IRDL: A Dialect for dialects

Mathieu Fehr, Théo Degioanni, and others

Wouldn't it be nice?

```
mlir-opt --dialect=cmath.irdl prog.mlir
```

```
irdl.dialect cmath {

  irdl.alias !any_float = !AnyOf<f32, f64>

  irdl.type complex {
    irdl.parameters (element_type: !any_float)
  }

  irdl.operation norm {
    irdl.constraintVar (T: !any_float)
    irdl.operands (c: !complex<T>)
    irdl.results (res: T)

    irdl.format "$c : $T"
  }
}
```

```
func @conorm(%p: !cmath.complex<f32>, %q: !cmath.complex<f32>)
       -> !cmath.complex<f32> {
    %norm_p = cmath.norm %p : !f32
    %norm_q = cmath.norm %q : !f32
    %pq = arith.mulf %norm_p, %norm_q : !f32
    return %pq : !cmath.complex<f32>
}
```

Defining a dialect in MLIR

ODS

```
def AddIOp : Op<Arith, "addi"> {
  let summary = "integer addition";
  let arguments = (ins Integer:$lhs,
                       Integer:$rhs);
  let results = (outs Integer:$res);
}
```

C++

```
class AddIOp : Op {
  // ...
  // ...
  // ...
}
```

EXE

MLIR Executable

COMPILE TIME

RUNTIME

???

?

C++

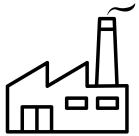
```
dialect.registerDynamicType(...)
dialect.registerDynamicAttr(...)
dialect.registerDynamicOp(...)
```

Our Goals

Concise

Introspectable

Dynamic



Generable

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

Our Goals

Concise

Generable

A **dialect** is exactly what we need

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

IRDL: IR Definition Language

```
irdl.dialect cmath {

  irdl.alias !any_float = !AnyOf<f32, f64>

  irdl.type complex {
    irdl.parameters (element_type: !any_float)
  }

  irdl.operation norm {
    irdl.constraintVar (T: !any_float)
    irdl.operands (c: !complex<T>)
    irdl.results (res: T)

    irdl.format "$c : $T"
  }
}
```

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

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

Types *have attribute parameters*.

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

IRDL Local Constraints

```
irdl.type complex {
  irdl.parameters (elementType: !any_float)
}
```

Type, Attribute, and Parameter constraints represent local structural invariants:

```
irdl.alias AnyComplex       = !complex<!any_float>
irdl.alias AnyComplex2      = !complex<!Any>
irdl.alias AnyComplex3      = !complex
irdl.alias ComplexF32       = !complex<!f32>
irdl.alias ComplexF32OrF64  = !complex<!AnyOf<!f32, !f64>>
irdl.alias ComplexNotF32    = !complex<!And<!any_float, Not<!f32>>>
```

Constraint Variables

```
irdl.attribute complex {
  irdl.constraint_var (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
```

Constraint variables require backtracking

```
irdl.operation test {
  irdl.constraint_var (T: !Any)
  irdl.operands(op: AnyOf<T, vector<T>>)
  irdl.results (res: T)
}
```

```
%res = test(%op) : (vector<f32>) -> f32
```

- Is `op` a `T` ? Yes, and `T` becomes `vector<f32>`.
- Is `res` a `T` (`vector<f32>`)? **No, backtracking.**
- Is `op` a `Vector<T>` of some `T` ? Yes, and `T` becomes `f32`.
- Is `res` a `T` (`f32`)? **Yes, success!**

Constraint variables require backtracking

```
irdl.operation test {
  irdl.constraint_var (T: !Any)
  irdl.operands(op: AnyOf<T, vector<T>>)
  irdl.results (re
}
```

Let's reduce the cost of these verifiers!

```
%res = test(%op) : (vector<f32>) -> f32
```

- Is `op` a `T` ? Yes, and `T` becomes `vector<f32>`.
- Is `res` a `T` (`vector<f32>`)? **No, backtracking.**
- Is `op` a `Vector<T>` of some `T` ? Yes, and `T` becomes `f32`.
- Is `res` a `T` (`f32`)? **Yes, success!**

IRDL-SSA

```
irdlssa.dialect cmath {
  irdlssa.type complex {
     %0 = irdlssa.is_type : f32
     %1 = irdlssa.is_type : f64
     %2 = irdlssa.any_of(%0, %1)
    irdlssa.parameters(%2)
  }

  irdlssa.operation norm {
     %0 = irdlssa.is_type : f32
     %1 = irdlssa.is_type : f64
     %2 = irdlssa.any_of(%0, %1)
     %3 = irdlssa.parametric : “cmath.complex”<%2>
    irdlssa.operands(%3)
    irdlssa.results(%2)
  }
}
```

Free optimizations

MLIR
Canonicalization

```
%0 = irdlssa.is_type : f32
%1 = irdlssa.is_type : f64
irdlssa.operands(%0)
```

```
%0 = irdlssa.is_type : f32
irdlssa.operands(%0)
```

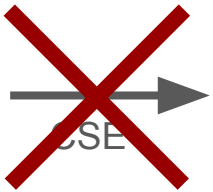
```
%0 = irdlssa.is_type : f32
%1 = irdlssa.parametric : complex<%0>
%2 = irdlssa.parametric : complex<%0>
irdlssa.operands(%1, %2)
```

```
%0 = irdlssa.is_type : f32
%1 = irdlssa.parametric : complex<%0>
irdlssa.operands(%1, %1)
```

Common Subexpression
Elimination (CSE)

Side-effect semantics

```
irdlssa.operation foo {
    %0 = irdlssa.is_type : f32
    %1 = irdlssa.is_type : f64
    %2 = irdlssa.any_of(%0, %1)
    %3 = irdlssa.any_of(%0, %1)
    irdlssa.operands(%2, %3)
}
```




```
irdlssa.operation foo {
    %0 = irdlssa.is_type : f32
    %1 = irdlssa.is_type : f64
    %2 = irdlssa.any_of(%0, %1)
    irdlssa.operands(%2, %2)
}
```

What about `(f32, f64)`?

Side-effect semantics

- `irdlssa.is_type` : no side effects
- `irdlssa.parametric` : no side effects
- `irdlssa.and` : no side effects
- `irdlssa.any` : **side effects!**
- `irdlssa.any_of` : **side effects!***

CSE will not pick up results that can come from side effects.

Domain-specific rewrites

```
%0 = irdlssa.is_type : f32
%o = irdlssa.parametric : “foo”<%0>
```

```
%o = irdlssa.is_type : “foo”<f32>
```

```
%2 = irdlssa.parametric : “foo”<%0>
%3 = irdlssa.parametric : “foo”<%1>
%o = irdlssa.any_of(%2, %3)
```

```
%2 = irdlssa.any_of(%0, %1)
%o = irdlssa.parametric : “foo”<%2>
```

Domain-specific rewrites

```
%0 = irdlssa.is_type : f32
%o = irdlssa.parametric : “foo”<%0>
```

```
%o = irdlssa.is_type : “foo”<f32>
```

Let’s lower the verifiers even further!

```
%2 = irdlssa.parametric : “foo”<%0>
%3 = irdlssa.parametric : “foo”<%1>
%o = irdlssa.any_of(%2, %3)
```

```
%2 = irdlssa.any_of(%0, %1)
%o = irdlssa.parametric : “foo”<%2>
```

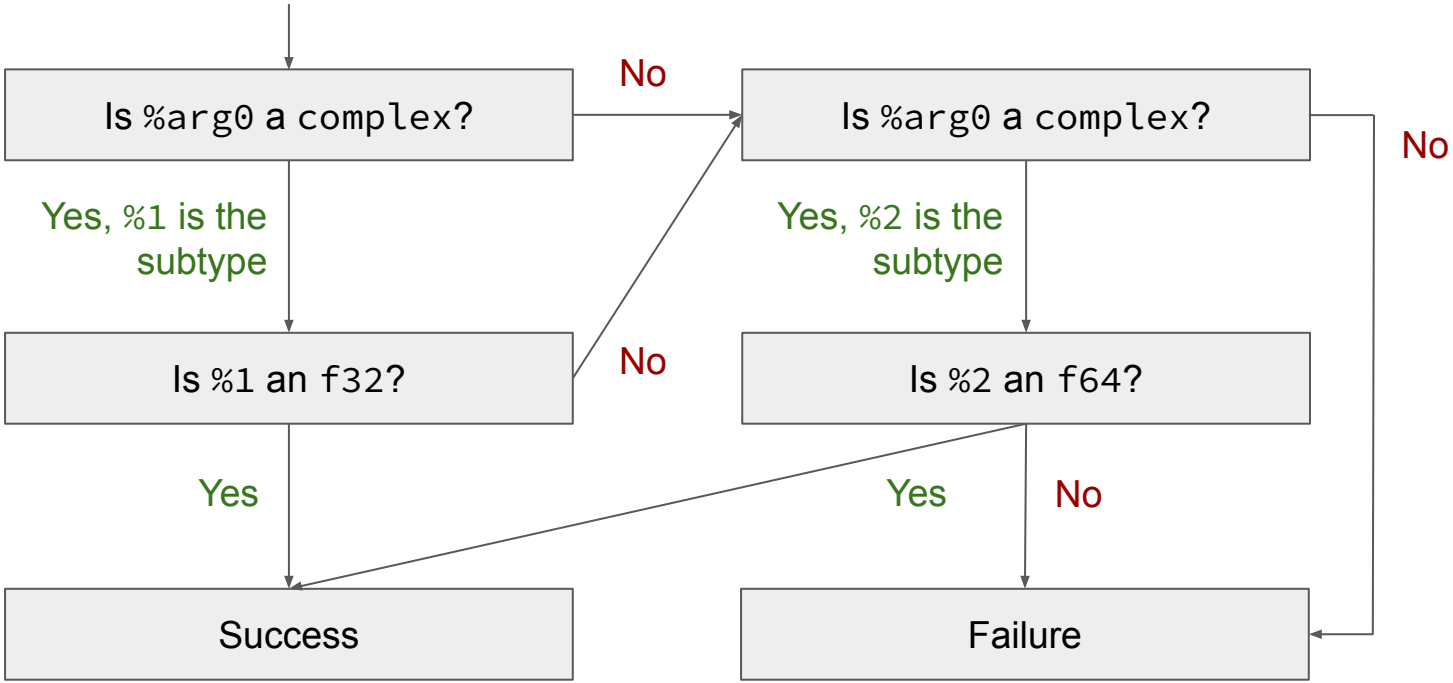
IRDL-Interp

`AnyOf<complex<f32>, complex<f64>>`

```
^bb0(%arg0: irdlinterp.type):
    irdlinterp.check_parametric(%arg0, “complex”, ^bb1, ^bb3)
^bb1(%1: irdlinterp.type):
    irdlinterp.check_type(%1, f32, ^bb2, ^bb3)
^bb2:
    irdlinterp.success
^bb3:
    irdlinterp.check_parametric(%arg0, “complex”, ^bb4, ^bb5)
^bb4(%2: irdlinterp.type):
    irdlinterp.check_type(%2, f64, ^bb2, ^bb5)
^bb5:
    irdlinterp.failure
```

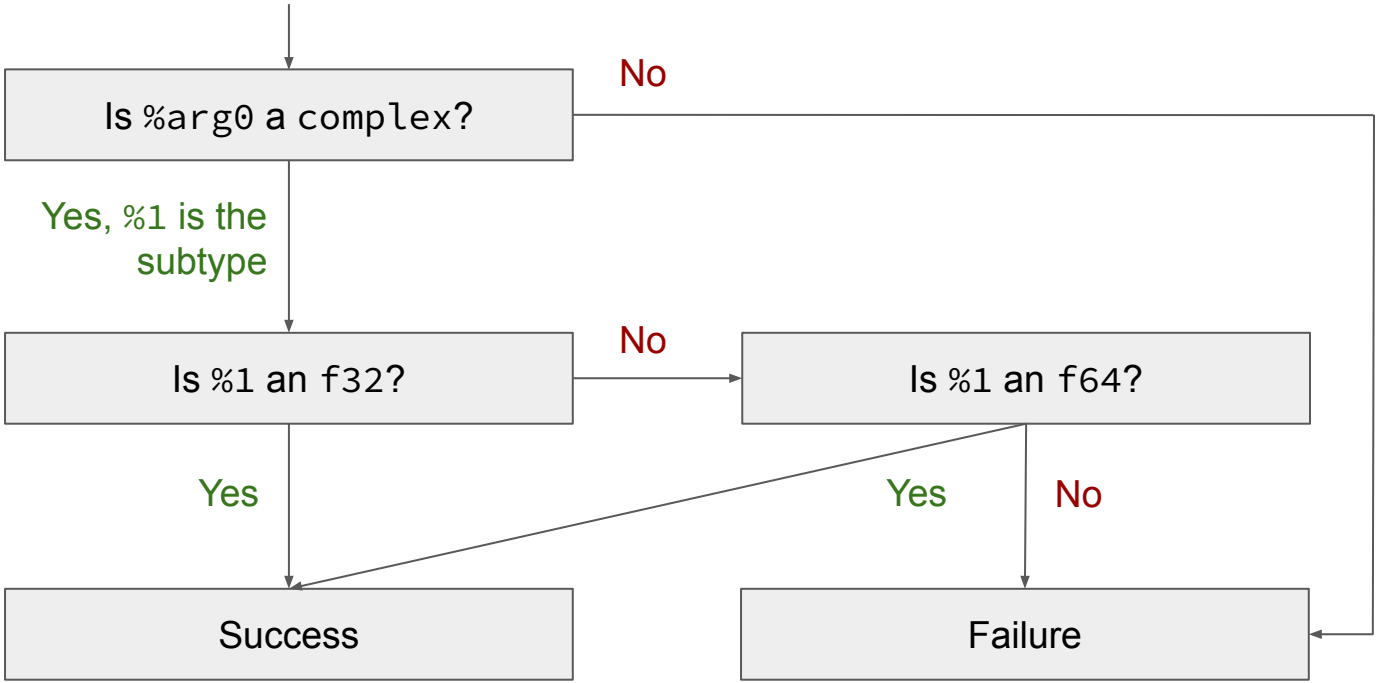
IRDL-Interp

`AnyOf<complex<f32>, complex<f64>>`



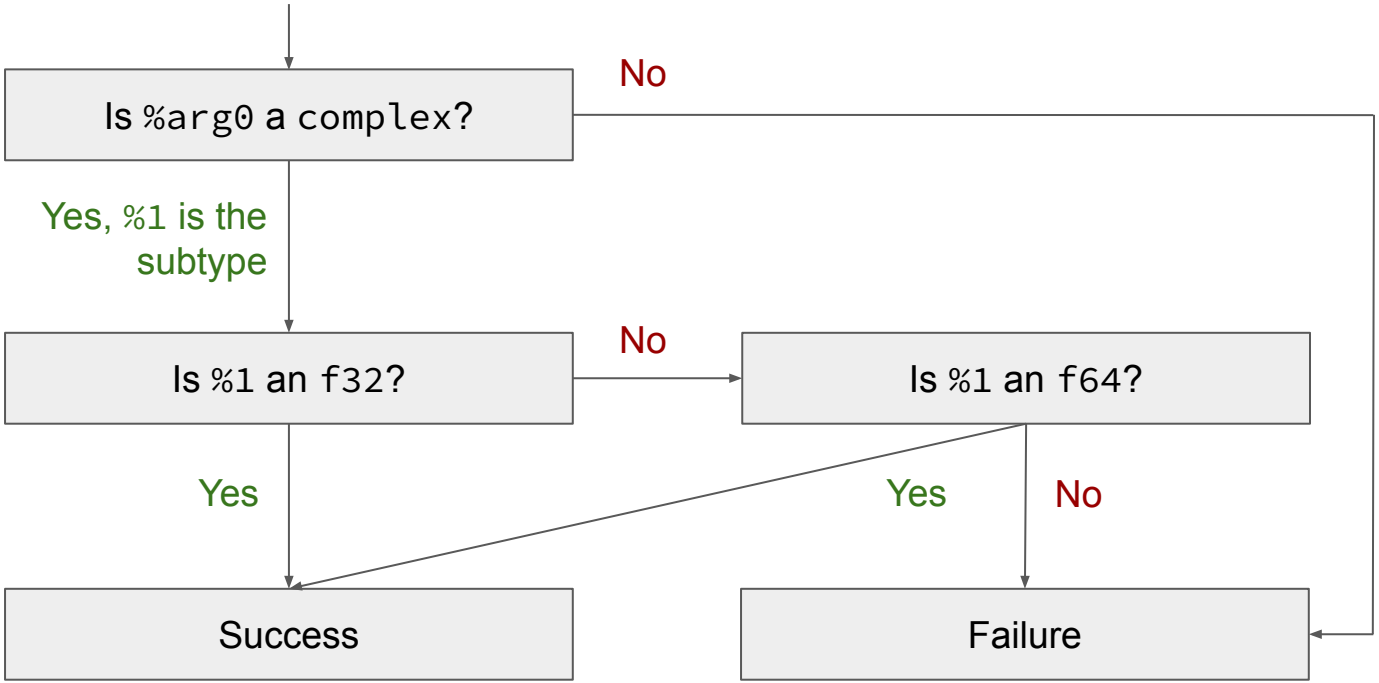
IRDL-Interp

`AnyOf<complex<f32>, complex<f64>>`



IRDL-Interp

`complex<AnyOf<f32, f64>>`



But there’s more!

```
lhs: AnyOf<complex<T>, matrix<T>>
rhs: T
```

```
(complex<f64>, f32)
```

- Is `lhs` a `complex` of some `T` ? Yes, `T` is `f64`.
- Is `rhs` an `f64`? **No, backtracking.**
- Is `lhs` a `matrix` of some `T` ? No, failure.

But there’s more!

```
lhs: AnyOf<complex<T>, matrix<T>>
rhs: T
```

```
(complex<f64>, f32)
```

- Is `lhs` a `complex` of some `T` ? Yes, `T` is `f64`.
- Is `rhs` an `f64`? **No, failure.**
- ~~Is `lhs` a `matrix` of some `T` ? No, failure.~~

No more backtracking in some cases.

Key takeaways

- IRDL allows to represent dialect definitions as MLIR programs
- These dialects can be registered at runtime from external languages
- IRDL generates efficient verifiers by lowering dialect definitions

Thank you!