a single type/attribute

IRDL-SSA: an SSA-based IRDL

```
irdl.Operation mul {
  irdl.ConstraintVar (T: complex<AnyOf<f32, f64>>)
  irdl.Operands (lhs: !T, rhs: !T)
  irdl.Results (res: !T)
}
```

```
irdlssa.operation mul {
  %0 = irdlssa.is_type : f32
  %1 = irdlssa.is_type : f64
  %2 = irdlssa.any_of(%0, %1)
  %3 = irdlssa.parametric_type : "cmath.complex"<%2>
  irdlssa.operands(lhs: %3, rhs: %3)
  irdlssa.results(res: %3)
}
```

Each SSA value correspond to a single type/attribute

IRDL-SSA: an SSA-based IRDL

```
irdl.Operation mul {
  irdl.ConstraintVar (T: complex<AnyOf<f32, f64>>)
  irdl.Operands (lhs: !T, rhs: !T)
  irdl.Results (res: !T)
}
```

```
irdlssa.operation mul {
  %0 = irdlssa.is_type : f32
  %1 = irdlssa.is_type : f64
  %2 = irdlssa.any_of(%0, %1)
  %3 = irdlssa.parametric_type : "cmath.complex"<%2>
  irdlssa.operands(lhs: %3, rhs: %3)
  irdlssa.results(res: %3)
}
```

Each SSA value correspond to a single type/attribute

IRDL-SSA: an SSA-based IRDL

```
irdl.Operation mul {
  irdl.ConstraintVar (T: complex<AnyOf<f32, f64>>)
  irdl.Operands (lhs: !T, rhs: !T)
  irdl.Results (res: !T)
}
```

```
irdlssa.operation mul {
  %0 = irdlssa.is_type : f32
  %1 = irdlssa.is_type : f64
  %2 = irdlssa.any_of(%0, %1)
  %3 = irdlssa.parametric_type : "cmath.complex"<%2>
  irdlssa.operands(lhs: %3, rhs: %3)
  irdlssa.results(res: %3)
}
```

Each SSA value correspond to a single type/attribute

IRDL-SSA: an SSA-based IRDL

```
irdl.Operation mul {
  irdl.ConstraintVar (T: complex<AnyOf<f32, f64>>)
  irdl.Operands (lhs: !T, rhs: !T)
  irdl.Results (res: !T)
}
```

```
irdlssa.operation mul {
  %0 = irdlssa.is_type : f32
  %1 = irdlssa.is_type : f64
  %2 = irdlssa.any_of(%0, %1)
  %3 = irdlssa.parametric_type : "cmath.complex"<%2>
  irdlssa.operands(lhs: %3, rhs: %3)
  irdlssa.results(res: %3)
}
```

Each SSA value correspond to a single type/attribute

IRDL-SSA: an SSA-based IRDL

```
irdlssa.operation mul {
  %0 = irdlssa.is_type : f32
  %1 = irdlssa.is_type : f64
  %2 = irdlssa.any_of(%0, %1)
  %3 = irdlssa.parametric_type : "cmath.complex"<%2>
  irdlssa.operands(lhs: %3, rhs: %3)
  irdlssa.results(res: %3)
}
```

```
%2 = cmath.mul(%0, %1) :
  (!cmath.complex<f64>, !cmath.complex<f64>) ->
   !cmath.complex<f64>
```

Each SSA value correspond to a single type/attribute

IRDL-SSA: an SSA-based IRDL

```
irdlssa.operation mul {
  %0 = irdlssa.is_type : f32
  %1 = irdlssa.is_type : f64
  %2 = irdlssa.any_of(%0, %1)
  %3 = irdlssa.parametric_type : "cmath.complex"<%2>
  irdlssa.operands(lhs: %3, rhs: %3)
  irdlssa.results(res: %3)
}
```

```
%2 = cmath.mul(%0, %1) :
  (!cmath.complex<f64>, !cmath.complex<f64>) ->
   !cmath.complex<f64>
```

Each SSA value correspond to a single type/attribute

IRDL-SSA: an SSA-based IRDL

```
irdlssa.operation mul {
  %0 = irdlssa.is_type : f32
  %1 = irdlssa.is_type : f64
  %2 = irdlssa.any_of(%0, %1)
  %3 = irdlssa.parametric_type : "cmath.complex"<%2>
  irdlssa.operands(lhs: %3, rhs: %3)
  irdlssa.results(res: %3)
}
```

```
%2 = cmath.mul(%0, %1) :
  (!cmath.complex<f64>, !cmath.complex<f64>) ->
   !cmath.complex<f64>
```

Each SSA value correspond to a single type/attribute

IRDL-SSA: an SSA-based IRDL

```
irdlssa.operation mul {
  %0 = irdlssa.is_type : f32
  %1 = irdlssa.is_type : f64
  %2 = irdlssa.any_of(%0, %1)
  %3 = irdlssa.parametric_type : "cmath.complex"<%2>
  irdlssa.operands(lhs: %3, rhs: %3)
  irdlssa.results(res: %3)
}
```

```
%2 = cmath.mul(%0, %1) :
  (!cmath.complex<f64>, !cmath.complex<f64>) ->
   !cmath.complex<f64>
```

Each SSA value correspond to a single type/attribute

IRDL-SSA: an SSA-based IRDL

```
irdlssa.operation mul {
  %0 = irdlssa.is_type : f32
  %1 = irdlssa.is_type : f64
  %2 = irdlssa.any_of(%0, %1)
  %3 = irdlssa.parametric_type : "cmath.complex"<%2>
  irdlssa.operands(lhs: %3, rhs: %3)
  irdlssa.results(res: %3)
}
```

```
%2 = cmath.mul(%0, %1) :
  (!cmath.complex<f64>, !cmath.complex<f64>) ->
   !cmath.complex<f64>
```

Each SSA value correspond to a single type/attribute

IRDL-SSA: an SSA-based IRDL

```
irdlssa.operation mul {
  %0 = irdlssa.is_type : f32
  %1 = irdlssa.is_type : f64
  %2 = irdlssa.any_of(%0, %1) // f64
  %3 = irdlssa.parametric_type : "cmath.complex"<%2>
  irdlssa.operands(lhs: %3, rhs: %3)
  irdlssa.results(res: %3)
}
```

```
%2 = cmath.mul(%0, %1) :
  (!cmath.complex<f64>, !cmath.complex<f64>) ->
   !cmath.complex<f64>
```

Each SSA value correspond to a single type/attribute

IRDL-SSA: an SSA-based IRDL

```
irdlssa.operation mul {
  %0 = irdlssa.is_type : f32
  %1 = irdlssa.is_type : f64
  %2 = irdlssa.any_of(%0, %1) // f64
  %3 = irdlssa.parametric_type : "cmath.complex"<%2> // cmath.complex<f64>
  irdlssa.operands(lhs: %3, rhs: %3)
  irdlssa.results(res: %3)
}
```

```
%2 = cmath.mul(%0, %1) :
  (!cmath.complex<f64>, !cmath.complex<f64>) ->
   !cmath.complex<f64>
```

IRDL-SSA: an SSA-based IRDL

```
irdlssa.operation mul {
  %0 = irdlssa.is_type : f32
  %1 = irdlssa.is_type : f64
  %2 = irdlssa.any_of(%0, %1) // f64
  %3 = irdlssa.parametric_type : "cmath.complex"<%2> // cmath.complex<f64>
  irdlssa.operands(lhs: %3, rhs: %3)
  irdlssa.results(res: %3)
}
```

Equality check

```
%2 = cmath.mul(%0, %1) :
  (!cmath.complex<f64>, !cmath.complex<f64>) ->
   !cmath.complex<f64>
```

IRDL-SSA: an SSA-based IRDL

```
irdlssa.operation mul {
  %0 = irdlssa.is_type : f32
  %1 = irdlssa.is_type : f64
  %2 = irdlssa.any_of(%0, %1) // f64
  %3 = irdlssa.parametric_type : "cmath.complex"<%2> // cmath.complex<f64>
  irdlssa.operands(lhs: %3, rhs: %3)
  irdlssa.results(res: %3)
}
```

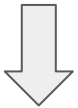
Equality check

```
%2 = cmath.mul(%0, %1) :
  (!cmath.complex<f64>, !cmath.complex<f64>) ->
   !cmath.complex<f64>
```

IRDL-SSA: Optimizations for free

```
%0 = irdlssa.is_type : f32
%1 = irdlssa.is_type : f64
%2 = irdlssa.any_of(%0, %1)
%3 = irdlssa.parametric_type : "cmath.complex"<%2>
%4 = irdlssa.parametric_type : "cmath.complex"<%2>
%5 = irdlssa.parametric_type : "cmath.complex"<%2>
```

Redundant computation



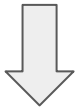
CSE

```
%0 = irdlssa.is_type : f32
%1 = irdlssa.is_type : f64
%2 = irdlssa.any_of(%0, %1)
%3 = irdlssa.parametric_type : "cmath.complex"<%2>
```

IRDL-SSA: Additional optimizations (WIP)

```
// Complex<AnyOf<f32, f64>>
%0 = irdlssa.is_type : f32
%1 = irdlssa.is_type : f64
%2 = irdlssa.any_of(%0, %1)
%3 = irdlssa.parametric_type : "cmath.complex"<%2>
```

parametric_type constraints require ***runtime introspection.***



```
// AnyOf<Complex<f32>, Complex<f64>>
%0 = irdlssa.is_type : "cmath.complex"<f32>
%1 = irdlssa.is_type : "cmath.complex"<f64>
%2 = irdlssa.any_of(%0, %1)
```

This can be optimized to only **pointer equality**.

THE UNIVERSITY *of* EDINBURGH

Analysing MLIR dialects

Translation of MLIR dialects to IRDL

Semi-automatic translation

affine.irdl

```
irdl.dialect affine {
  irdl.operation affine.apply {
```

arith.irdl

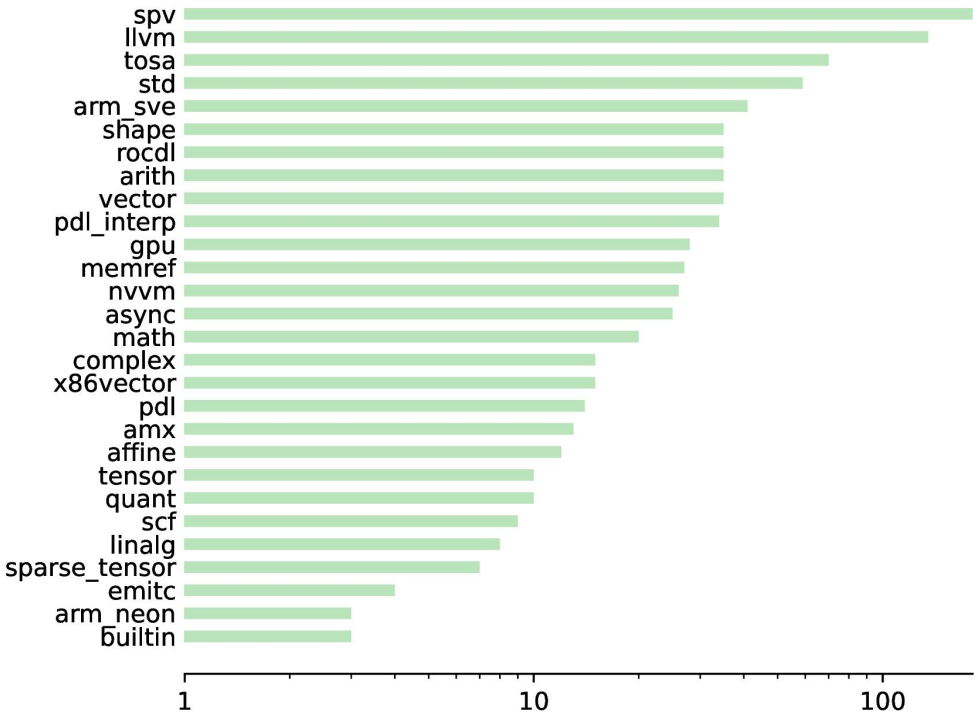
```
irdl.dialect arith {
  irdl.operation arith.addf {
```

linalg.irdl

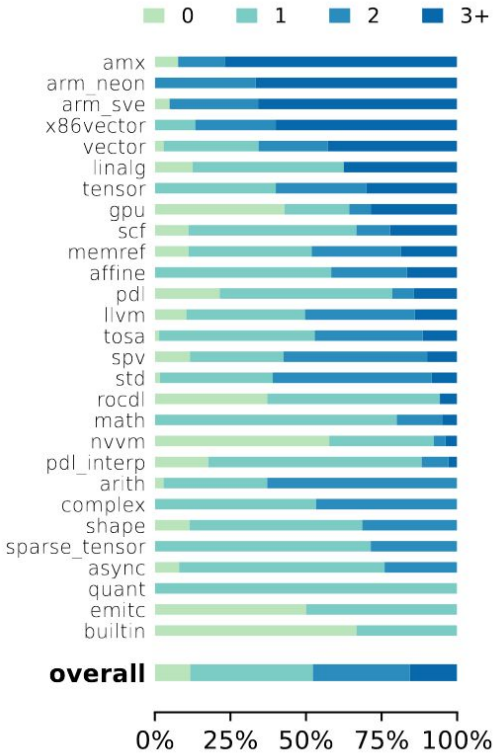
```
irdl.dialect linalg {
  irdl.type range {
                irdl.parameters()
  }
  irdl.operation linalg.index {
                irdl.results(result: CppClass<::mlir::IndexType>)
  }

  irdl.operation linalg.init_tensor {
                irdl.operands(sizes: Variadic<CppClass<::mlir::IndexType>>)
                irdl.results(result: And<CppClass<::mlir::TensorType>, Any>)
  }

  irdl.operation linalg.pad_tensor {
                irdl.operands(source: And<CppClass<::mlir::TensorType>, Any>,
                low: Variadic<CppClass<::mlir::IndexType>>,
                high: Variadic<CppClass<::mlir::IndexType>>)
                irdl.results(result: And<CppClass<::mlir::TensorType>, Any>)
  }

  irdl.operation linalg.range {
                irdl.operands(min: CppClass<::mlir::IndexType>,
                max: CppClass<::mlir::IndexType>,
                step: CppClass<::mlir::IndexType>)
                irdl.results(__empty__: CppClass<RangeType>)
  }

  irdl.operation linalg.tensor_collapse_shape {
                irdl.operands(src: And<CppClass<::mlir::TensorType>, Any>)
                irdl.results(result: And<CppClass<::mlir::TensorType>, Any>)
  }

  irdl.operation linalg.tensor_expand_shape {
                irdl.operands(src: And<CppClass<::mlir::TensorType>, Any>)
                irdl.results(result: And<CppClass<::mlir::TensorType>, Any>)
  }

  irdl.operation linalg.tiled_loop {
                irdl.operands(lowerBound: Variadic<CppClass<::mlir::IndexType>>,
                upperBound: Variadic<CppClass<::mlir::IndexType>>,
                step: Variadic<CppClass<::mlir::IndexType>>,
                inputs: Variadic<Any>,
                outputs: Variadic<And<CppClass<::mlir::ShapedType>, Any>>)
                irdl.results(results:
Variadic<And<And<CppClass<::mlir::TensorType>,
CPPPredicate<"$_self.cast<::mlir::ShapedType>().hasRank()">>, Any>>)
  }
```

Automatic analysis

MLIR: The # of Operations in default dialects

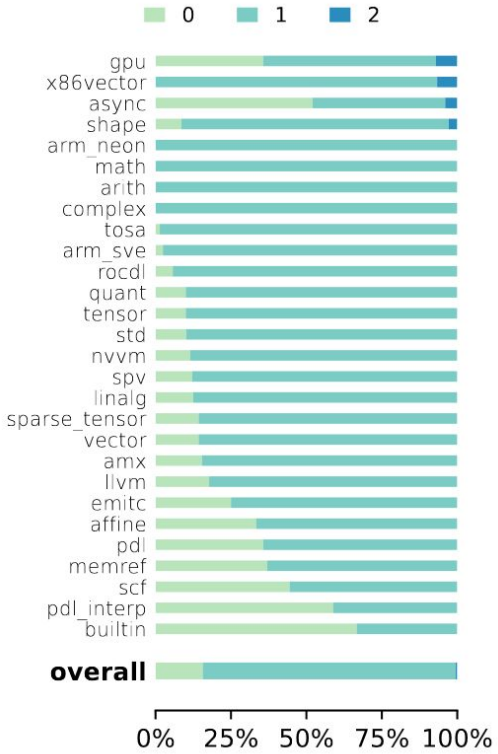


Operand use in operations across MLIR dialects



Many three-operand-ops in arm_neon and arm_sve.

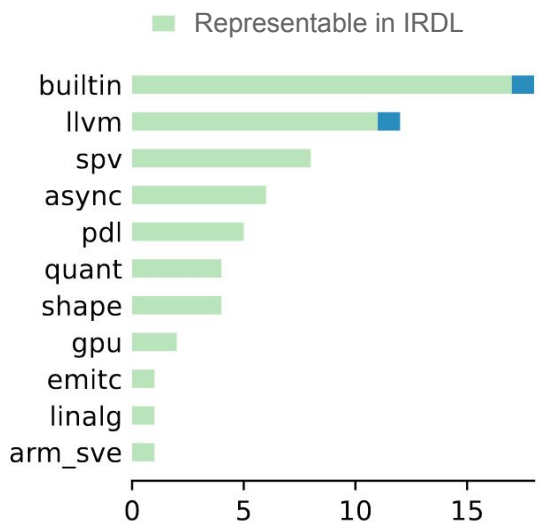
MLIR IR Operations do not require many return types



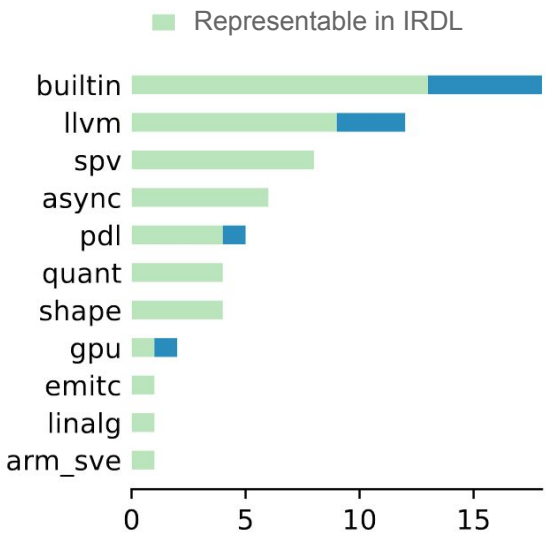
We rarely use more than two return values!

Most types do not require additional C++

Structural Definition

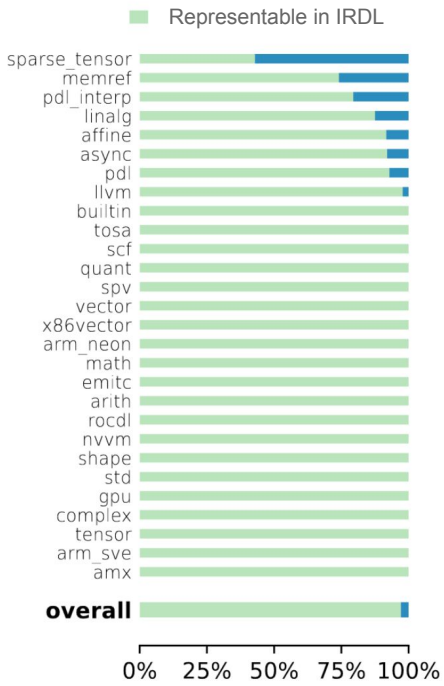


Verifier Definition

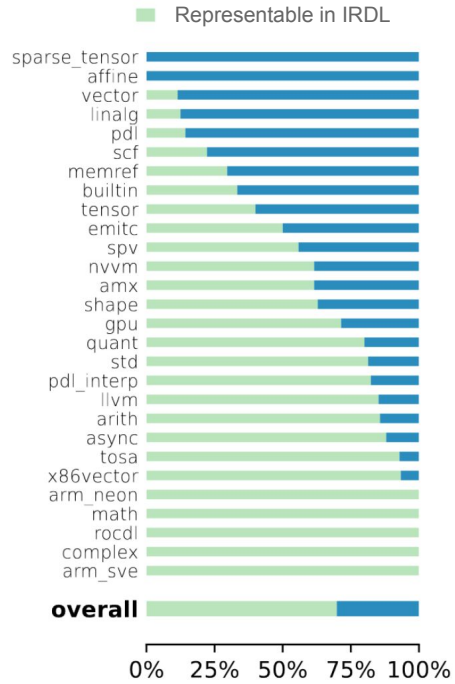


Most operations do not require additional C++

Structural Definition



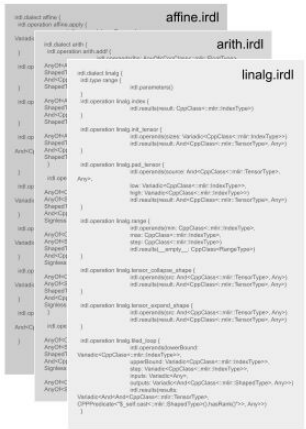
Verifier Definition



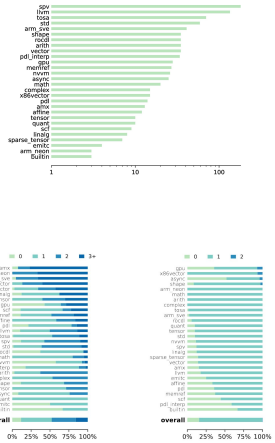
THE UNIVERSITY *of* EDINBURGH

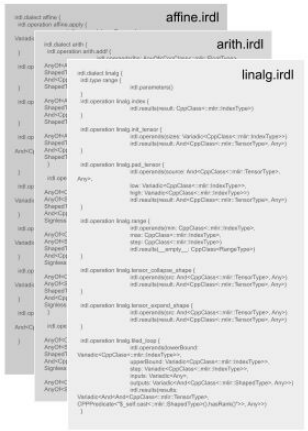
Conclusion

Conclusion



Conclusion





MLIR

Conclusion