Forcefully Embedding MLIR into Python

by George Mitenkov

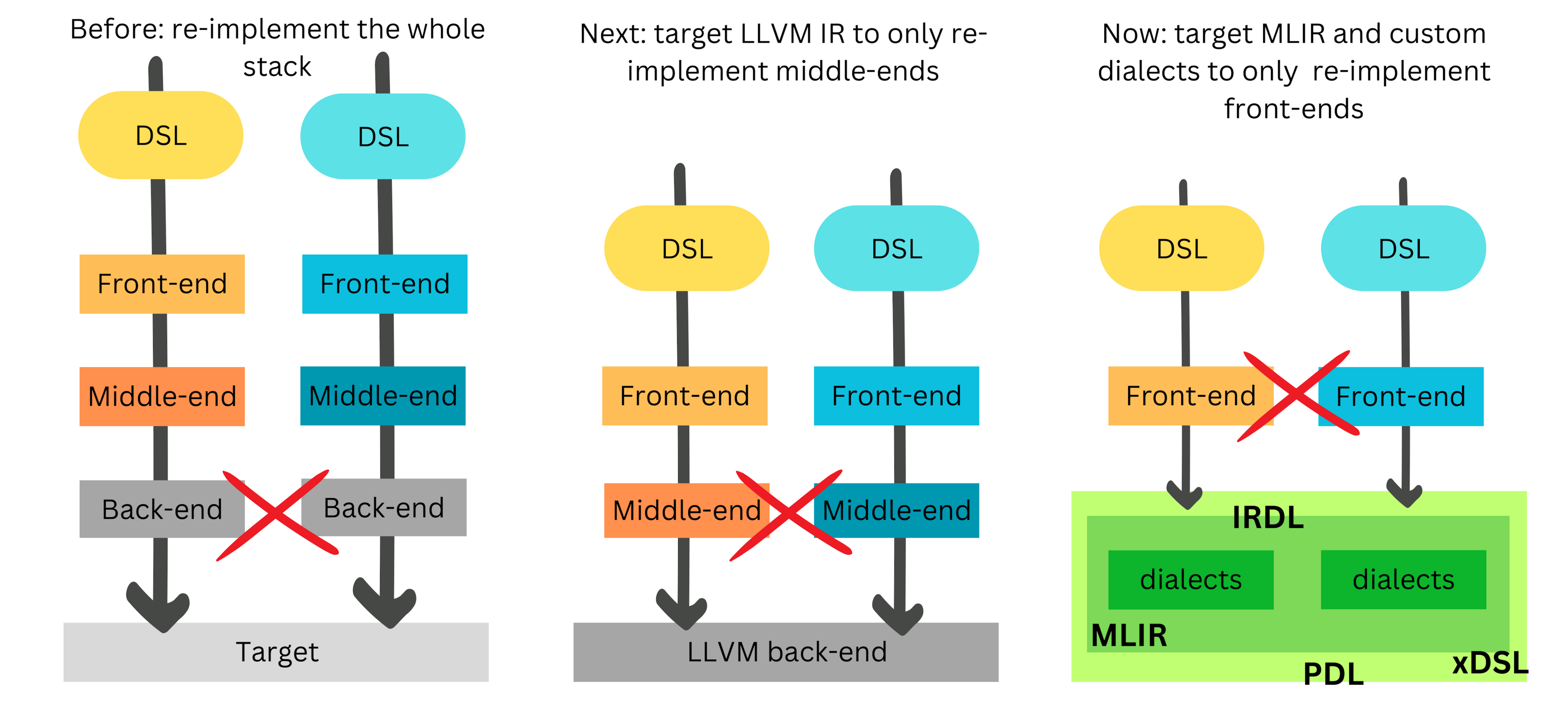
Try it out: https://github.com/xdslproject/xdsl/tree/main/xdsl/frontend

1. Background

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

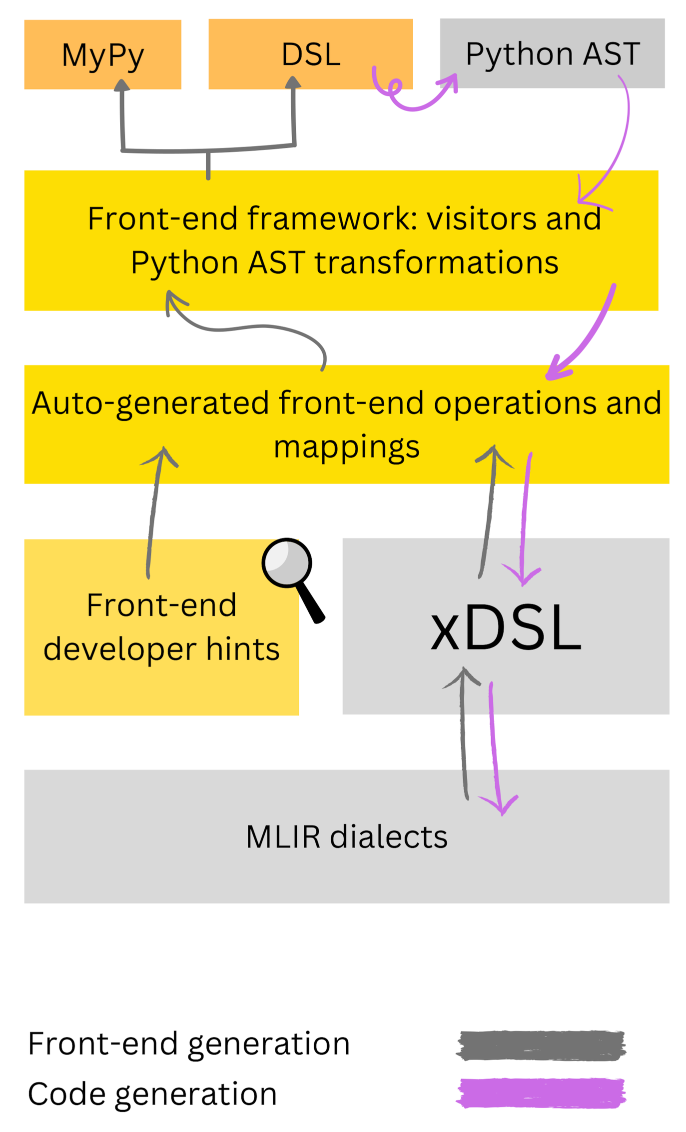
- Python bindings are used to *build* MLIR
  - Need to manually insert operations, regions, etc.
  - Front-end features are missing: no type checking, no type inference
  - Attempts such as Nelli [1] have different focus
- Using plain MLIR is difficult
  - Need to know the dialects
  - Have to write IR is in SSA form

2. Building a Compiler



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

3. Generating the Front-end in Python



[1]:
```python
p = FrontendProgram()
with CodeContext(p):
    def simple_sum() -> int:
        s: int = 0
        for i in range(5):
            s = s * i
        return s
```

Pythonic DSL written within code block: can extract Python AST for code generation. Also enables MyPy type checking!

Mapping MLIR types to Python types

Non-SSA form

[2]:
```python
p.compile()
print(p.module)
```

```
builtin.module {
  func.func @simple_sum() -> i32 {
    %s_0 = arith.constant 0 : i32
    %s = affine.for %i 0 to 5
            iter_args(%s_iter = %s_0) -> (i32) {
      %s_next = arith.muli %s_iter, %i : i32
      affine.yield s_next : i32
    }
    func.return %sum : i32
  }
}
```

Code can be statically compiled to MLIR

[3]:
```python
assert simple_sum() == 10
```

Or maybe even dynamically executed?

```
---------------------------------------------------------------------------
AssertionError                    Traceback (most recent call last)
Cell In[3], line 1
----> 1 assert simple_sum() == 10

AssertionError:
```

4. Mapping MLIR into Python

```python
# MLIR types
W = TypeVar("W", bound=int)
class FloatType(Generic[W], FrontendType):
    pass

f64 = FloatType[Literal[64]]
```

```python
# simple MLIR operations
W = TypeVar("W", bound=int)

@resolver
def addf(a: FloatType[W],  b: FloatType[W]) -> FloatType[W]:
    ...
```

```python
# front-end developer can map to Python
p = FrontendProgram()

p.alias(int, i64)
p.alias(list[int], TensorType[i64, Literal[-1]])
```

```python
W = TypeVar("W", bound=int)
class FloatType(Generic[W], FrontendType):

    def __add__(self: FloatType[W], other: FloatType[W]) -> FloatType[W]:
        return arith.addf(self, other)
```

```python
@block
def bb0(a: i32, b: i32):
    return bb1(a) if a != b else bb2(b)
@block
def bb3(a: i32, b: i32):
    return bb4(a + b, a * b)
```

```python
def foo(x: i32, c: i1):
    @block
    def entry(y: i32, d: i1):
        res = y
        if d:
            res += 2
        return res

    return entry(x, c)
```

```python
class ListType(Generic[V], FrontendType):
    @resolver
    @overload
    def __add__(self: ListType[V], other: ListType[V]) -> ListType[V]:
        ...

    @resolver
    @overload
    def __add__(self: ListType[V], other: V) -> ListType[V]:
        ...

    def overload__add__(self, other):
        if isinstance(other, ListType):
            return resolve__add__2(self, other)
        return resolve__add__1(self, other)
```

5. 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 *mem2reg* for it yet [2]

Solution: *desymref*, a light version of *mem2reg* suitable for Python front-end

[1]:
```python
p = FrontendProgram()
with CodeContext(p):
    def maybe_store_5(cond: i1) -> i32:
        b: i32 = 0
        if cond:
            b = 5
        return b

# keep symref output
p.compile(desymref=False)
print(p.module)
```

```
func.func @maybe_store_5(%cond: i1) -> i32 {
  symref.declare {symbol = "b"}
  %value = arith.constant 0 : i32
  symref.update %value : i32 {symbol = "b"}
  scf.if %cond {
      %new_value = arith.constant 0 : i32
      symref.update %new_value : i32 {symbol = "b"}
      scf.yield
  } else {
      scf.yield
  }
  %result = symref.fetch : i32 {symbol = "b"}
  func.return %result : i32
}
```

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

declare	=	*llvm.alloca*
update	=	*llvm.store*
fetch	=	*llvm.load*

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.

[desymref-decls]: Any declaration of some symbol A in SSACFG 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 SSACFG 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.

[promote-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.

```
func.func @claim1() -> i32 {
  symref.declare {symbol = "b"}
  %t1 = arith.constant 2 : i32
  symref.update %t1 : i32 {symbol = "b"}
  %t2 = symref.fetch :i32 {symbol = "b"}
  %t3 = arith.constant 3 : i32
  %t4 = arith.addi %t2, %t3 : i32
  symref.update %t4 : i32 {symbol = "b"}
  %t5 = symref.fetch :i32 {symbol = "b"}
  func.return %t5 : i32
}
```

```
func.func @claim1() -> i32 {
  %t1 = arith.constant 2 : i32
  %t3 = arith.constant 3 : i32
  %t4 = arith.addi %t1, %t3 : i32
  func.return %t4 : i32
}
```

```
symref.fetch {symbol = "b"}
%t1 = arith.constant 2 : i32
symref.update %t1 : i32 {symbol = "b"}
%t2 = symref.fetch :i32 {symbol = "b"}
%t3 = arith.constant 3 : i32
%t4 = arith.addi %t2, %t3 : i32
symref.update %t4 {symbol = "b"}
```

```
symref.fetch {symbol = "b"}
%t1 = arith.constant 2 : i32
%t3 = arith.constant 3 : i32
%t4 = arith.addi %t1, %t3 : i32
symref.update %t4 : i32 {symbol = "b"}
```

```
scf.if %cond {
  %x = arith.constant 0 : i32
  symref.update %x : i32 {symbol = "b"}
  scf.yield
} else {
  scf.yield
}
```

```
%b = symref.fetch : i32 {symbol = "b"}
%tmp = scf.if %cond -> (i32) {
  %x = arith.constant 0 : i32
  scf.yield %x
} else {
  scf.yield %b
}
symref.update %tmp : i32 {symbol = "b"}
```

desymrefy(op):

for r in op.regions:
for b in r.blocks:
for o in b.ops:
desymrefy(op)
	for r in op.regions:
	for b in r.blocks:
	for o in b.ops:
	promote_symref(op)
		No nested symbols beyond this point!	for r in op.regions:
			for b in r.blocks:
			for o in b.ops:
			desymref_decls(op)
				for r in op.regions:
				for b in r.blocks:
				for o in b.ops:
				desymref_uses(op)

6. Conclusion & Future Work

[1]:
```python
from frontend.dialects.riscv import malloc
import numpy as np

text: Const[..] = np.genfromtxt('col.txt',dtype='str')
def main():
    t0 = len(text)
    mem = malloc(t0)
    ...
```

Can evaluate expressions at compile-time, but how to embed the call?

Compiling is easy because we know it's a dialect call, what about execution?

[2]:
```python
def foo(xs: list[int]):
    xs[0] = 1
```

```
func.func @foo1(xs: tensor<?xi32>) -> tensor<?xi32> {
    // modification
    func.return %new : tensor<?xi32>
}
func.func @foo2(xs: memref<tensor<?xi32>>) {
  // modification
}
```

Pass by reference or by value?

[3]:
```python
with CodeContext(p)
    def bar(a: i8, b: i8) -> i8:
        return a + b
```

[4]:
```python
assert bar(5, 5) == 10
assert bar(200, 200) == 400
```

Both asserts succeed, overload add as checked add based on the type?

[5]:
```python
x = 0
scf.if_v1(True, [x := 2], [])
x = scf.if_v2(True, [2], [])
t, _ = scf.if_v3(True)
with t:
    x = 2
```

Walrus operator in else block overrides the value

Same as MLIR

Operations return regions, override __entry__ to give execution semantics

In this poster we showed ...

- How to map (in 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 ensure the behaviour of Python is the same during execution, e.g. integer overflow?

ETH zürich

[1]: https://github.com/makslevental/nelli
[2]: https://discourse.llvm.org/t/rfc-generic-mem2reg-in-mlir/69926