1. Background

We need user-facing DSLs, because MLIR does not work as a front-end

- Python bindings are used to build MLIR
  - Need to manually insert operations, regions, etc.
  - Front-end features are missing: no type checking, no type inference
  - Attempts such as Neli [1] have different focus

- Using plain MLIR is difficult
  - Need to know the dialects
  - Have to write IR in SSA form

2. Building a Compiler

Before re-implement the whole stack

Next: target LLVM IR to only re-implement middle-ends

Now: target MLIR and customize dialects to only re-implement front-ends

3. Generating the Front-end in Python

High-level MLIR dialects are good enough, can we use them to... generate a front-end?

4. Mapping MLIR into Python

# MLIR types
W = TypeWrap("W", bound=int)
class FloatTypeGenericDef(W, FrontendTypeDef)

def bfloat16(a, b = None):
    return a + b if a is None else b

def fp32(a, b = None):
    return a + b if a is None else b

def fp64(a, b = None):
    return a + b if a is None else b

def fp16(a, b = None):
    return a + b if a is None else b

def int8(a, b = None):
    return a + b if a is None else b

# simple MLIR operations
W = TypeWrap("W", bound=int)
class FloatTypeGenericDef(W, FrontendTypeDef)

def add(input_1, input_2):
    return input_1 + input_2

def subtract(input_1, input_2):
    return input_1 - input_2

def multiply(input_1, input_2):
    return input_1 * input_2

def divide(input_1, input_2):
    return input_1 / input_2

# SSA Construction

We want to enable non-SSA front-ends. However, it can be challenging

- Producing SSA code directly is not easy and error-prone
- Can use memref dialect, but MLIR has no memref2reg for it yet [2]

Solution: desymref, a light version of memref2reg suitable for Python front-end

we define symref dialect. In symref dialect, each variable is associated with a symbol. This mimics how Python treats variables.

With symref code generation becomes easy. Re-assignments of different type can be treated as new variables (just like in Python) in absence of control flow.

5. SSA Construction

[desymref-decls]: Any declaration of some symbol A in SSA_CFG region, such that A is not used in the nested regions of any operations, can be pruned by SSA-construction algorithm.

[desymref-uses]: Any non-declared symbol A in a SSA_CFG region, such that it is not used in the nested regions of any operations, all uses can be pruned by SSA-construction algorithm apart from single fetch in entry block and single update in exit block.

[promise-symref]: For any operation O and a symbol A in its regions, such that for each region A is fetched once in entry block and updated once in exit block, both symbol uses can be pruned by creating a fetch and update to A around the operation O.

6. Conclusion & Future Work

In this poster we showed...

- How to map to auto-generation-friendly way a subset of MLIR into Python.
- A prototype of a non-SSA front-end which support compilation and can be extend to execution.

Open questions remain

- How to mix-in dialects with conflicting operations efficiently?
- List[int] as MLIR tensor: pass by value or pass by reference?
- Efficient auto-generation, FDL?
- How to deal with operations with regions, library calls (numpy) and attributes?
- How to structure the behaviour Python is the same during execution, e.g. integer overflow?