Making upstream MLIR more friendly to programming languages

Fabian Mora (U. of Delaware)
Mehdi Amini (NVIDIA)

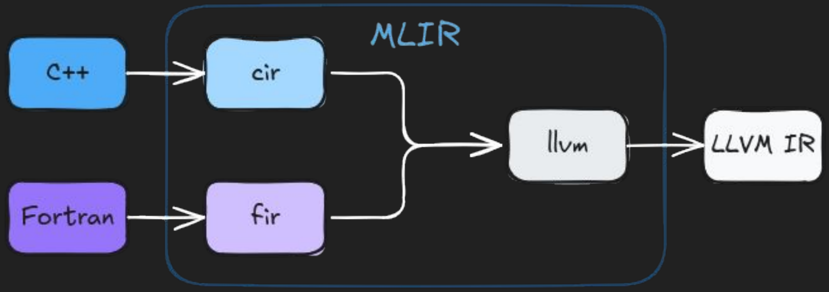
Motivation

ML compilers and MLIR

- MLIR has seen great success in ML compilers:
  - IREE
  - XLA
- Consequently, ML compilers have driven a lot of development in upstream MLIR

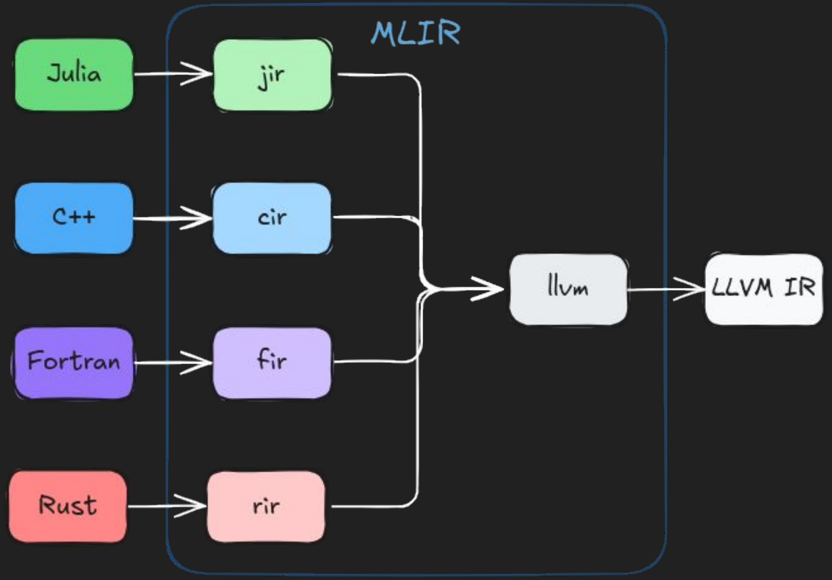
Flang and ClangIR

- Flang is getting closer to production
  - https://discourse.llvm.org/t/proposal-rename-flang-new-to-flang
- ClangIR recently got approved for upstreaming
  - https://discourse.llvm.org/t/rfc-upstreaming-clangir



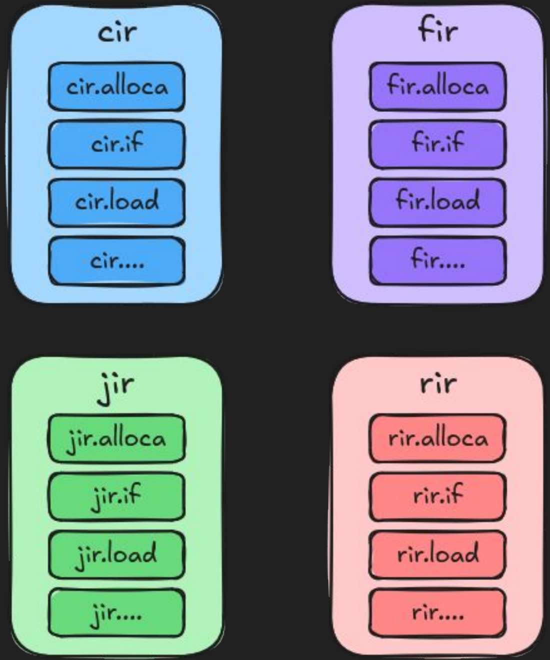
Beyond Flang and ClangIR

- We should expect more generic programming languages exploring MLIR



The burden to downstream compilers

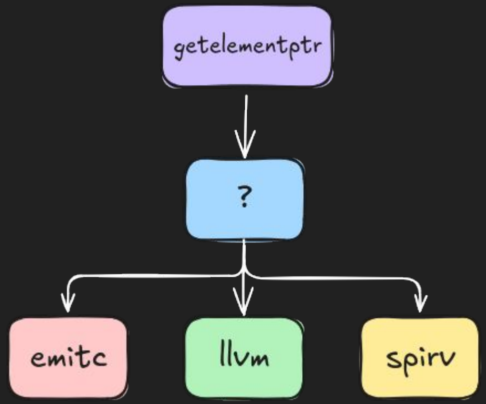
- There are many gaps in upstream MLIR for representing programming languages
- This leads to work duplication
  - Operations, eg. alloca, load, loop-like
  - Analyzes, eg. alias analysis, control-flow, variable lifetime
  - Transformations, eg. control-flow flattening



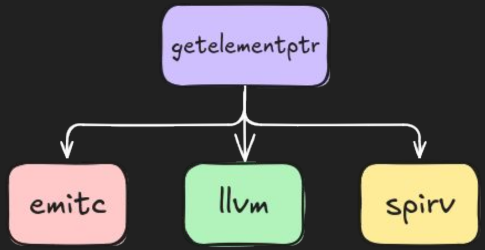
Limitations

Lack of high-level support for basic types

- Many basic constructs are representable only in target dialects
    - Eg. structs, allocas, load, stores
- Forces developers to choose a target / ABI early on the pipeline

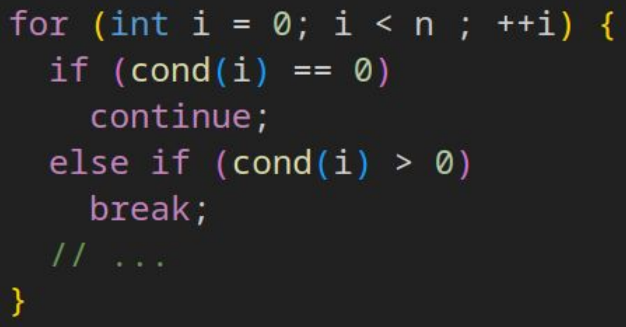


No high-level dialect for struct



No early exit control-flow

- Generic high-level control-flow is illegal per the language reference
    - continues, breaks, throws, are illegal
    - transformations and analyses fail on these



```
for (int i = 0; i < n ; ++i) {
  if (cond(i) == 0)
    continue;
  else if (cond(i) > 0)
    break;
  // ...
}
```

Illegal control-flow

https://discourse.llvm.org/t/rfc-region-based-control-flow-with-early-exits-in-mlir/76998

The `ptr` dialect: Modularizing LLVM ptr ops

The `ptr` dialect

A proposal to extract pointer related ops from LLVM into their own dialect:

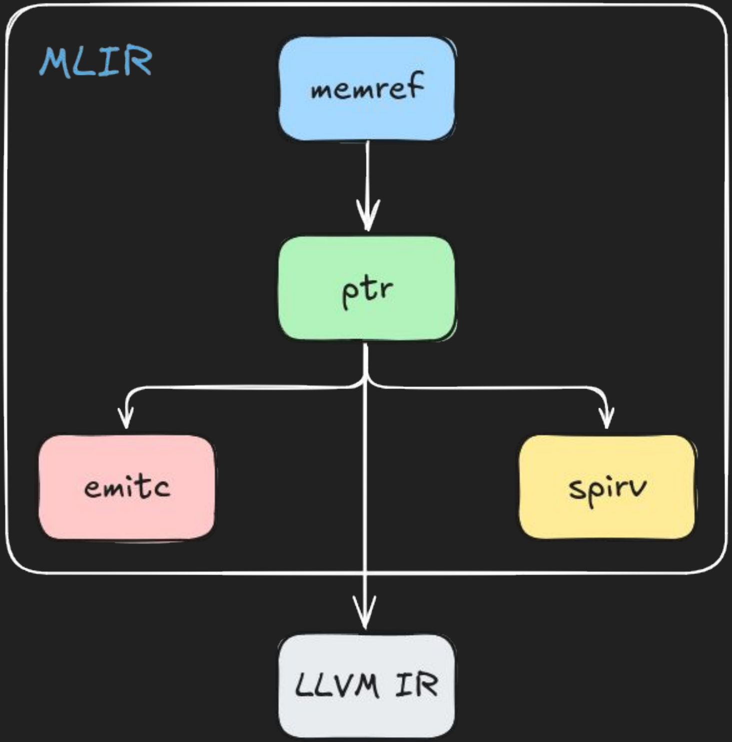
- work for higher-level types: pre-LLVM / pre-ABI lowering
- make them target independent: it'll work for non-LLVM backend (e.g. EmitC)
  - https://discourse.llvm.org/t/rfc-ptr-dialect-modularizing-ptr-ops-in-the-llvm-dialect

```
!ptr_rw = !ptr.ptr<#ptr.rw>
func.func @fill(%x: !ptr_rw
                %c: tensor<4xf32>, %n: i32) {
 %c0 = arith.constant 0 : i32
 %c1 = arith.constant 1 : i32
 scf.for %i = %c0 to %n step %c1 : i32 {
   %off_f32 = ptr.type_offset tensor<4xf32> : i32
   %off = arith.muli %i, %off_f32 : i32
   %x_off = ptr.ptradd %x, %off : !ptr_rw, i32
   ptr.store %x_off, %c : !ptr_rw, tensor<4xf32>
 }
}
```

Fill function for non-LLVM types

`ptr`: features and design

- Pointer operations on LLVM types will be translated directly into LLVM IR
- Conversions to SPIR-V and EmitC will be added

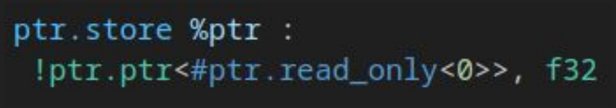


Lowerings of ptr

`ptr`: features and design

The memory space can introduce restrictions on operations:

- Eg. a memory space can be constant, and thus indicate that stores are illegal

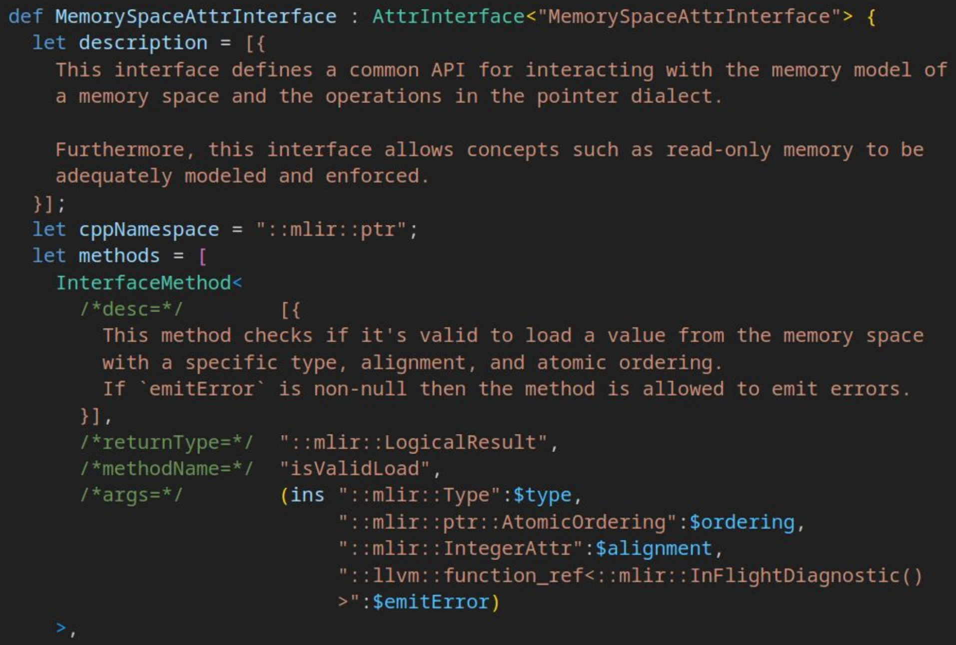


```
ptr.store %ptr :
  !ptr.ptr<#ptr.read_only<0>>, f32
```

Verification error

`ptr`: features and design

- `ptr` abstracts pointer semantics via encoding the properties of the memory space as an attribute interface
- The memory model is inspired by the LLVM memory model



```
def MemorySpaceAttrInterface : AttrInterface<"MemorySpaceAttrInterface"> {
  let description = [{
    This interface defines a common API for interacting with the memory model of
    a memory space and the operations in the pointer dialect.

    Furthermore, this interface allows concepts such as read-only memory to be
    adequately modeled and enforced.
  }];
  let cppNamespace = "::mlir::ptr";
  let methods = [
    InterfaceMethod<
      /*desc=*/        [{
        This method checks if it's valid to load a value from the memory space
        with a specific type, alignment, and atomic ordering.
        If `emitError` is non-null then the method is allowed to emit errors.
      }],
      /*returnType=*/  "::mlir::LogicalResult",
      /*methodName=*/  "isValidLoad",
      /*args=*/        (ins "::mlir::Type":$type,
                            "::mlir::ptr::AtomicOrdering":$ordering,
                            "::mlir::IntegerAttr":$alignment,
                            "::llvm::function_ref<::mlir::InFlightDiagnostic()
                            >":$emitError)
    >,
```

Attribute interface

`ptr`: features and design

- `ptr` has conversions operations to and from memref
- Allows turning the bare ptr conversion into a pass

```
func.call @foo(%memref): (memref<2x4xf64>)-> ()

// mlir-opt --apply-bare-ptr-convention
%ptr = ptr.from_memref %memref: memref<2x4xf64> ->
!ptr.ptr
func.call @foo(%ptr): (!ptr.ptr) -> ()
```

Bare ptr convention

`ptr`: performance impacts

- Test description:
    - Synthetic test with 100k operations converting memref to LLVM IR
- <3% LLVM translation performance impact

Test	Metric	Slowdown
Parse and print	Total time	0.9911
	Text Parser	0.9929
	Bytecode Output	0.9859
Convert to LLVM	Total time	1.0003
	Bytecode Parser	0.9912
	Bytecode Output	**1.0229**
	To LLVM	**0.9929**
	Canonicalize	**0.9933**
Translate to LLVM IR	Total time	1.0219
	To LLVMIR	**1.0392**

Modularizing LLVM Dialect

Modularizing LLVM Dialect

Some dialects are so close to LLVM semantics that they can be directly translated to LLVM IR:

- SCF dialect
- Arith dialect (when not operating on tensors)
- Ptr dialect (when operating on LLVM types)

=> Important for JITs: save compile-time by
avoiding unnecessary dialect conversions!

Modularizing LLVM dialect vs new dialects

Modularizing:

- Reduces pipeline complexity:
    - No: arith -> llvm dialect 1:1 conversion
- Help to solve missing abstractions for programming languages (like the `ptr` support for other types)
- Support for other target dialects like SPIR-V or EmitC
- What about divergence with LLVM?
    - LLVM-specific operations will be kept in LLVM Dialect

Modularizing LLVM dialect vs new dialects

Modularizing:

- Reduces pipeline complexity:
  - No: arith -> llvm dialect 1:1 conversion
- Help to solve missing abstractions for programming languages (like the `ptr` support for other types)
- Support for other target dialects like SPIR-V or EmitC
- What about divergence with LLVM?
  - LLVM-specific operations will be kept in LLVM Dialect

New dialects:

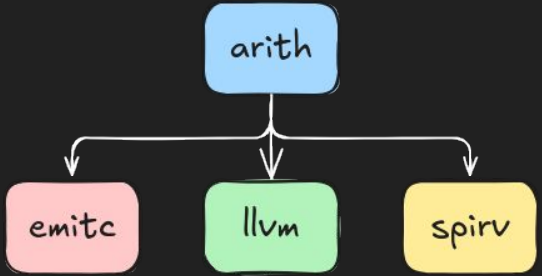
- Preserves LLVM Dialect as 1-1 mapping with LLVM IR
- Creates redundancy between dialects
  - Eg. arith and LLVM
  - Cost of documentation and redefining semantics
  - Canonicalizer duplication

What about arith? Can we remove redundancy?

- In most cases Arith and LLVM are redundant
  - Both operate on almost identical set of types
- Arith cannot be removed without hurting SPIR-V and EmitC
- What about removing arith ops from LLVM?

```
%a = arith.addi %b, %c overflow<nsw, nuw> : i32
%a = llvm.add   %b, %c overflow<nsw, nuw> : i32
```

Add in arith and LLVM



Arith lowerings

What about arith? Can we remove redundancy?

- We created a proof of concept arith translation to LLVM interface to test performance
- Test description:
  - Synthetic test with 600k operations translating arith + memref + func to LLVM IR
- arith -> llvmir is 1.5-1.85 faster than arith -> llvm -> llvmir

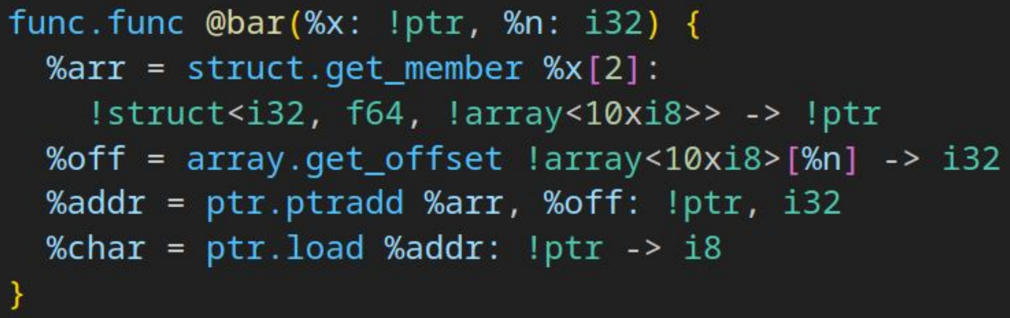
Metric	Speedup
Convert to LLVM	1.8457
Translate to LLVM IR	0.8845
To-LLVM + To-LLVMIR	1.4969

Test results

The road ahead/RFC

A proposal for further modularizing/creating dialects

- As with pointers, there are other primitive types without high-level dialect support:
  - struct
  - arrays
- These types should have non-target dialect support in upstream MLIR
- In most cases these are dialects with few operations



```
func.func @bar(%x: !ptr, %n: i32) {
  %arr = struct.get_member %x[2]:
    !struct<i32, f64, !array<10xi8>> -> !ptr
  %off = array.get_offset !array<10xi8>[%n] -> i32
  %addr = ptr.ptradd %arr, %off: !ptr, i32
  %char = ptr.load %addr: !ptr -> i8
}
```

Struct and array dialects

A PL dialect collection?

A collection of dialects for representing common programming language primitives, suitable to be emitted from various frontends.

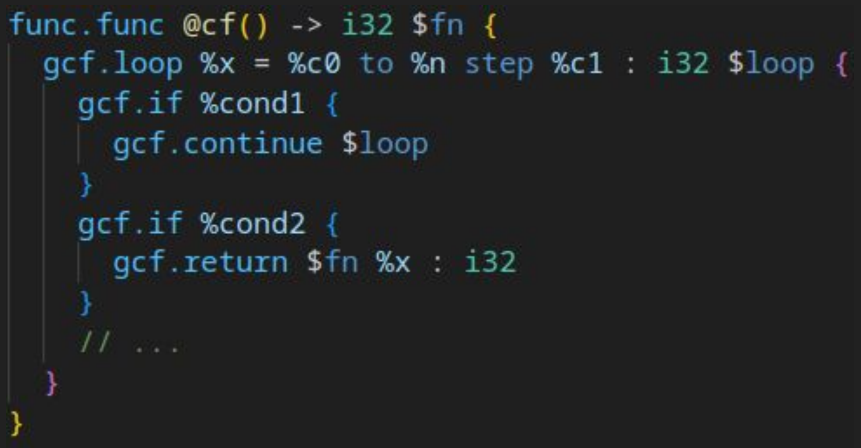
```
func.func @bar(%n: index) {
 // Low-level unique pointers
 %ptr = pl.alloc %n, f32: !ptr
 pl.at_scope_exit {
   pl.free %ptr: !ptr
 }
 // High-level exception handling
 pl.try {
   // ...
   %exc = pl.runtime_exception "runtime error"
   pl.throw %exec
 } catch(%exc: !pl.exception) {
   // ...
 }
}
```

PL example

General structured control flow

Programming Languages need early-exit support:

- Support should keep IR overhead low
- Control-flow from ops to ancestors seems a good tradeoff
- Think of Interfaces to model the specifics
- Impact on analysis? Dominance, etc.



```
func.func @cf() -> i32 $fn {
  gcf.loop %x = %c0 to %n step %c1 : i32 $loop {
    gcf.if %cond1 {
      gcf.continue $loop
    }
    gcf.if %cond2 {
      gcf.return $fn %x : i32
    }
    // ...
  }
}
```

Generic control-flow

https://discourse.llvm.org/t/rfc-region-based-control-flow-with-early-exits-in-mlir/76998

Target ABI abstraction

ABI abstraction should be a long term goal:

- Promoting clang ABI handling into MLIR target codegen abstractions.
- Mirror C type system in MLIR to implement the itanium C++ calling convention without requiring clang.

```
func.func @bar(%param: !struct<i32, i32>)
  -> !struct<!array<100xi32>>
  attributes {
  target = #x86.target
} // ...
// mlir-opt --aply-calling-convention
func.func @bar(%ret: !ref<!struct<!array<100xi32>>> {sret},
               %param: i64)
  attributes {
  target = #x86.target
} // ...
```

Target calling convention expansion

Questions?