PyDSL

A Python subset for a better MLIR programming experience

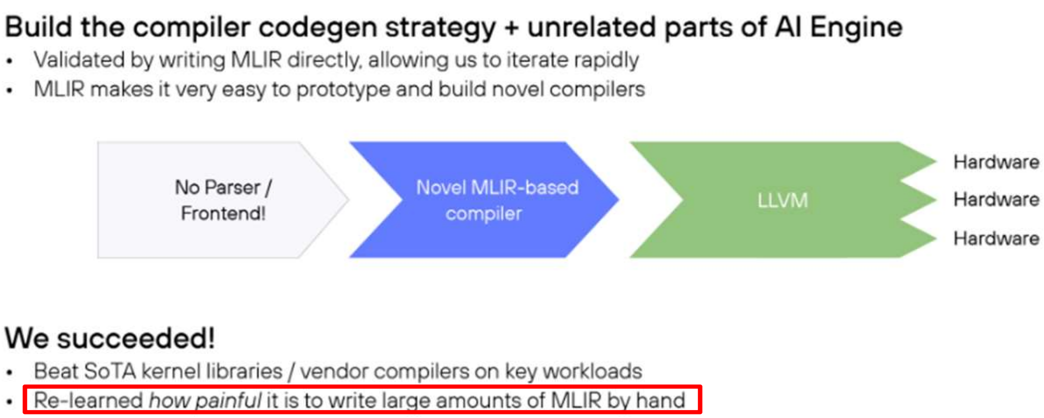
Kevin Lee (k323lee@uwaterloo.ca),

Kai-Ting Wang (kai.ting.wang@huawei.com)


```python
from pydsl.type import UInt32, F32, Index
from pydsl.memref import MemRefFactory, DYNAMIC
from pydsl.frontend import compile
from pydsl.affine import    \
    affine_range as arange, \
    affine_map as am,       \
    dimension as D,         \
    symbol as S

MemRefF32 = MemRefFactory((DYNAMIC, DYNAMIC), F32)

@compile(locals(), dump_mlir=True)
def jacobi(T: Index, N: Index, a: MemRefF32, b: MemRefF32) -> UInt32:
    dummy: UInt32 = 5

    for _ in arange(S(T)):
        for i in arange(1, S(N) - 1):
            for j in arange(1, S(N) - 1):
                const:F32 = 0.2
                b[am(D(i), D(j))] = (a[am(D(i), D(j))] +       \
                                     a[am(D(i), D(j) - 1)] +   \
                                     a[am(D(i), D(j) + 1)] +   \
                                     a[am(D(i) - 1, D(j))] +   \
                                     a[am(D(i) + 1, D(j))]) * const


        for i in arange(1, S(N) - 1):
            for j in arange(1, S(N) - 1):
                const: F32 = 0.2
                a[am(D(i), D(j))] = (b[am(D(i), D(j))] +       \
                                     b[am(D(i), D(j) - 1)] +   \
                                     b[am(D(i), D(j) + 1)] +   \
                                     b[am(D(i) - 1, D(j))] +   \
                                     b[am(D(i) + 1, D(j))]) * const

    return dummy
```

Why are we doing this?

- ❑ MLIR is *very verbose*. Even Mojo compiler engineers complain about it!
- ❑ MLIR is a compiler frontend's output. Language users shouldn't need to use it.
- ❑ MLIR's Python binding is also verbose: it's meant to be used by compiler engineers, not language users



Verbosity comparison between MLIR and PyDSL

These are the same programs

Existing work

Mojo

- modular.com/blog/mojo-llvm-2023
- Closed-source as of now
- Superset of Python: there are new keywords and syntax
- Promises bring-your-own-dialect
- Language itself is mostly independent of Python

```
struct MyPair:
  var first: Int
  var second: F32

  def __init__(self, first: Int, second: F32):
    self.first = first
    self.second = second
```

MLIR Python Util

- github.com/makslevental/mlir-python-utils/
- Not a compiler (in author's own words)
- Relies on runtime behavior: not very extensible for significant mismatch between Python and MLIR syntax (e.g. for loop yielding)
- Not yet support affine

```
K = 10
memref_i64 = T.memref(K, K, T.i64)

@func
@canonicalize(using=scf)
def memfoo(A: memref_i64, B: memref_i64, C: memref_i64):
    one = constant(1)
    two = constant(2)
    if one > two:
        three = constant(3)
    else:
        for i in range(0, K):
            for j in range(0, K):
                C[i, j] = A[i, j] * B[i, j]
```

MLL

- github.com/imv1990/mll
- Pythonic, but also has C-like syntax. Not a superset of Python.
- Also promises bring-your-own-dialect

```
func funcName(a : i32, b : i32, arr : array<10*20*i32>) -> i32{
  print (a, b)
  return (a + b + arr[0,0])
}

arr = array<10*20*i32>.dense(10)
print(funcName(10, 20, arr))
```

Overview of PyDSL

- Supports multiple MLIR dialects: arith, scf, func, memref, affine, transform
  - Each feature mapped Pythonically to the language
- Supports a function dedicated to sequentially transform the payload function
- Preliminary support for compiling and calling the function directly
- Static typing: mandatory type hinting
- Preliminary high-level typing support (ongoing work)

Behind the scenes

Let's look at a simple dummy example and how it is converted into MLIR. Note that some variables (e.g. `Memref64`, `i`) are not used.

```python
from pydsl.type import F32, F64, Index
from pydsl.memref import MemRefFactory
from pydsl.scf import range
from pydsl.frontend import compile

Memref64 = MemRefFactory((40, 40), F64)

@compile(locals(), dump_mlir=True)
def hello(a: F32, b: F32) -> F32:
    d: F32 = 12.5
    l: Index = 5

    for i in range(l):
        e: F32 = 3.0
        f = e + d

    return (a / b) + d
```

Behind the scenes

This is the annotation that kicks off the whole process. User include this to indicate that they want the function below to be compiled.
`hello(...)` is *never* executed by the runtime.

```
from pydsl.type import F32, F64, Index
from pydsl.memref import MemRefFactory
from pydsl.scf import range
from pydsl.frontend import compile

Memref64 = MemRefFactory((40, 40), F64)

@compile(locals(), dump_mlir=True)
def hello(a: F32, b: F32) -> F32:
    d: F32 = 12.5
    l: Index = 5

    for i in range(l):
        e: F32 = 3.0
        f = e + d

    return (a / b) + d
```

```
def compile(
        f_locals: dict[str, Any],
        transform_seq: Callable[[Any], None] | None = None,
        dump_mlir: bool = False,
        auto_build: bool = True,
        ) -> Callable[..., CompiledFunction]:
    """
    Compile the function into MLIR and lower it to a tempor

    The lowered function is a CompiledFunction object which

    f_locals: a dictionary of local variables you want the
    transform_seq: the function acting as the transform seq
    dump_mlir: whether or not to print out the MLIR output
    """

    def compile_payload(f: Callable) -> CompiledFunction:
        cf = CompiledFunction(f,
                              f_locals,
                              transform_seq=transform_seq
                              auto_build=auto_build)

        if dump_mlir:
            cf.dump_mlir()

        return cf

    return compile_payload
```

Initialize the variables we already defined

locals() is a built-in Python function that grabs every local variable that is defined before we start the compilation process. This lets users import keywords and define custom-shaped Memrefs

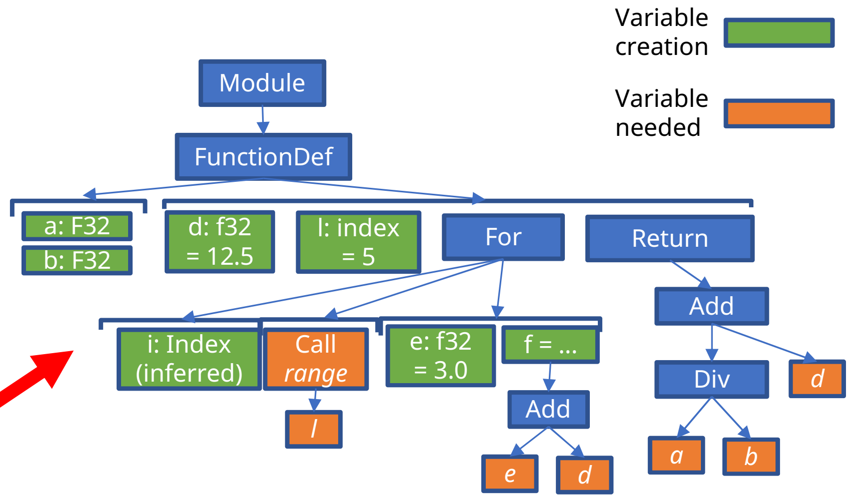
```python
from pydsl.type import F32, F64, Index
from pydsl.memref import MemRefFactory
from pydsl.scf import range
from pydsl.frontend import compile

Memref64 = MemRefFactory((40, 40), F64)

@compile(locals(), dump_mlir=True)
def hello(a: F32, b: F32) -> F32:
    d: F32 = 12.5
    l: Index = 5

    for i in range(l):
        e: F32 = 3.0
        f = e + d

    return (a / b) + d
```

```
{
 '__name__': '__main__',
 '__doc__': None,
 '__package__': None,
 '__loader__': <_frozen_importlib_external.SourceFileLoader object at 0xffff9
 '__spec__': None,
 '__annotations__': {},
 '__builtins__': <module 'builtins' (built-in)>,
 '__file__': '/home/k50036948/BishengCpp/new/llvm-lts/mlir/python/pydsl/examp
 '__cached__': None,
 'sys': <module 'sys' (built-in)>,
 'Path': <class 'pathlib.Path'>,
 'F32': <class 'pydsl.type.F32'>,
 'F64': <class 'pydsl.type.F64'>,
 'Index': <class 'pydsl.type.Index'>,
 'UInt16': <class 'pydsl.type.UInt16'>,
 'MemRefFactory': <functools._lru_cache_wrapper object at 0xfffeeb7d28d0>,
 'range': <class 'pydsl.scf.range'>,
 'compile': <function compile at 0xfffeeb51d120>,
 'Memref64': <class 'pydsl.memref.MemrefUnnamedSubclass'>
}
```

Those in the red box are used by our code.

Parse Python with Python

The first step of compilation is the parser. Lucky for us, Python has a library to let us parse Python into a tree



Simplified for visualization

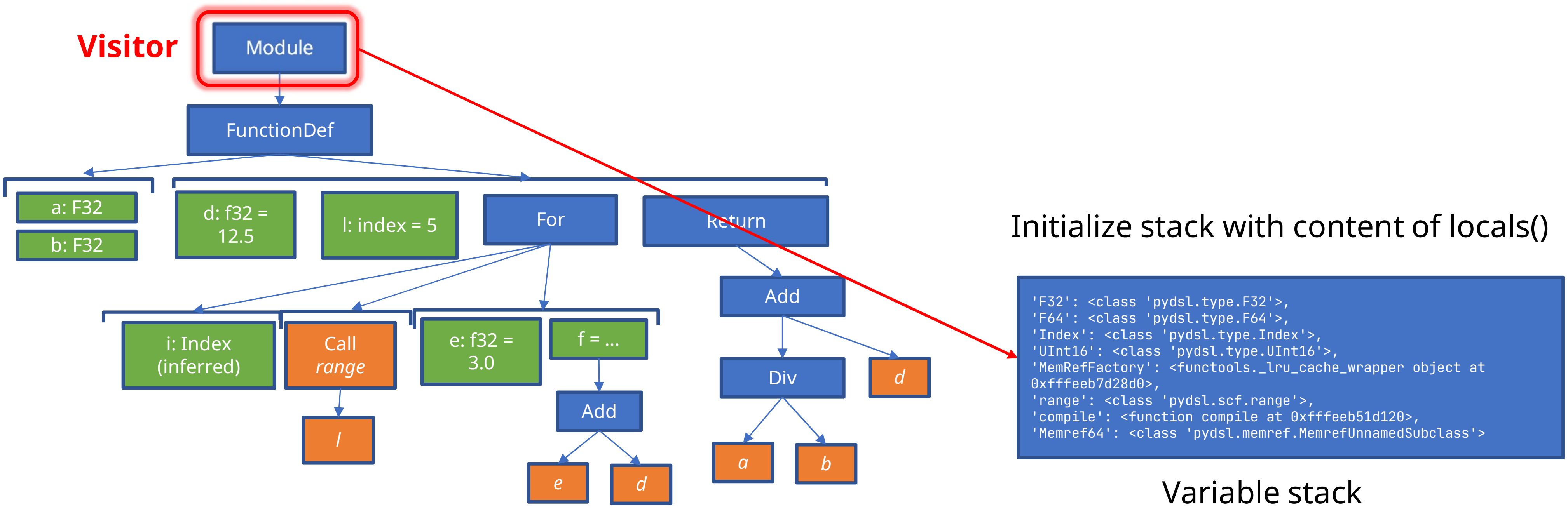
```python
from pydsl.type import F32, F64, Index
from pydsl.memref import MemRefFactory
from pydsl.scf import range
from pydsl.frontend import compile

Memref64 = MemRefFactory((40, 40), F64)

@compile(locals(), dump_mlir=True)
def hello(a: F32, b: F32) -> F32:
    d: F32 = 12.5
    l: Index = 5

    for i in range(l):
        e: F32 = 3.0
        f = e + d

    return (a / b) + d
```

```
ast.dump(ast.parse(
inspect.getsource(…)))
```

```
Module(body=[FunctionDef(name='hello', args=arguments(posonlyargs=[], args=[arg(arg='a',
annotation=Name(id='F32', ctx=Load())), arg(arg='b', annotation=Name(id='F32',
ctx=Load()))], kwonlyargs=[], kw_defaults=[], defaults=[]),
body=[AnnAssign(target=Name(id='d', ctx=Store()), annotation=Name(id='F32', ctx=Load()),
value=Constant(value=12.5), simple=1), AnnAssign(target=Name(id='l', ctx=Store()),
annotation=Name(id='Index', ctx=Load()), value=Constant(value=5), simple=1),
For(target=Name(id='i', ctx=Store()), iter=Call(func=Name(id='range', ctx=Load()),
args=[Name(id='l', ctx=Load())], keywords=[]), body=[AnnAssign(target=Name(id='e',
ctx=Store()), annotation=Name(id='F32', ctx=Load()), value=Constant(value=3.0),
simple=1), Assign(targets=[Name(id='f', ctx=Store())], value=BinOp(left=Name(id='e',
ctx=Load()), op=Add(), right=Name(id='d', ctx=Load())))], orelse=[]),
Return(value=BinOp(left=BinOp(left=Name(id='a', ctx=Load()), op=Div(), right=Name(id='b',
ctx=Load())), op=Add(), right=Name(id='d', ctx=Load())))], decorator_list=[],
returns=Name(id='F32', ctx=Load()), type_params=[])], type_ignores=[])
```

Initialize the variables

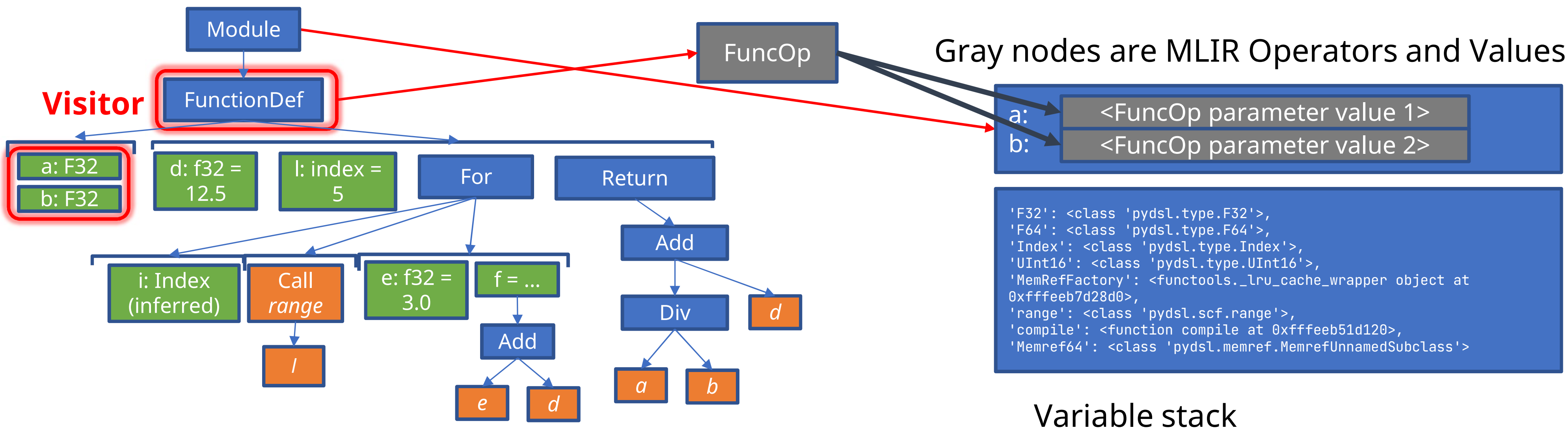
We emulate Python's variable stack. We also use a Visitor class to visit every node in the module.



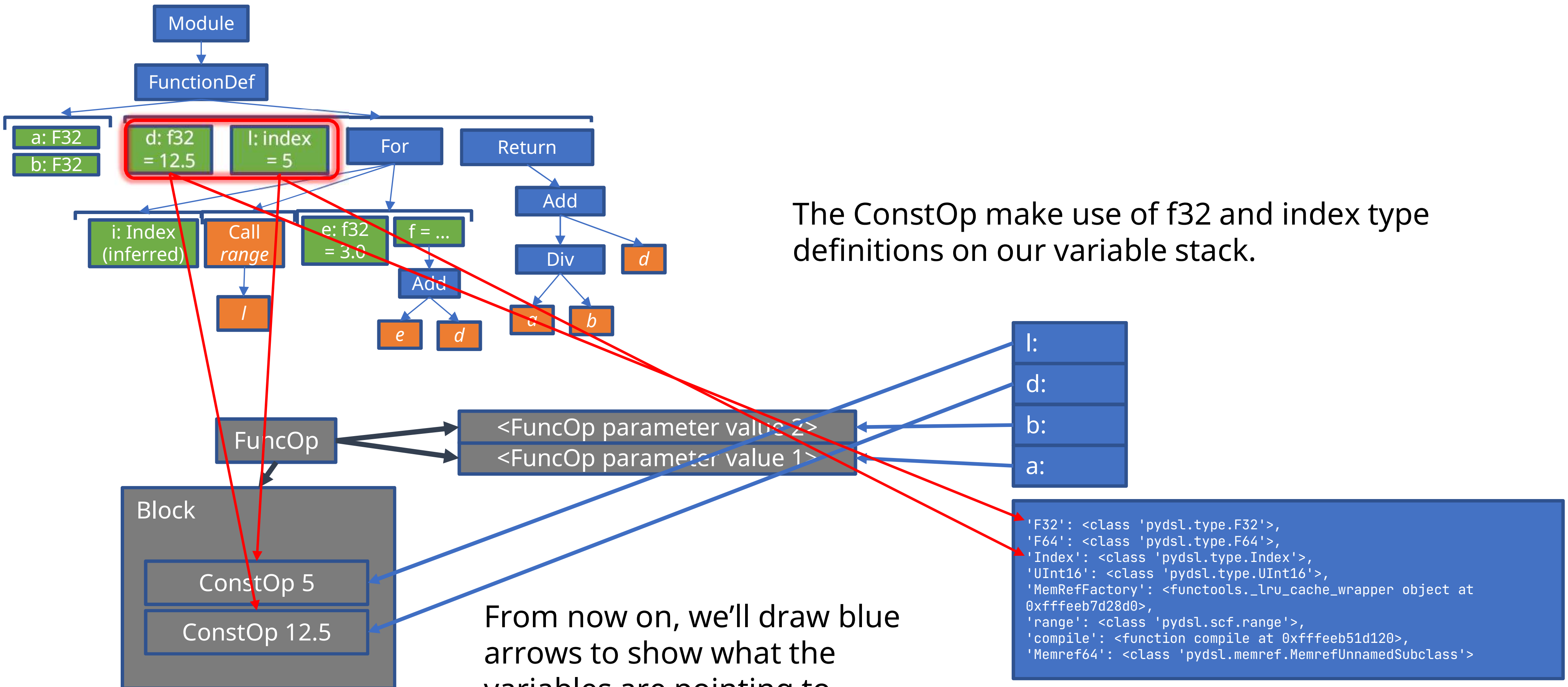
Visit all the nodes and emit MLIR

At a function, push a new scope onto the stack. As well, all Ops created are within the FuncOp block.

Some steps are skipped for conciseness.



Initialize these variables sequentially. These creates additional ConstOp within FuncOp



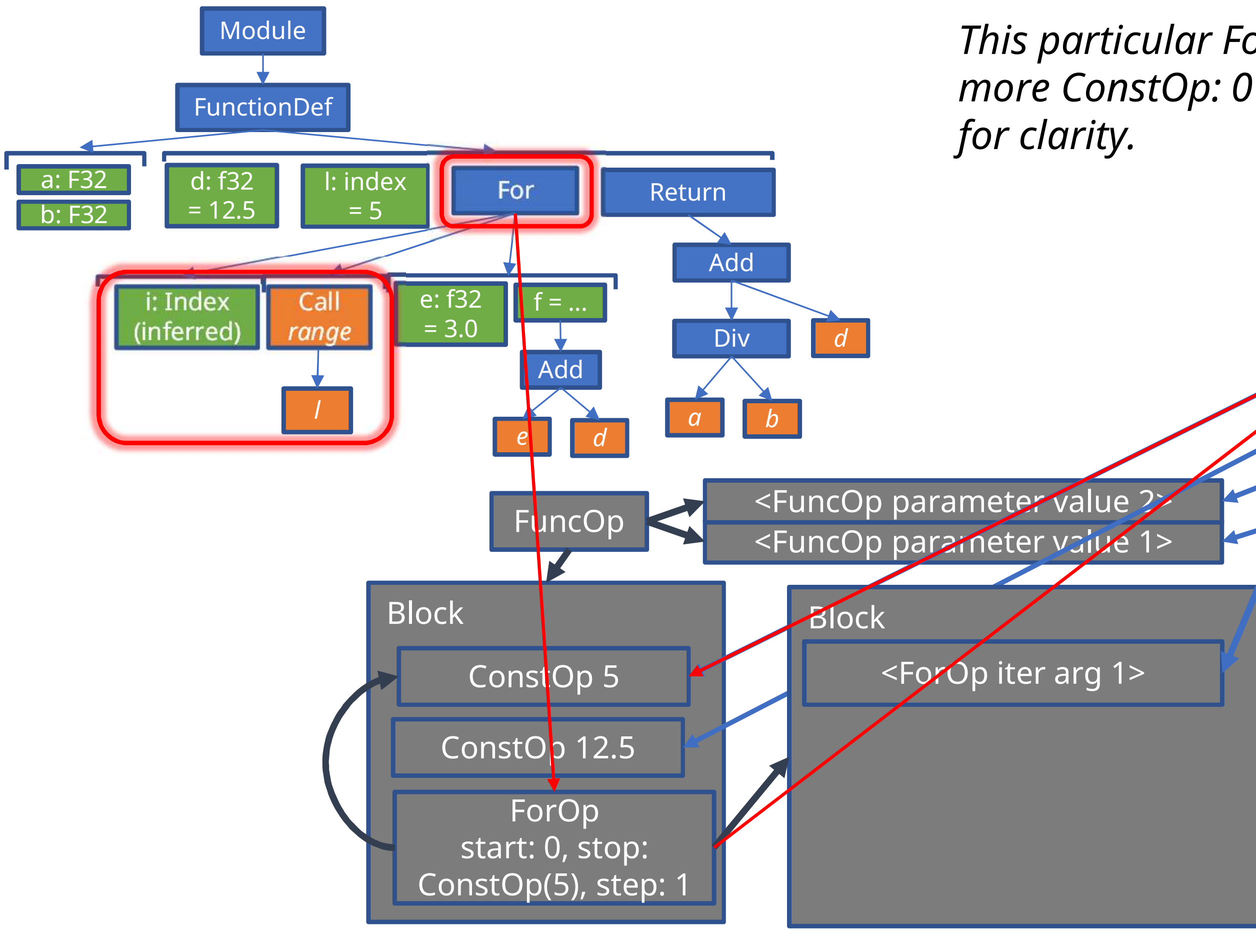
The ConstOp make use of f32 and index type definitions on our variable stack.

From now on, we'll draw blue arrows to show what the variables are pointing to

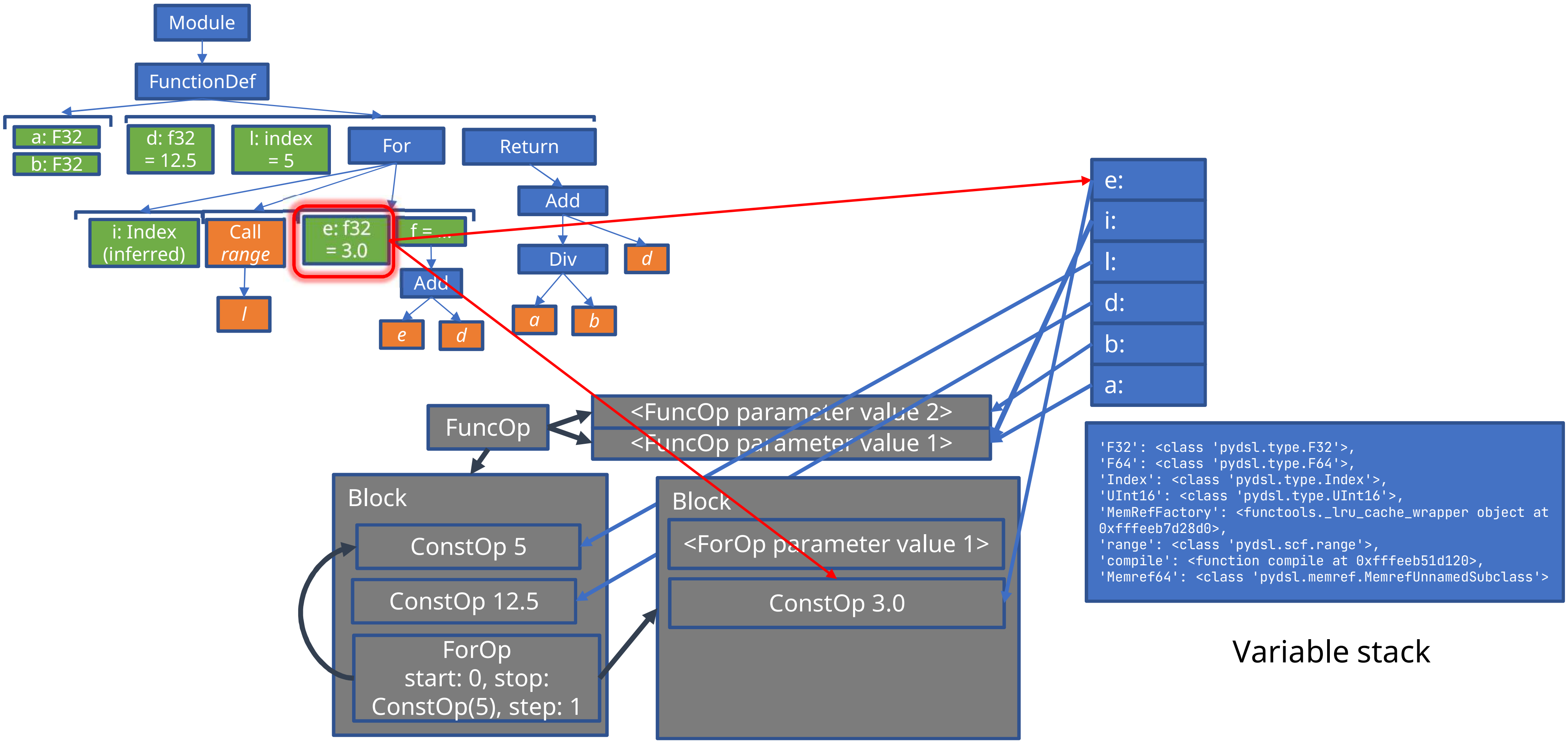
Variable stack

We determine what l is by querying the stack. Since l is ConstOp 5, ForOp now points to that ConstOp.

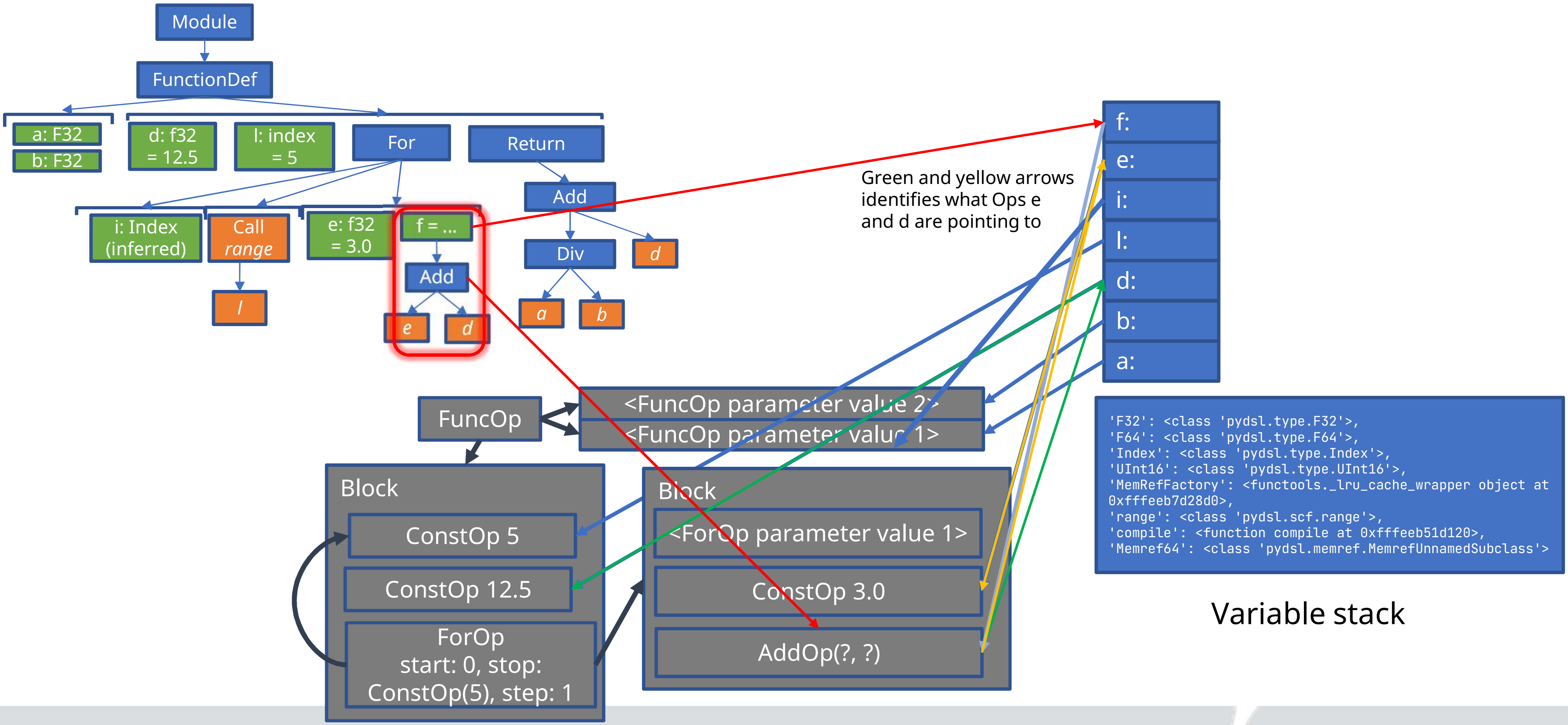
This particular For transformation actually also creates 2 more ConstOp: 0 for start and 1 for step. I omitted them for clarity.



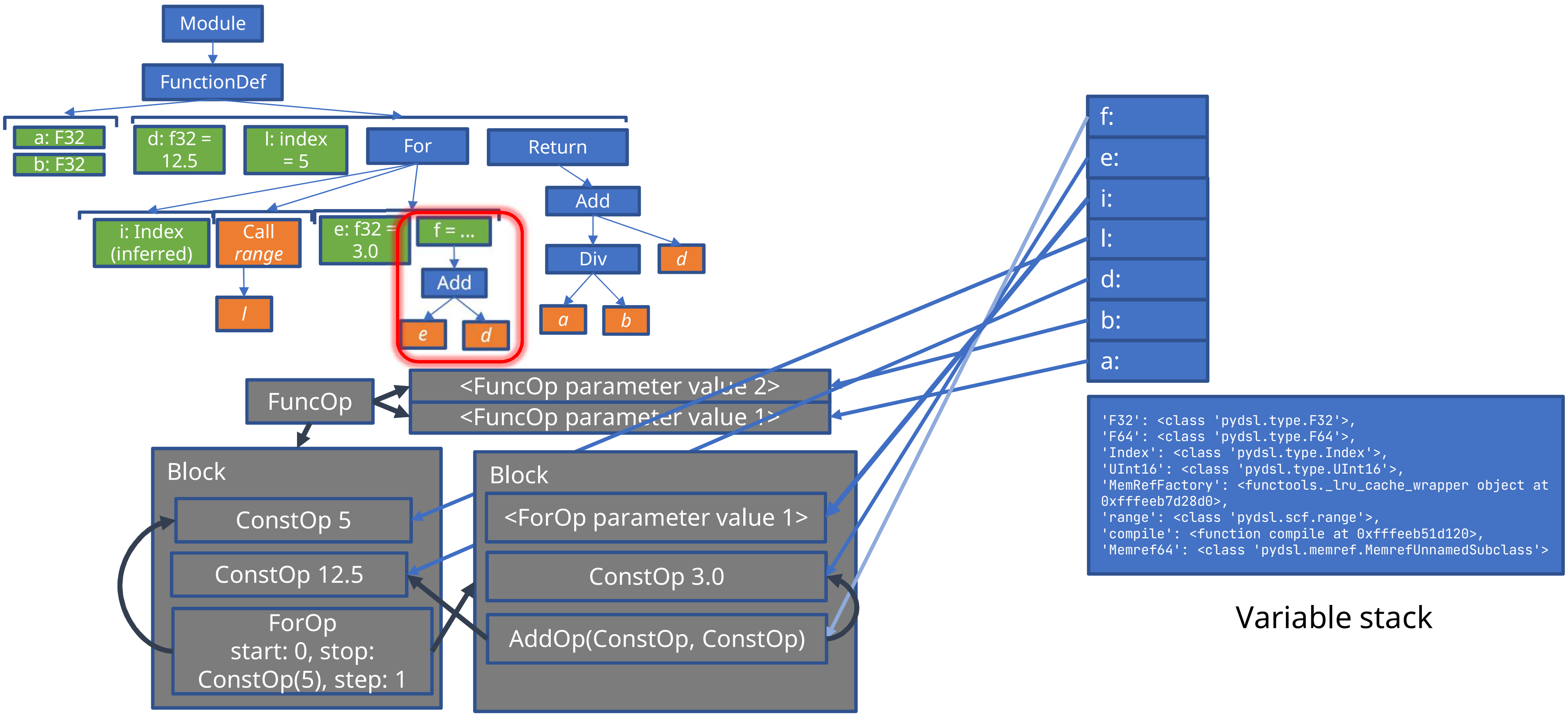
Variable stack


Module
FunctionDef
a: F32
b: F32
d: f32
= 12.5
l: index
= 5
For
Return
Add
i: Index
(inferred)
Call
range
e: f32
= 3.0
f = ...
Div
d
Add
l
e
d
a
b
e:
i:
l:
d:
b:
a:
FuncOp
<FuncOp parameter value 2>
<FuncOp parameter value 1>
Block
ConstOp 5
ConstOp 12.5
ForOp
start: 0, stop:
ConstOp(5), step: 1
Block
<ForOp parameter value 1>
ConstOp 3.0
'F32': <class 'pydsl.type.F32'>,
'F64': <class 'pydsl.type.F64'>,
'Index': <class 'pydsl.type.Index'>,
'UInt16': <class 'pydsl.type.UInt16'>,
'MemRefFactory': <functools._lru_cache_wrapper object at 0xfffeeb7d28d0>,
'range': <class 'pydsl.scf.range'>,
'compile': <function compile at 0xfffeeb51d120>,
'Memref64': <class 'pydsl.memref.MemrefUnnamedSubclass'>
Variable stack

When a variable is defined by arithmetic operation, it just refers to the Op.
The Op also checks the variable stack to see what other Ops it needs to point to

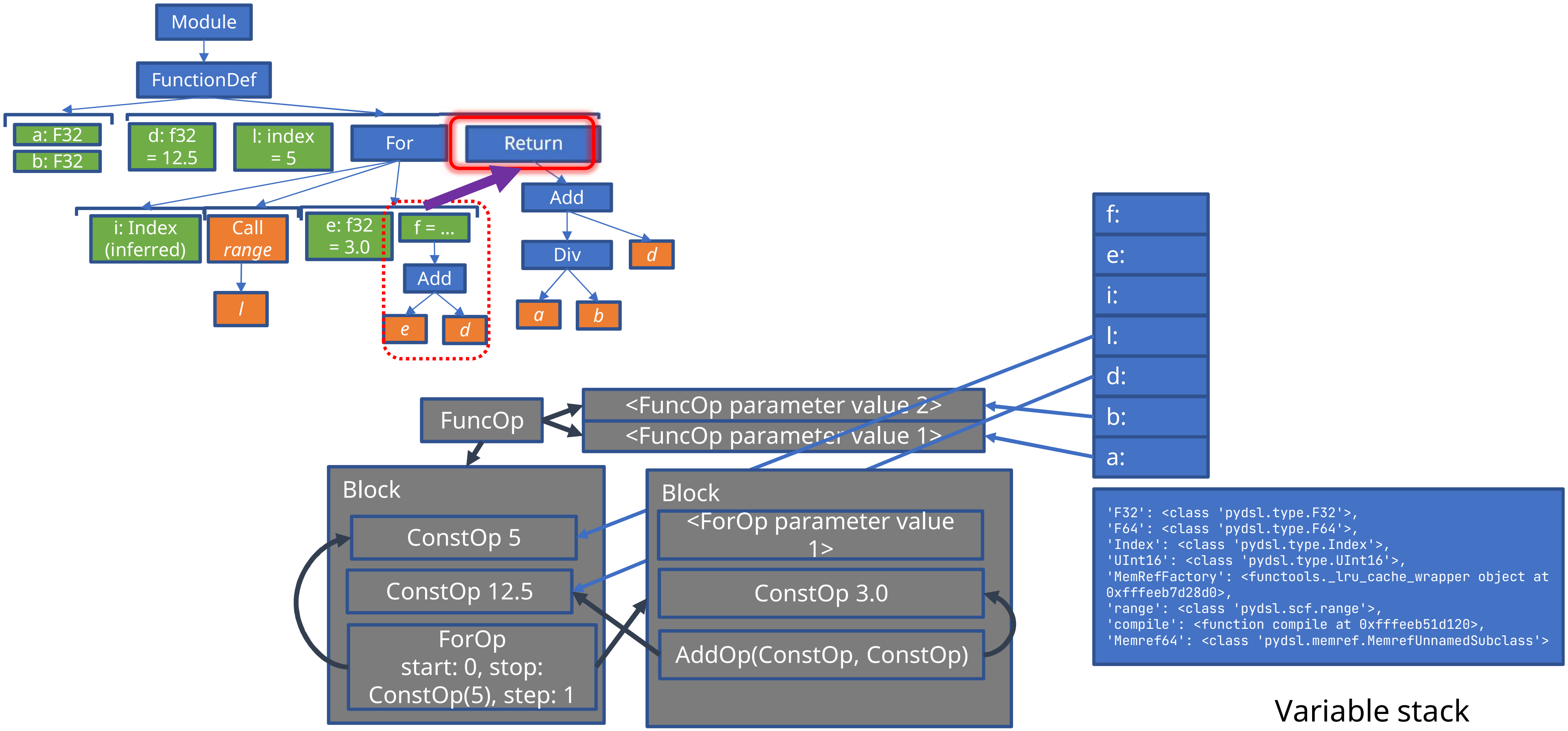


Thus…



Module
FunctionDef
a: F32
b: F32
d: f32 = 12.5
l: index = 5
For
Return
i: Index (inferred)
Call *range*
l
e: f32 = 3.0
f = ...
Add
e
d
Add
Div
d
a
b
FuncOp
<FuncOp parameter value 2>
<FuncOp parameter value 1>
Block
ConstOp 5
ConstOp 12.5
ForOp start: 0, stop: ConstOp(5), step: 1
Block
<ForOp parameter value 1>
ConstOp 3.0
AddOp(ConstOp, ConstOp)
f:
e:
i:
l:
d:
b:
a:
'F32': <class 'pydsl.type.F32'>,
'F64': <class 'pydsl.type.F64'>,
'Index': <class 'pydsl.type.Index'>,
'UInt16': <class 'pydsl.type.UInt16'>,
'MemRefFactory': <functools._lru_cache_wrapper object at 0xfffeeb7d28d0>,
'range': <class 'pydsl.scf.range'>,
'compile': <function compile at 0xfffeeb51d120>,
'Memref64': <class 'pydsl.memref.MemrefUnnamedSubclass'>
Variable stack

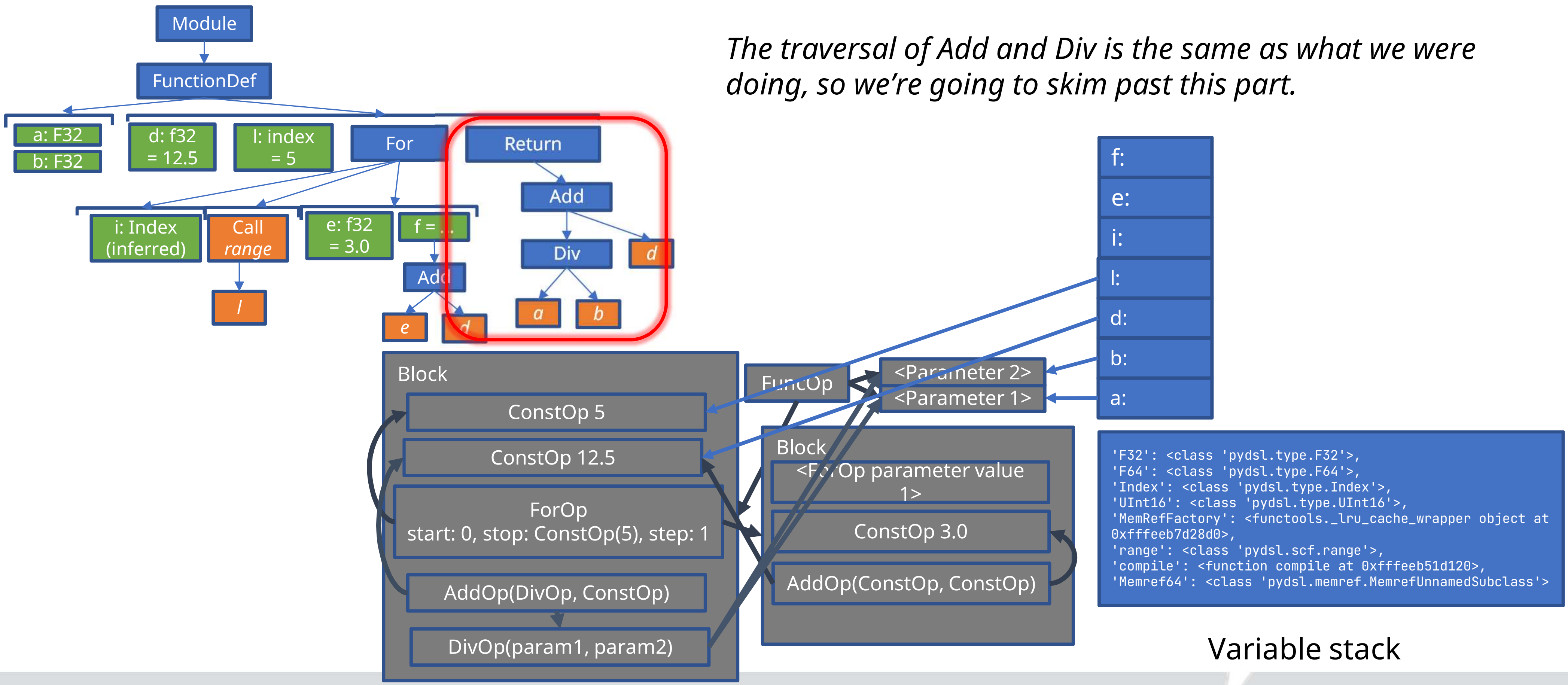
We then traverse to Return



Variable stack

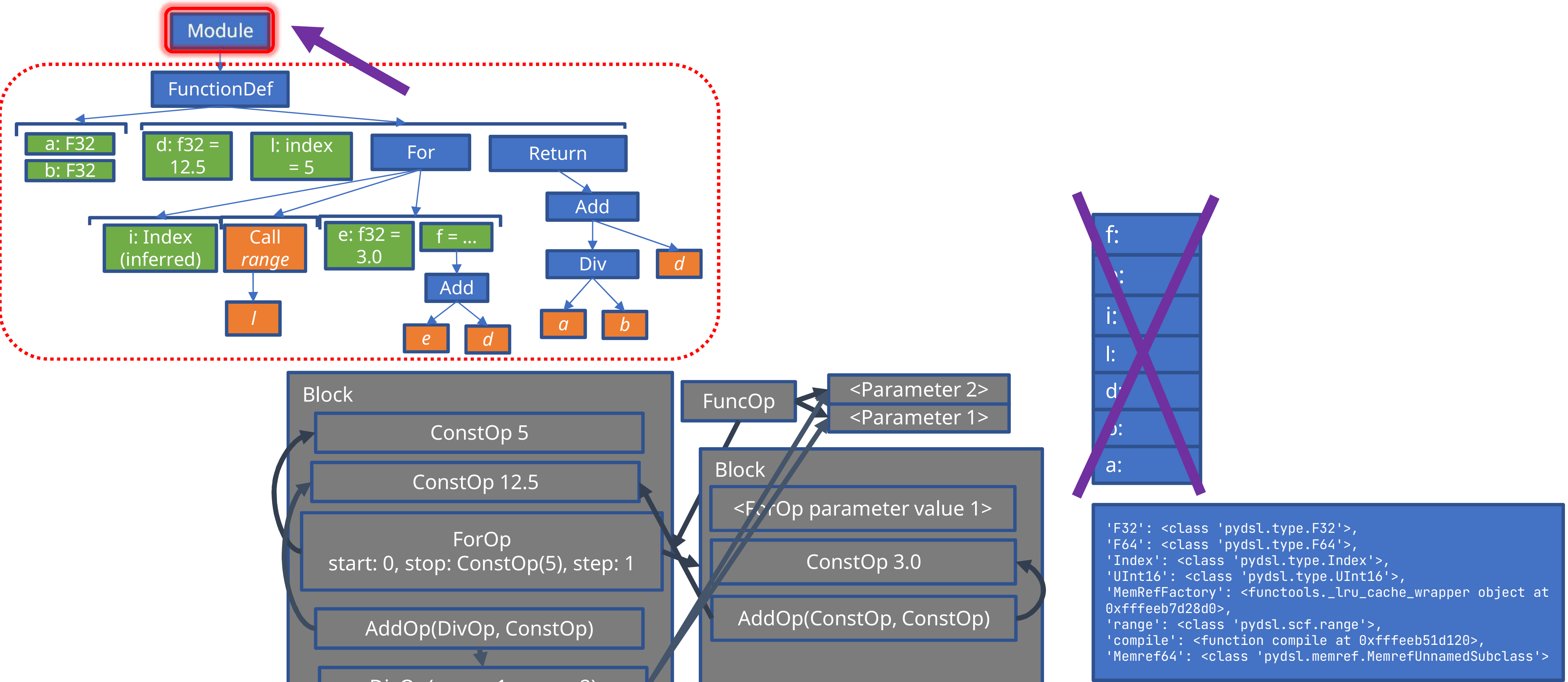
Every FuncOp needs to end on a ReturnOp. For now, if the user does not define a return statement in a function, the compiler would throw an error. We do have a return statement, so we proceed as usual.

The traversal of Add and Div is the same as what we were doing, so we're going to skim past this part.



Variable stack

When we leave a function, we pop the top-most scope from the stack. The visit function is done.



Variable stack

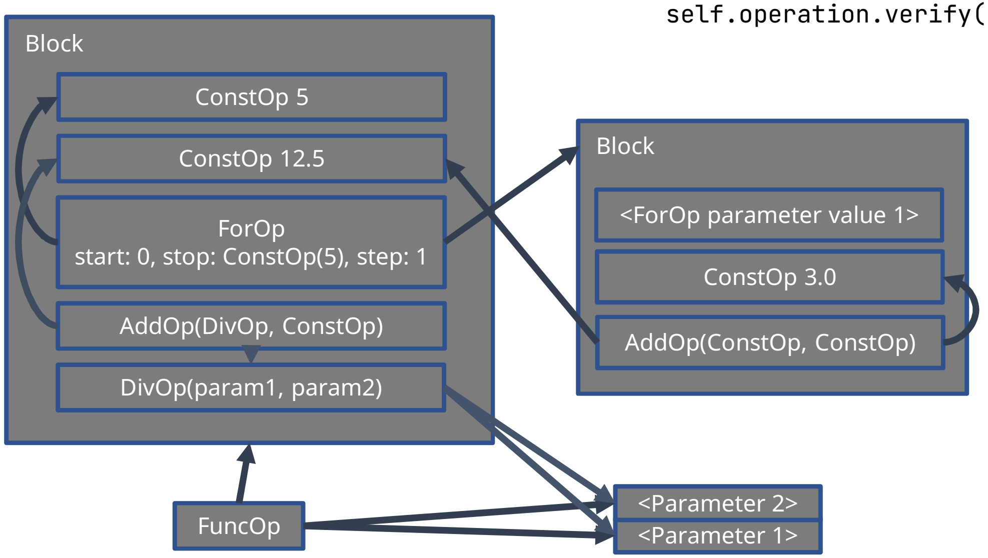
Everything from here on is trivial for us. We ask MLIR to check this graph for consistency, then ask it to dump it as a string. The compilation is done!

MLIR graph

Nothing changes

`self.operation.verify()`

`str(self)`




```
module {
  func.func public @hello(%arg0: f32, %arg1: f32) -> f32 {
    %cst = arith.constant 1.200000e+01 : f32
    %c5 = arith.constant 5 : index
    %c0 = arith.constant 0 : index
    %c1 = arith.constant 1 : index
    scf.for %arg2 = %c0 to %c5 step %c1 {
      %cst_0 = arith.constant 3.000000e+00 : f32
      %2 = arith.addf %cst_0, %cst : f32
    }
    %0 = arith.divf %arg0, %arg1 : f32
    %1 = arith.addf %0, %cst : f32
    return %1 : f32
  }
}
```

This entire graph is actually enclosed in a module block, but we'll omit that.

Affine dialect syntax

- Affine dialect is an important use case for us:
- We define custom iterable macro/"metafunction" class which transforms the for loop that's using it, where `affine_range`'s behavior can be defined.

```
from pydsl.type import UInt32, F64, Index
from pydsl.memref import MemRefFactory
from pydsl.frontend import compile
from pydsl.affine import                    \
    affine_range as arange,                 \
    affine_map as am,                       \
    dimension as D,                         \
    symbol as S
// ...
```

```
def lu(v0: Index, arg1: MemRefF64) -> Index:
    a: UInt32 = 5

    for arg2 in arange(S(v0)):
        for arg3 in arange(D(arg2)):
            for arg4 in arange(D(arg3)):
                arg1[am(D(arg2), D(arg3))] =        \
                    arg1[am(D(arg2), D(arg3))]      \
                    - (arg1[am(D(arg2), D(arg4))]   \
                    * arg1[am(D(arg4), D(arg3))])
```

Transformation dialect syntax

- Presented unique syntax design challenges as `for` (and many other) statements cannot be annotated

```python
def transform_seq(targ: AnyOp):
    fuse_into(
    fuse_into(
    fuse_into(
    fuse_into(
        match(targ, 'fuse_1'),
        match(targ, 'fuse_target1')),
        match(targ, 'fuse_target2')),
        match(targ, 'fuse_target3')),
        match(targ, 'fuse_target4'))

    fuse(match(targ, 'fuse_4'), match(targ, 'fuse_3'), 2)

    tile(match(targ, 'tile'), [32, 32, 32], 6)
```

```python
@compile(locals(), transform_seq=transform_seq, dump_mlir=True,
auto_build=False)
def lu(v0: Index, arg1: MemrefF64) -> UInt32:
    a: UInt32 = 5

    """@tag("tile")"""
    for arg2 in arange(S(v0)):

        """@tag("fuse_4")"""
        for arg3 in arange(D(arg2)):

            """@tag("fuse_1")"""
            for arg4 in arange(D(arg3)):
                arg1[am(D(arg2), D(arg3))] =    \
                    arg1[am(D(arg2), D(arg3))]  \
                    - (arg1[am(D(arg2), D(arg4))]
                    * arg1[am(D(arg4), D(arg3))])
// ...
```

Problem: Python is *really* permissive

- We're not so worried about what Python can't express as much as what it *can*.
- **For example, these are legal code in Python that runs perfectly well. But they do not map well to MLIR.**

```
# b is never defined before this point

if True:
    b = 6

print(b) # This prints 6
```

If we change True to False, this program simply crashes in Python.

```
for i in range(5):
    pass

print(i) # This prints 4
```

...and this is just tip of the iceberg.

To imitate the run-time behavior of Python, we need additional static analysis and use of YieldOp for variables to be usable even after an MLIR scope is closed.

This isn't important to our use case at the moment, so it's not yet worked on.

It's worth noting that...

We're making a minimum viable product (for now)

- Few people are actively working on this project
- Focus of this work is to see whether this approach to compilation can meet our goal:
  - Support nested affine for loop
  - Support affine load and store
  - Keep the compiler simple
  - Ways of specifying specific MLIR features should be idiomatic
  - Like mlir-python-util, support running the compiled program directly from Python (helps with making benchmarks/tests)

E.g. No duck typing: every variable's type is defined at compile time

```python
def weird_function(b: bool):
    a = 5

    if b:
        a = "h"

    # both str and int can *= 5
    a *= 5

    print(a) # 25 or "hhhhh"?
```

We only define types that are necessary for our use case:

- Int
- Index
- Float
- Memref
- AnyOp

Other limitations

These require code analysis pre-pass:

- ❑ Cannot use a variable defined in a for loop outside of the loop
- ❑ Cannot yield a variable to the next iteration in a for loop (so doesn't yet support accumulating variables)
- ❑ Affine symbols and dimensions must be specified verbosely

Other:

- ❑ Ugly compilation error messages, though not hard to fix (AST provides line/column number for each node)
- ❑ Many other important Python features are missing

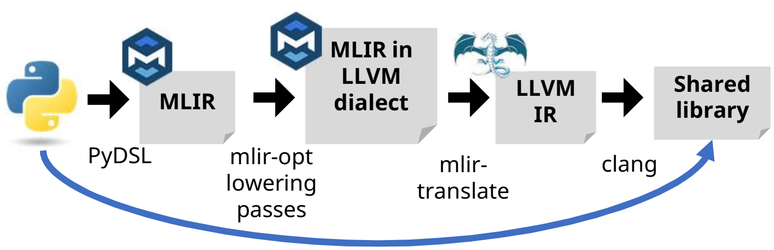
Work-in-progress features

Directly calling compiled MLIR functions from Python

```python
@compile(locals(), dump_mlir=True)
def hello(a: F32, b: F32) -> F32:
    l: Index = 5

    for i in range(l):
        d: F32 = 12.5
        l = i
        p: Index = 3
        c = l + p

    return (a / b)


retval = hello(25, 3) # this now calls the .so library
print(retval)
```

stdout (ignoring warnings)

```
module {
  func.func public @hello(%arg0: f32, %arg1: f32) -> f32
{
    %cst = arith.constant 1.200000e+01 : f32
    %c5 = arith.constant 5 : index
    %c0 = arith.constant 0 : index
    %c1 = arith.constant 1 : index
    scf.for %arg2 = %c0 to %c5 step %c1 {
      %cst_0 = arith.constant 3.000000e+00 : f32
      %2 = arith.addf %cst_0, %cst : f32
    }
    %0 = arith.divf %arg0, %arg1 : f32
    %1 = arith.addf %0, %cst : f32
    return %1 : f32
  }
}

20.333332061767578
```




```python
self._loaded_so = cdll.LoadLibrary(self._so.name)
```

Compile-time abstraction with zero runtime cost: cleaner compiler code and polymorphism

E.g. every numerical type is a Python class that implements operator overloading mapping to the appropriate arith operator.

lower functions converts the class into its raw MLIR representation

```python
@cache
def lower(self) -> OpView:
    return self.value

@classmethod
def lower_class(cls) -> mlir.Type:
    return cls.mlir_type.get()

def _same_type_assertion(self, val):
    return type(self) is type(val)

def __add__(self, rhs: 'Float') -> 'Float':
    self._same_type_assertion(rhs)
    return type(self)(arith.AddFOp(self.value, rhs.value))

def __sub__(self, rhs: 'Float') -> 'Float':
    self._same_type_assertion(rhs)
    return type(self)(arith.SubFOp(self.value, rhs.value))

def __mul__(self, rhs: 'Float') -> 'Float':
    self._same_type_assertion(rhs)
    return type(self)(arith.MulFOp(self.value, rhs.value))

def __truediv__(self, rhs: 'Float') -> 'Float':
    self._same_type_assertion(rhs)
    return type(self)(arith.DivFOp(self.value, rhs.value))
```

Thank you!

Addendum: As of the presentation, the library is not yet open source. We plan to release it to the public shortly.