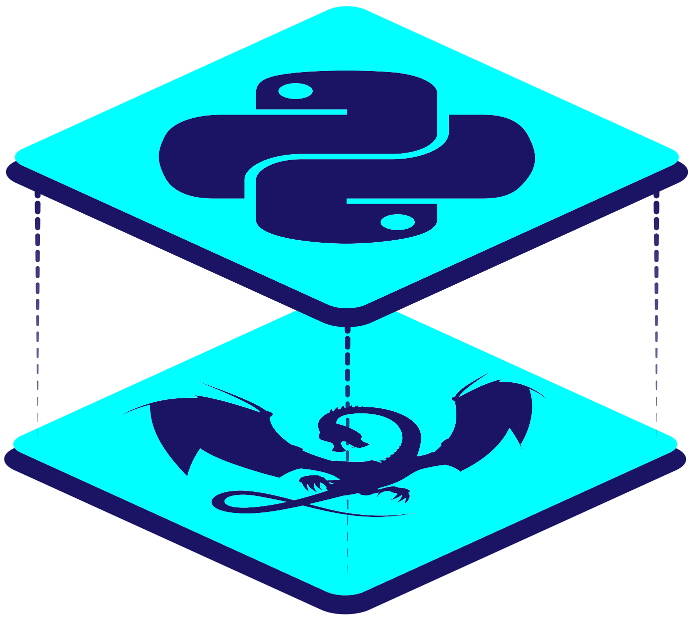
Mapping Python to LLVM

January 09, 2023



Firstly, thank you to everyone who engaged with, tried out, or gave feedback on Codon over the past few weeks! This is the first in a series of blog posts that describe how Codon works internally, various design decisions, performance considerations, and so on.

At a high level, the Codon compiler works in these steps:

1. Parse source code into an abstract syntax tree (AST).

2. Perform type checking on the AST using a modified Hindley-Milner-like algorithm.
3. Convert the typed AST to an intermediate representation called CIR.
4. Perform numerous analyses, transformations and optimizations on the CIR.
5. Convert from CIR to LLVM IR.
6. Run the LLVM optimization pipeline, among other things.

This post will mainly focus on the LLVM IR generation, and how we map various Python constructs to LLVM IR. Sometimes this is very straightforward, while other times it can be very tricky to get right, as we'll see below.

(This post assumes a basic familiarity with LLVM and code generation. See Eli Bendersky's excellent blog posts for an informal introduction or refresher, or this great tutorial.)

Getting started: Python types to LLVM types

One of the first things we need to consider is how to convert from Python types to LLVM IR types. Some of the conversions are quite easy:

- `int` can become an LLVM `i64` (note that this deviates from Python's arbitrary- width integers, but it turns out 64 bits are more than enough for most applications)
- `float` can become an LLVM `double`
- `bool` can become an LLVM `i8` (we could use `i1` in theory, but `i8` is compatible with C/C++)

What about tuples? We can convert tuple types to `struct` s containing the tuple's element types. For example, the type of `(42, 3.14, True)` becomes the LLVM structure type `{i64, double, i8}` . Since tuples are immutable, we can pass these structs by value, and in many cases tuples can be completely optimized out by LLVM's optimization passes.

What about user-defined classes? We can actually do something very similar to what we did for tuples, *except* instead of passing by value, we dynamically allocate and pass by pointer, which allows mutations to be handled correctly. For example, consider:

```
class C:
    a: int
    b: float
    c: bool
```

When we instantiate `C()`, under the hood we'll dynamically allocate memory to store the contents of the same `{i64, double, i8}`, and return a pointer to that memory as the result of instantiation (after calling `C.__init__()`).

There are a few other LLVM types that we expose through Codon, like `Ptr[T]` to represent a pointer to an object of type `T`. Most of the other Python types, though, can be constructed from these building blocks. For example, the built-in collection types like `list`, `dict` and `set` are all implemented within Codon as classes; some other built-in types are implemented as named tuples, and so on.

Operators

Now that we know how to do type conversions, we need to think about how to actually do operations on those types. For example, how do we add two `int` s as in `2 + 2` ? Python implements various operators like `+` , `-` , `*` etc. through its *magic method* interface. For example, `a + b` (typically) becomes `a.__add__(b)` , `a - b` becomes `a.__sub__(b)` and so on.

It turns out we can do the same in Codon, and in fact, all operators are defined by magic methods within Codon's standard library. For example, `list.__add__` might look like:

```
class List:
    ...

    def __add__(self, other):
        result = copy(self)
        result.extend(other)
        return result
```

Magic methods are convenient because they allow *all* operators to be implemented in standard library code, rather than having some be hard-coded in the compiler. But, you may ask, what about operators of primitive types, like `int.__add__`? That's where Codon's inline LLVM IR functionality comes into play: primitive operators are all implemented using inline LLVM IR (via the `@llvm` decorator):

```
class int:
    ...

    @llvm
    def __add__(self, other: int) -> int:
        %tmp = add i64 %self, %other
        ret i64 %tmp

    @llvm
    def __add__(self, other: float) -> float:
        %tmp1 = sitofp i64 %self to double
        %tmp2 = fadd double %tmp1, %other
        ret double %tmp2
```

Note that Codon supports method overloading: the compiler will choose the correct `__add__` based on the right-hand side's type. Now, actually figuring out which magic methods need to be called for a given expression is a complex task in itself and is handled by the type checker, which we'll discuss in detail in a future blog post. For now, we'll assume we have all of this information at our disposal, which is indeed the case by the time we're generating LLVM IR.

Control flow

At this point, we more or less know how to generate code for expressions, so the natural next step is higher-level control flow like `if`, `while`, `for` etc. Unlike (most) expressions, control flow constructs require us to generate multiple *basic blocks*—blocks of straight-line code without any branches, themselves linked together by branches or conditional branches. The basic block is a fundamental concept in LLVM (and compilers in general), and acts as the substrate for LLVM IR.

The compiler itself maintains a pointer to the "current" basic block, B. Different control flow constructs require us to generate a number of basic blocks and link them together in various ways. Let's look at an example:

```
if condition:
    true_branch
else:
    false_branch
```

Compilation would roughly proceed as follows (remember, we always generate code in the current basic block B):

1. Create four new basic blocks: B_{cond}, B_{true}, B_{false} and B_{exit}.

2. Generate a branch to B_{cond}.

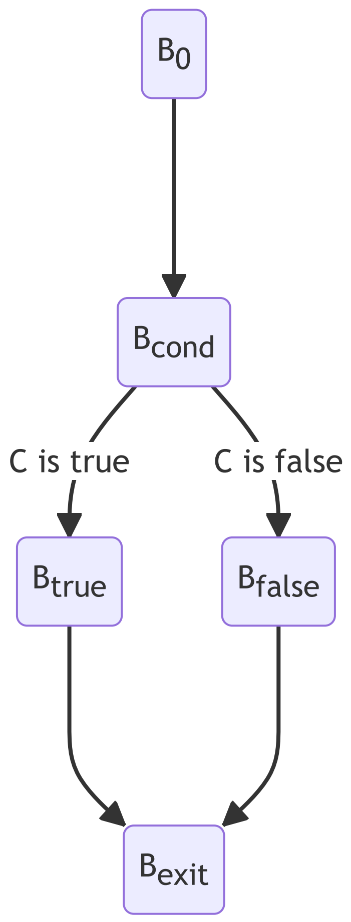
3. Set $B \leftarrow B_{\text{cond}}$ and generate code C for `condition`, then generate a conditional branch to B_{true} and B_{false} based on C.

4. Set $B \leftarrow B_{\text{true}}$ and generate code for `true_branch`, then add a branch to B_{exit}.

5. Set $B \leftarrow B_{\text{false}}$ and generate code for `false_branch`, then add a branch to B_{exit}.

6. Set $B \leftarrow B_{\text{exit}}$.

As a diagram...



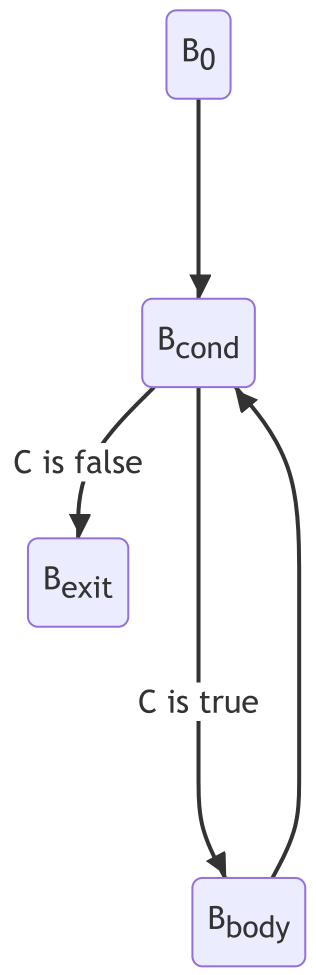
Let's look at one other example:

```
while condition:
    body
```

Here's how compilation would go:

1. Create three new basic blocks: B_{cond}, B_{body} and B_{exit}
2. Generate a branch to B_{cond}.
3. Set $B \leftarrow B_{\text{cond}}$ and generate code C for `condition`, then generate a conditional branch to B_{body} and B_{exit} based on C.
4. Set $B \leftarrow B_{\text{body}}$ and generate code for `body`, then add a branch to B_{cond}.
5. Set $B \leftarrow B_{\text{exit}}$.

Again as a diagram…



What about `break` and `continue` ? These turn out to be very straightforward when thinking in terms of basic blocks: `break` simply becomes a branch to B_{exit} and `continue` becomes a branch to B_{cond}.

Other control flow constructs (like `elif`) work in an analogous way, and in fact can be constructed using just `if` - `else` and `while` . There is one particular statement that requires special care, however, as we'll see next.

`try` - `except` - `finally`

While most of Python's control flow constructs are relatively easy to compile, `try` - `except` - `finally` is quite possibly one of the most difficult and nuanced things to get right when it comes to mapping Python to LLVM IR, on a number of different levels. Exception handling itself is actually not too difficult, as we can use the Itanium C++ ABI for zero-cost exceptions along with LLVM's exception handling instructions: (`landingpad` , `resume` , `invoke`) and appropriate *personality function*. We won't go into too much detail since this is all very similar to what C++ does. Just know that there is some additional bookkeeping required for knowing when we're inside a `try` block, which requires us to call functions with the LLVM `invoke` instruction (rather than the usual `call`) and to specify a basic block to branch to if an exception occurs (i.e. the basic block corresponding to `except`).

The *real* difficulty with `try` - `except` - `finally` comes from the `finally` . It turns out `finally` has certain semantics that many programmers might not even be aware of. For example, take a look at this code:

```
def foo():
    try:
        return 1
    finally:
        return 2
```

What does `foo()` return? If you're not familiar with `finally` semantics, you might be inclined to say `1`, but the correct answer is `2`: `finally` blocks are *always* executed, which makes sense as they will typically be performing some cleanup action that shouldn't be skipped.

This has some important implications for compilation: namely, *branches always need to be aware of enclosing* `try`*-*`finally` *blocks*. Let's take another concrete example; remember when I said `break` and `continue` are straightforward? That turns out to not be entirely true in the presence of `finally`:

```
def bar():
    for i in range(10):
        print(i)
        try:
            if i == 5:
                break
        finally:
            continue
```

What does `bar()` print? Well, when `i == 5`, we'll reach the `break` statement, *but the `break` needs to actually branch to the `finally` block*, otherwise the `finally` block would we skipped over. Now, the `finally` itself has a `continue`, which overrides the previous `break` and resumes the loop. So, in the end, all the integers `0` … `9` are printed.

To actually generate correct code for `try`-`except`-`finally`, we need a couple different ingredients:

- Firstly, we need to maintain a stack of enclosing `try`-`except`-`finally` blocks, since if we reach a `return`, `break` or `continue`, we need to know whether we really need to branch to some `finally` block.
- Next, we need to set up a state machine for each series of nested `try`-`except`-`finally` blocks, since once we *do* reach the `finally`, we need to know

how we got there in order to determine what action we need to take next. For instance, if we got there via a `return`, we need to actually execute that return statement at the end of the block; or perhaps we got there by catching an exception that needs to be delegated to a parent `try`-`except`-`finally`.

At the end of the day, we need to construct a state machine with the following states:

- `NORMAL`: We reached the `finally` through normal execution of the code. Nothing special needs to be done; we can just branch to the next block normally.

- `THROWN`: We've thrown an exception and are executing the `finally` before propagating the exception out of the function. The exception object itself will be stored in a pre-defined place that we can access from the `finally` block.

- `CAUGHT`: We've caught an exception and are reaching the `finally` through some `except` block. Again nothing special needs to be done here; we can just branch to the next block normally.

- `RETURN` : We've reached the `finally` after encountering an enclosed `return` statement. After executing the `finally` , we need to execute the `return` . The return value itself will be stored in a pre-defined place that we can access from the `finally` block.

- `BREAK` : We've reached the `finally` after encountering a `break` . We need to actually execute the `break` after executing the `finally` block.

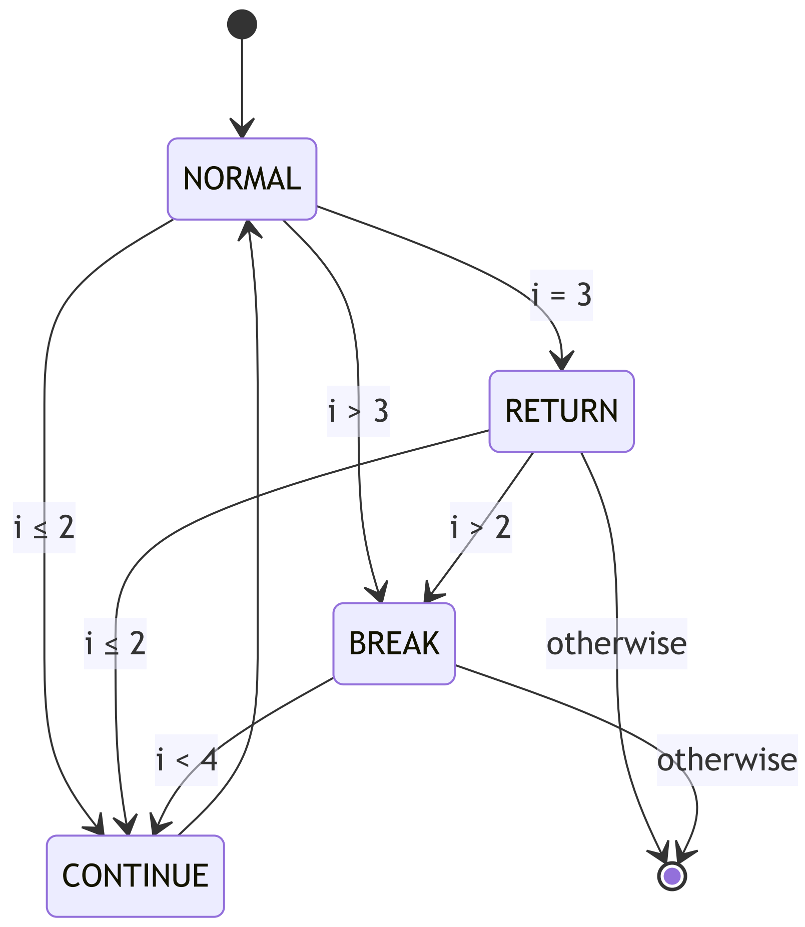
- `CONTINUE` : We've reached the `finally` after encountering a `continue` . We need to actually execute the `continue` after executing the `finally` block.

Importantly, these actions are *recursive*, so when we say *"execute the* `return` *after executing the* `finally` *block"*, that may entail branching to *another* enclosing `finally` block and repeating the same action.

Let's look at a real example of what this state machine looks like. Here's a function:

```python
def baz():
    for i in range(5):
        try:
            try:
                if i == 3:
                    return i
            finally:
                if i > 2:
                    break
        finally:
            if i < 4:
                continue
    return i
```

And here are the various transitions between the states for different conditions throughout the loop:



The internal state machine will transition between the states based on these conditions until the loop terminates, signified by reaching the end state.

Variables and functions

LLVM IR uses *static single assignment form*, or SSA, which effectively means LLVM IR variables must be assigned *exactly* once. As a result, we can't map Python variables directly to LLVM IR variables. Instead, we map each Python variable to a stack-allocated piece of memory:

```
x = 42
```

becomes:

```
%x = alloca i64, align 8
store i64 42, i64* %x, align 8
```

`alloca` is an LLVM IR instruction that allocates space on the current stack frame; `alloca i64` allocates space for a 64-bit integer. Treating variables this way is standard practice when compiling to LLVM IR, and C/C++ compilers will do the same (e.g. `long x = 42` produces this exact code with Clang). Notice also that `alloca` actually returns a *pointer*, so when we dereference a variable (by e.g. using `x` in a later expression), the compiler inserts an implicit dereference of the underlying `alloca` pointer. In C, we can avoid this dereference via the *address-of* operator `&`, as in `&x`; Codon actually supports this via the `__ptr__` built-in: `__ptr__(x)` returns the address of `x`. This is often useful when interfacing with C functions that take a pointer as an argument.

Many variables are implicitly introduced by the parser and/or type checker. For example:

```
a, b = b, a
```

… a common Python idiom for swapping the values of two variables, will implicitly be transformed into

```
tmp = (b, a)
a = tmp[0]
b = tmp[1]
```

thus introducing the new variable `tmp`. We'll discuss these types of front-end transformations in much more detail in a subsequent blog post.

As for functions: we can actually map Python functions directly to LLVM functions:

```
def foo(a: int, b: int):
    return a + b
```

becomes:

```
define i64 @foo(i64 %0, i64 %1) {
  %a = alloca i64, align 8
  %b = alloca i64, align 8
  store i64 %0, i64* %a, align 8
  store i64 %1, i64* %b, align 8
  %a0 = load i64, i64* %a, align 8
  %b0 = load i64, i64* %b, align 8
  %ans = add nsw i64 %a0, %b0
  ret i64 %ans
}
```

Notice that we allocate space for the arguments via `alloca`, since they should be treated like normal variables inside the function. The only complication to this otherwise fairly straightforward conversion is generators, which we'll talk about next.

Generators

Generators are one of the hallmarks of the Python language, and are used extensively in the standard library. For example, many of the built-in functions operate on or return generators:

```
map(lambda x: x + 1, [1, 2, 3])  # generator giving 2
reversed('abc')                  # generator giving '
zip('abc', [1, 2, 3])            # generator giving (
```

Generators can be defined by using `yield`:

```
def squares(n):
    for i in range(1, n + 1):
        yield i * i
```

`list(squares(5))` gives all the square numbers up to 5^2: `[1, 4, 9, 16, 25]`.

Since generators are so fundamental in Python, we need a way to handle them efficiently in LLVM. Fortunately, LLVM supports coroutines, which we can use as the foundation for implementing generators. (Note that C++20 also adds support for coroutines.)

You can learn about all the intricacies of LLVM coroutines in the docs. Briefly, though, coroutines are like functions that allow suspension and resumption (much like what happens with Python's `yield`). Coroutines maintain their state (i.e. local variables, position in the function, yielded value) in what's called a *coroutine frame*. Coroutines in LLVM are indeed also like normal functions, but delineate their resume/suspend points with special intrinsics and "return" a handle to their coroutine frames. Here are some of the more important LLVM intrinsics we'll use when generating code for coroutines:

- `@llvm.coro.id` : Gives us a token that can identify the coroutine, which we'll pass to many of the other intrinsics.
- `@llvm.coro.size.i64` : Tells us the size of the coroutine frame for dynamic allocation.
- `@llvm.coro.begin` : Gives us a "handle" to the coroutine frame.
- `@llvm.coro.suspend` : Marks a suspension point.
- `@llvm.coro.end` : Marks the end of the coroutine and destroys the coroutine frame.

These intrinsics are all used *within* the coroutine function itself, but how do we actually call and manipulate the coroutine externally? Well, there are intrinsics for that as well:

- `@llvm.coro.resume` : Resumes a coroutine given a coroutine handle.
- `@llvm.coro.done` : Checks if a coroutine is at its final suspend point.
- `@llvm.coro.promise` : Returns a pointer to the *coroutine promise*: a region of memory that stores values “yielded” from the coroutine.
- `@llvm.coro.destroy` : Destroys a finished coroutine.

With these primitives in hand, we can implement all we need to support generators:

- Functions with `yield` will be converted to LLVM coroutines.
- We can generate code for `yield` statements by storing the yielded value in the coroutine promise, then calling `@llvm.coro.suspend` .
- We can implement Python’s `next()` built-in by calling `@llvm.coro.done` to see if the given generator is finished, then calling `@llvm.coro.resume` to resume it and finally `@llvm.coro.promise` to obtain the generated value.

- We can implement `for x in generator` by repeatedly calling `@llvm.coro.resume` / `@llvm.coro.promise` until `@llvm.coro.done` tells us to stop.

Let's look at an example:

```
for i in range(3):
    print(i)
```

Here's the (simplified) LLVM IR we'll generate for this snippet:

```
entry:
  %g = call i8* @range(i64 3)
  br label %for

for:
  call void @llvm.coro.resume(i8* %g)
  %done = call i1 @llvm.coro.done(i8* %g)
  br i1 %done, label %exit, label %body

body:
  %p1 = call i8* @llvm.coro.promise(i8* %g, i32 8, i1
  %p2 = bitcast i8* %p1 to i64*
  %i = load i64, i64* %p2
```

```
    call void @print(i32 %i)
    br label %for

  exit:
    call void @llvm.coro.destroy(i8* %g)
```

Just to recap...

- The call to `@range` gives us a *handle* (with type `i8*`) to the range generator.
- We resume the generator with `@llvm.coro.resume` (note that all coroutines will be initially suspended to match Python's generator semantics).
- If `@llvm.coro.done` tells us the generator is done, we exit the loop and call `@llvm.coro.destroy` to destroy it.
- Otherwise, in the body of the loop, we obtain the next value for `i` by calling `@llvm.coro.promise` (the other arguments are simply the alignment of the promise (`i32 8`) and whether we want to obtain a promise given a handle or vice versa (`i1 false`)).

At this point you might be asking yourself: isn't that a lot of code to just loop over `0, 1, 2` and print each value?

That's where LLVM's coroutine optimization passes come in: it turns out coroutines are particularly amenable to compiler optimizations that can typically eliminate *all* of the coroutine overhead entirely. Specifically, by far the most common case when dealing with generators in Python is for them to be created and destroyed in the same function, in which case the coroutine passes can inline them, elide heap allocation for the frame and so on. Here's the result of running the coroutine passes on our example above:

```
call void @print(i64 0)
call void @print(i64 1)
call void @print(i64 2)
```

We were able to get rid of everything coroutine-related! By default, LLVM's coroutine passes rely on an analysis of `@llvm.coro.destroy` calls. Codon's LLVM fork instead relies on an escape analysis of the coroutine handle, which we found to be much better at determining which coroutines can be optimized.

Program structure

We have most of the key ingredients at this point, but what about the overarching program structure? Python doesn't have an explicit `main()` function as an entry point like C does. LLVM, on the other hand, does, so we need to reconcile that difference.

The way we handle this in Codon is by putting everything at the top level into its own implicit function:

```
a = 42
print(a * 2)
```

becomes:

```
a = 0
def main():
    global a
    a = 42
    print(a * 2)
```

This also allows us to do some setup and initialization as well, such as initializing the runtime/GC, setting up `sys.argv` , etc.

This approach usually works fairly seamlessly, but there is one subtle case that needs special care: exporting Codon functions in a shared library. For example, consider the following:

```
def bar() -> int:
    ... # some complex operation

M = bar()

def foo(n: int) -> int:
    return n + M
```

and assume we want to generate a shared library `libfoo.so` from Codon that exposes the `foo` function. If we do so naively, the global variable `M` will actually not be initialized: we need to run the implicit `main()` we created in order to initialize `M`. We solve this by running `main()` as a global constructor if we're compiling to a shared object.

Optimizations

Now is a good time to talk about optimizations. LLVM has an extensive repertoire of IR passes for doing various things. These include many of the standard compiler optimizations like constant folding, dead code elimination, loop invariant code motion, etc. These passes are grouped into standard pipelines, like the " `O0` " pipeline or the " `O3` " pipeline, which correspond to the `-O0` and `-O3` flags on a C/C++ compiler like Clang.

Codon uses these default optimization pipelines, but also adds a few of its own LLVM passes. One example is `AllocationRemover` , which 1) removes unused heap (GC) allocations and 2) demotes small heap allocations to the stack (i.e. to an `alloca` instruction). This turns out to be useful when instantiating classes; for example creating an instance of a class `C` that never escapes its enclosing function can actually be done purely on the stack:

```
def foo():
    c = C('hello')  # codegen'd as dynamic allocatior
    ...  # C never escapes foo
```

The same goes for the built-in collection types like lists or dictionaries:

```
a = [1, 2, 3]
print(a[0] + a[1] + a[2])  # optimized to 6; no alloc
```

Codon uses a couple other custom LLVM passes, but the vast majority of its custom optimizations are performed in its own IR, which we'll discuss in a future post.

Examples

To conclude this post, I want to showcase a few neat examples that illustrate many of these pieces coming together, combined with LLVM's optimization passes. I've simplified the naming in the LLVM IR outputs for clarity.

User-defined generator with loop

```
def squares(n):
    for i in range(1, n + 1):
        yield i**2
```

```
x = 0
for s in squares(1000):
    x += s

print(x)
```

This gets optimized away entirely, and the final answer `333833500` is printed directly:

```
%answer = call { i64, i8* } @int_to_str(i64 33383350
```

Tail recursion elimination

```
from sys import argv
N = int(argv[1])

def collatz(n):
    if n == 1:
        return 0
    else:
        return 1 + collatz(3*n + 1 if n % 2 else n //

print(collatz(N))
```

The recursion is eliminated and replaced with a loop:

```
collatz_entry:
  %tmp1 = icmp eq i64 %N, 1
  br i1 %tmp1, label %collatz_exit, label %if.false

if.false:
  %tmp2 = phi i64 [ %tmp7, %ternary.exit ], [ %N, %cc
  %accumulator = phi i64 [ %tmp8, %ternary.exit ], [
  %tmp3 = and i64 %tmp2, 1
  %not1 = icmp eq i64 %tmp3, 0
  br i1 %not1, label %ternary.false, label %ternary.t

ternary.true:
  %tmp4 = mul i64 %tmp2, 3
  %tmp5 = add i64 %tmp4, 1
  br label %ternary.exit

ternary.false:
  %tmp6 = sdiv i64 %tmp2, 2
  br label %ternary.exit

ternary.exit:
  %tmp7 = phi i64 [ %tmp5, %ternary.true ], [ %tmp6,
  %tmp8 = add i64 %accumulator, 1
  %not2 = icmp eq i64 %tmp7, 1
  br i1 %not2, label %collatz_exit, label %if.false
```

```
collatz_exit:
  %answer = phi i64 [ 0, %collatz_entry ], [ %tmp8, %
```

List of tuples

```
from sys import argv
from random import random
from math import hypot

# Random points in quadrant
points = [(random(), random()) for _ in range(10_000_

# How many points are inside/outside of quarter-circl
inside, outside = 0, 0
for x, y in points:
    if hypot(x, y) < 1:
        inside += 1
    else:
        outside += 1

# Estimate pi using the ratio
pi = 4 * inside / (inside + outside)
print(pi)
```

Tuples are eliminated, and the `inside` and `outside` variables are converted to LLVM IR SSA variables:

```
for.body:
  %idx = phi i64 [ %idx_add, %for.body ], [ 0, %for.c
  %inside = phi i64 [ %inside_add, %for.body ], [ 0,
  %outside = phi i64 [ %outside_add, %for.body ], [ 0
  %p1 = getelementptr { double, double }, ptr %points
  %x = load double, ptr %p1, align 8
  %p2 = getelementptr { double, double }, ptr %points
  %y = load double, ptr %p2, align 8
  %h = tail call double @hypot(double %x, double %y)
  %tmp1 = fcmp uge double %h, 1.000000e+00
  %tmp2 = zext i1 %tmp1 to i64
  %outside_add = add i64 %outside, %tmp2
  %tmp3 = xor i1 %tmp1, true
  %tmp4 = zext i1 %tmp3 to i64
  %inside_add = add i64 %inside, %tmp4
  %idx_add = add nuw nsw i64 %idx, 1
  %exitcond = icmp eq i64 %idx, %N
  br i1 %exitcond, label %for.exit, label %for.body
```

Conclusion

I hope this post helped shed some light on the various methods we can employ to compile Python code to LLVM. You can view the LLVM IR output of your own code with `codon build -llvm` (remember to add the `-release` flag to enable optimizations).

There *are* many things we took for granted here, like how we determine the data types to begin with, or how we put the source code in a format that's suitable for code generation. These, among other things, will be topics of future posts in this series. Stay tuned!