MLVM
Design

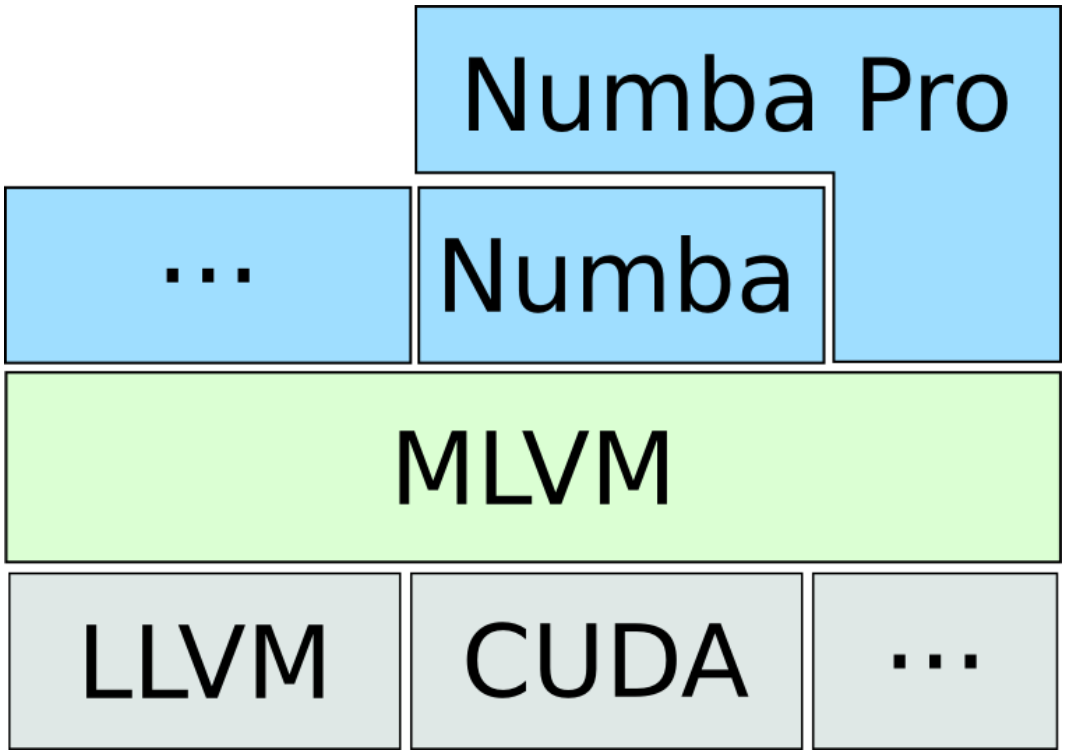
Goals

- Common codegen and execution management for Numba, and other projects...
- Abstraction on top of LLVM
- Separation of frontend and backend
- Share features through extensions

Design

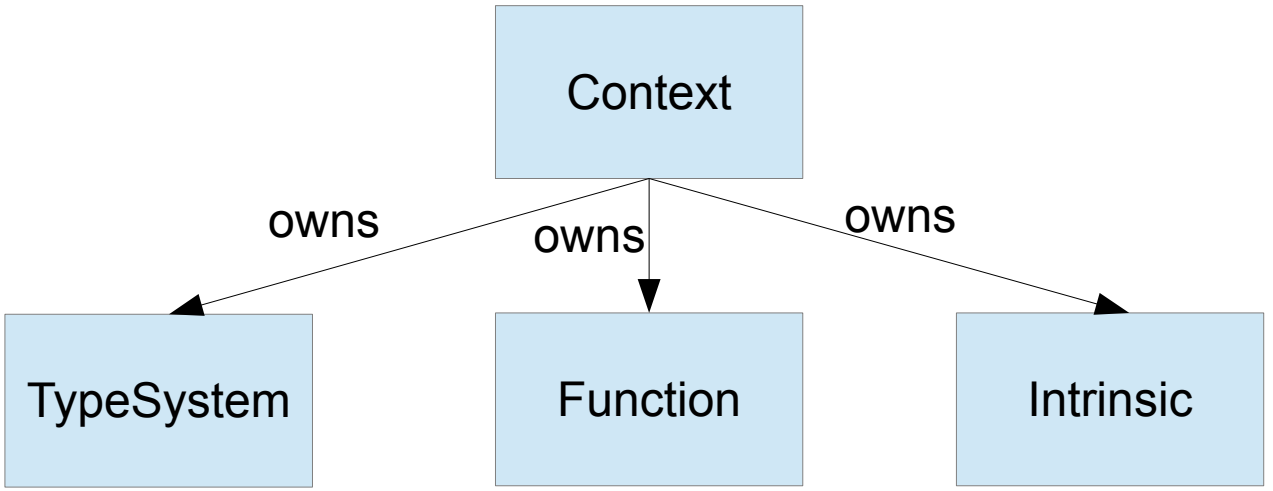
- Loose coupling of frontend and backend
- Provides basic default features
- Allows user to add features through extensions

Abstraction of Low-Level Detail



Frontend API

- Type is just a name
- Function can be overloaded
- Intrinsic is extensible



C-like IR

- Like C without most control-flow constructs
  - only branch is left
- User configurable implicit casting
- Printable for debugging
- Use IR Builder to populate functions
  - just like llvm-py

Extending the Frontend

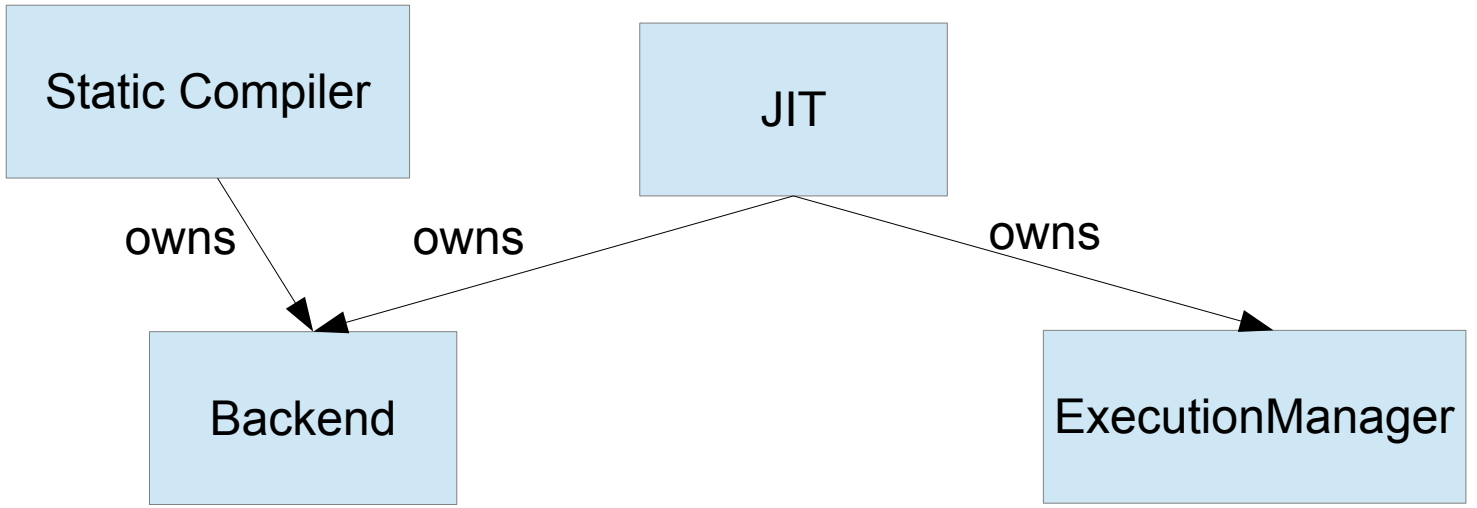
```
context = Context(TypeSystem())
context.install(ext_arraytype)
```

- This adds array-type extension
- Provides intrinsics:
  - array_load(ary, idx)
  - array_store(ary, val, idx)

```
define int32 foo (
        array_float        in %arg_0,
        array_float        in %arg_1,
        array_float       out %arg_2,
             int32        in %arg_3,
    )
{
  %const_0 = 0                                              ; int32
  %const_1 = 1                                              ; int32
  %const_2 = 1                                              ; int32
  %const_3 = 3.14                                           ; float
    %var_0 = %const_0                                       ; int32
block_0:
         br block_1
block_1:
        %9 = cmp.lt              %var_0,  %arg_3            ; pred
         br %9    [block_2, block_4]
block_2:
       %10 = cast.int32.address   %var_0                    ; address
       %11 = call.intr array_load %arg_0,  %10              ; float
       %12 = cast.int32.address   %var_0                    ; address
       %13 = call.intr array_load %arg_1,  %12              ; float
       %14 = add                  %11,  %13                 ; float
       %15 = mul                  %14,  %const_3            ; float
       %16 = cast.int32.address   %var_0                    ; address
            call.intr array_store %arg_2,  %15,  %16
block_3:
       ...
```

Backend API

- Backend implements compilation and linkage
- ExecutionManager implements JIT details
- JIT coordinates Backend and ExecutionManager
  - JIT can have multiple backends
- Static compiler



Compile for execution:

```
backend = LLVMBackend()

manager = LLVMExecutionManager()

 jit = JIT(manager, {'': backend})

python_callable = jit.compile(funcdef)
```

Install extension:

```
backend.install(ext_arraytype)
```

Static compilation:

```
backend = LLVMBackend(opt=LLVMBackend.OPT_MAXIMUM)

backend.install(ext_arraytype)

header = LLVMCWrapperGenerator()

compiler = Compiler(LLVMCompiler(), {'': backend},
                                    wrapper=header)

compiler.add_function(funcdef)

compiler.write_assembly(asmfile)

compiler.write_object(objfile)

header.write(headerfile)
```