Introduction to MLIR's ODS

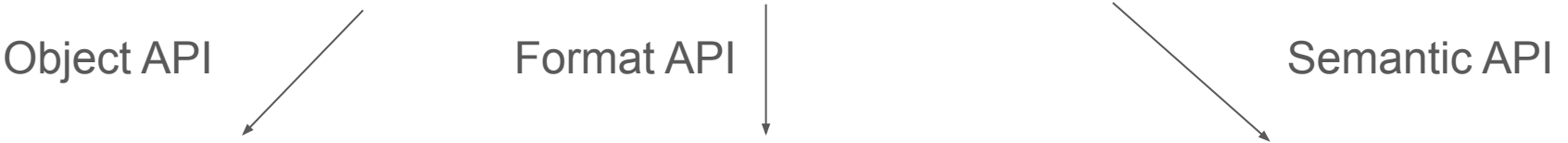
I.e. we get to define our own operations now!

Markus Böck (ETH Zurich)

ACM Europe School on MLIR 2026 – August 11, 2026

A new Operation requires a lot of C++ logic and boilerplate

```
%res = arith.addi %lhs, %rhs : i32
```



- Getters
- Setters
- Builders
- Verifier

- Parser
- Printer
- Serializer
- Deserializer

- Folders
- Rewrite Patterns
- Interface registration
- Trait registration
- …

Ca. 500 lines of C++ code

(we don't like this)

MLIR's ODS: A declarative syntax for defining Operations

- Framework for code generation based on LLVM's TableGen DSL
- `arith.addi` definition: 8 LOC (vs. 500 LOC of generated C++):

```
def Arith_AddIOp : Op<Arith_Dialect, "addi", [...]> {
 let arguments = (ins SignlessIntegerLike:$lhs,
                      SignlessIntegerLike:$rhs);
 let results   = (outs SignlessIntegerLike:$result);
 let assemblyFormat = "$lhs `,` $rhs attr-dict `:` type($result)";
}
```

Basics of TableGen: Classes and Definitions

```
class ExampleClass<int parameter> {
 int field_ = !add(parameter, 1);
 string otherField = ?;
}
```

Akin to C++ Classes

- Generators look for `def`s of these (e.g. `class Op`)
- Fields read by the Generator defined and **documented** there!
- Only **Code Reuse** construct

```
def ExampleDef : ExampleClass<5> {
 let otherField = "a value";
}
```

Akin to C++ Objects

- Generators (e.g. ODS) generate code per `def`
- `let`-statements used to set field values

TableGen Types and How to Create Them

TableGen Types	Literal Syntax	C++ equivalent
`int`	`0 -5 0x5`	`int64_t`

`string` / `code`	`"C style string"`
	`[{`
	` Raw string literal`
	`}]`
		`std::string`

`bit`	`0` / `1` `true`/`false`	`bool`
`dag`	`(op Value:$name…)`	N/A
`list<type>`	`[ ... ]`	`std::vector`
`ClassName`	`ClassName<...>  DefName`	`ClassName`

Exercise 1: Exploring the List Project Together!

```
$ git checkout exercises/day-2-session-1
$ git pull
# Build and run tests.
$ cmake --build build --target check-tutorial
```

```
include “mlir/IR/OpBase.td”
```

Defining Operations by overriding fields in the Op class

```
def List_EmptyOp : Op<List_Dialect, "empty">

// Base class for all ops.
class Op<Dialect dialect, string mnemonic,
         list<Trait> props = []> {
 // The dialect of the op.
 Dialect opDialect = dialect;

 // The mnemonic of the op.
 string opName = mnemonic;

 // The C++ namespace to use for this op.
 string cppNamespace = dialect.cppNamespace;
 …
}
```

```
// ListOps.h.inc
namespace mlir::list {
// Substring before the first `_` is ignored.
  class EmptyOp;
}
```

```
// roundtrip.mlir
%0 = list.empty ...
```

	Naming Convention
Op def-Name	PascalCase (C++ class matches!)
Op mnemonic	snake_case (sometimes camelCase)

include “mlir/IR/Constraints.td”

Defining Operations by overriding fields in the Op class

TypeConstraint

```
// In List_EmptyOp:
let results = (outs List_ListType:$result);

class Op<Dialect dialect, string mnemonic,
         list<Trait> props = []> {
 …
 // Dag containing the arguments of the op.
 // Default to 0 arguments.
 dag arguments = (ins);

 // The list of results of the op.
 // Default to 0 results.
 dag results = (outs);
 …
}
```

```
// ListOps.h.inc
namespace mlir::list {
  class EmptyOp : … {
  public:
     …
     TypedValue<ListType> getResult();
     …
  }
}
```

- TypeConstraint allows one or more type (or values of types) as result type or operand
- Extendable via C++ predicates

include “mlir/IR/CommonTypeConstraints.td”

The Most Commonly used TypeConstraints

TableGen Name	MLIR Syntax	C++ equivalent
I1, I8, ..., I64, AnyInteger	i1, i8,...,i64	bool, int8_t,...,int64_t
AnyType	N/A	template <class T>?
AnyTypeOf<[...]>	N/A	N/A
Any User defined Type! (List_ListType)	N/A	N/A
Optional<...>	N/A	std::optional<...>
Variadic<...>	N/A	std::vector<...>

Exercise 2: Implementing our First Operations!

- Implement the following Ops by creating fitting definitions in `ListOps.td`:

```
%1 = "list.is_empty"(%0) : (!list.list<ANY>) -> i1
"list.print"(%0) : (!list.list<ANY>) -> ()
%2 = "list.range"(%lb, %ub) : (i32, i32) -> !list.list<ANY>
%3 = "list.reverse"(%0) : (!list.list<ANY>) -> !list.list<ANY>
```

- Write tests for each in test/list/`roundtrip.mlir`! You will need to use the generic syntax shown here.
- Check your work using `cmake --build build --target check-tutorial`

Note: ANY above is a placeholder for allowing any element type.

Generic Syntax is Verbose and contains Redundancy

```
%2 = "list.range"(%lb, %ub) : (i32, i32) -> !list.list<ANY>
```

Verbose and non-informative

Redundant! (we only allow `i32` anyhow)

Instead we'd love:

```
%2 = list.range %lb to %ub : !list.list<ANY>
```

Declaratively defining Op syntax using `assemblyFormat`

```
%2 = list.range %lb to %ub : !list.list<ANY>
```

```
def List_RangeOp : List_Op<"range"> {
 let arguments = (ins I32:$lower, I32:$upper);
 let results = (outs List_ListType:$result);

 let assemblyFormat = [{
   $lower `to` $upper attr-dict `:` type($result)
 }];
}
```

Literals

- Describes syntax **after** the `list.range` prefix
- Requirements:
  - Every argument must be in the syntax
  - All types, except singletons, must be in the syntax using `type`
  - There always has to be a `attr-dict`

Type Deduction Reduces Redundant Types in Syntax

```
%3 = "list.reverse"(%0) : (!list.list<ANY>) -> !list.list<ANY>
```

Redundant! 1 would suffice

```
%3 = list.reverse %0 : !list.list<ANY>
```

```
// mlir/IR/OpBase.td:
class AllTypesMatch<list<string> names> …

def List_ReverseOp : List_Op<"reverse", [AllTypesMatch<["input", "result"]>]> {

 let arguments = (ins List_ListType:$input);
 let results = (outs List_ListType:$result);

 let assemblyFormat = "$input attr-dict `:` qualified(type($input))";
}
```

Only one of input or result needed

Common Syntax conventions for 0ps

- Follow `<arguments> attr-dict : <types>` convention

Do ✅

```
%lo = list.push_back %li, %item : !list.list<i32>, i32 -> !list.list<i32>
```

Don't 🚫

```
%lo = list.push_back %li : !list.list<i32>, %item : i32  -> !list.list<i32>
```

- Types usually defined as `<arg-types> -> <result-types>`

Don't 🚫

```
%lo = list.push_back %li, %item : !list.list<i32>, i32, !list.list<i32>
```

(`->` not necessary if one of `<arg-types>` or `<result-types>` can be deduced or is not present)

Exercise 3: Add `assemblyFormat` to your Ops

- Add a custom syntax to `is_empty`, `print`, `range` and `reverse`
  - Use the bare minimum of `type($name)` directives!
  - Make sure to update your tests! You can prototype your syntax in the tests as well
  - Follow conventions discussed previously
- Stretch Goal: Add `pop_front`, `push_front` and `peek_front`. Come up with a good `assemblyFormat` (same criteria as above)!

```
%1 = "list.peek_front"(%0) : (!list.list<ANY>) -> ANY
%2 = "list.pop_front"(%0) : (!list.list<ANY>) -> !list.list<ANY>
%3 = "list.push_front"(%item, %0) : (ANY, !list.list<ANY>) -> !list.list<ANY>
```

(Note: Look at `List_ItemTypeMatchesElementType` for even more powerful type deduction. Try to understand how it is implemented using `TypesMatchWith`)

Adding Regions to Operations using the `regions` field

```
class Op<Dialect dialect, string mnemonic,
         list<Trait> props = []> {
 …
  // The list of regions of the op.
  // Default to 0 regions.
  dag regions = (region);
 …
}
```

RegionConstraint	Description
AnyRegion	Unconstrained
SizedRegion<N>	Region with exactly N blocks
MinSizedRegion<N>	Region with at least N blocks

Common Traits used in conjunction with Regions

Trait	Description
`SingleBlockImplicitTerminator <"CppOpClassName">`	Enforces the presence of a specific Operation as `Terminator`
`Terminator`	Marks an Operation as a Terminator
`HasParent<"CppOpClassName"> ParentOneOf<[...]>`	Enforces that the parent Operation (the one owning the Region an Operation is contained in) is a specific Operation

```
def SCF_YieldOp : SCF_Op<"yield", [
 Terminator,
 ParentOneOf<["ForOp", "IfOp", "WhileOp"]>,
]>...
```

Exercise 4: Implementing `MapOp` and `YieldOp`

- Implement a `MapOp` (and `YieldOp`) which transforms every element of a list:

```
func.func @map(%list: !list.list<i32>) -> !list.list<i64> {
 %mapped = list.map %list -> !list.list<i64> with {
 ^bb0(%elem : i32):
   %ext = arith.extsi %elem : i32 to i64
   list.yield %ext : i64
 }
 return %mapped : !list.list<i64>
}
```

- Note: Everything in between the braces (`{}`) is part of the Region syntax (`$region` in `assemblyFormat`)

Not All Op Structure can be Defined in TableGen

```
func.func @map(%list: !list.list<i32>) -> !list.list<i64> {
  %mapped = list.map %list with {
  ^bb0(%elem : i16):
    …
  }
  return %mapped : !list.list<i64>
}
```

🚫 MapOp should not allow this!

Need to add extra logic in C++

Defining extra Verification in C++

```
def List_MapOp : List_Op<"map", [...]> {
  …
  // Verify before entering Regions.
  let hasVerifier = 1;
  // OR verify after entering Regions.
  let hasRegionVerifier = 1;
  …
}
```

```
/// ListOps.cpp
LogicalResult MapOp::verify() {
  …
}
```

```
/// ListOps.cpp
LogicalResult MapOp::verifyRegions() {
  …
}
```

Writing a Verifier in C++

```
LogicalResult MapOp::verify() {
 Block &body = getBodyBlock();
 if (body.getNumArguments() != 1)
   return emitOpError("expects the body to have exactly one argument");

 Type inputElementType = getInput().getType().getElementType();
 if (body.getArgument(0).getType() != inputElementType)
   return emitOpError("expects the body argument type (")
          << body.getArgument(0).getType()
          << ") to match the element type of the operand list ("
          << inputElementType << ")";

 return success();
}
```

Defining custom Syntax in C++

Exercise 7: Pretty MapOp