llvm_cbuilder: A Quick Tour

Writing code generation logic in pure llvmpy requires one to think like a compiler -- dealing with basic-blocks, branches, SSA, etc. The resulting code looks like assembly code, with no apparent hint about the control-flow.

llvm_cbuilder is originally designed to simplify the translation of C code into llvmpy. It provides a simple API for mimicking C programming in Python. With llvm_cbulder, one can write very low-level code, probably even lower-level than C. At the same time, one can use if-else, loop, structures and a bit of OOP.

A Simple Example

Let's see some action. We will define a function that calculates the square of a double-precision float.

In [1]:
```
from llvm.core import *
from llvm_cbuilder import *
import llvm_cbuilder.shortnames as C

class Square(CDefinition):
    # prototype: double square(double x)
    _name_ = 'square'                # function name
    _retty_ = C.double
    _argtys_ = [ ('x', C.double) ]

    def body(self, x):
        y = x * x                    # just write out the expression
        self.ret(y)
```

Notice how numerical expressions can be written naturally.

Let's see the emitted code:

In [2]:
```
m = Module.new('my_module')
llvm_square = Square()(m)        # define square() in my_module
print(m)
```

```
; ModuleID = 'my_module'

define double @square(double %x) {
decl:
  %x1 = alloca double
  br label %body

body:                                             ; preds = %decl
  store double %x, double* %x1
  %0 = load double* %x1
  %1 = load double* %x1
  %2 = fmul double %0, %1
  ret double %2
}
```

Let's generate a ctype function object to call `square()` in the Python code:

In [3]:
```
exe = CExecutor(m)
square = exe.get_ctype_function(llvm_square, "double, double")
result = square(1.2)
```

```
print(result)
```

```
1.44
```

Control Flow Constructs

The main strength of llvm_cbuilder is the control-flow constructs. They use the python "with" statement to setup new code blocks to contain different paths of the control flow.

The following example demonstrates both the if-else and loop contructs:

In [4]:
```
class IsPrime(CDefinition):
    # prototype int isprime(int x)
    _name_ = 'isprime'
    _retty_ = C.int
    _argtys_ = [ ('x', C.int) ]

    def body(self, x):
        two = self.constant(C.int, 2)
        true = one = self.constant(C.int, 1)
        false = zero = self.constant(C.int, 0)

        with self.ifelse( x <= two ) as ifelse:
            with ifelse.then():
                self.ret(true)

        with self.ifelse( (x % two) == zero ) as ifelse:
            with ifelse.then():
                self.ret(false)

        idx = self.var(C.int, 3, name='idx')
        with self.loop() as loop:
            with loop.condition() as setcond:
                setcond( idx < x )

            with loop.body():
                with self.ifelse( (x % idx) == zero ) as ifelse:
                    with ifelse.then():
                        self.ret(false)
                idx += two
        self.ret(true)
```

The code above is quite verbose. It is like writing in Pascal or Ada. But, it is still easier than writing in llvmpy directly. Take a look at the generated LLVM IR below.

In [5]:
```
llvm_isprime = IsPrime()(m)
print(llvm_isprime)
```

```
define i32 @isprime(i32 %x) {
decl:
  %x1 = alloca i32
  %idx = alloca i32
  br label %body

body:                                             ; preds = %decl
  store i32 %x, i32* %x1
  %0 = load i32* %x1
  %1 = icmp sle i32 %0, 2
```

```
  br i1 %1, label %if.then, label %if.else

if.then:                                          ; preds = %body
  ret i32 1

if.else:                                          ; preds = %body
  br label %if.end

if.end:                                           ; preds = %if.else
  %2 = load i32* %x1
  %3 = srem i32 %2, 2
  %4 = icmp eq i32 %3, 0
  br i1 %4, label %if.then2, label %if.else3

if.then2:                                         ; preds = %if.end
  ret i32 0

if.else3:                                         ; preds = %if.end
  br label %if.end4

if.end4:                                          ; preds = %if.else3
  store i32 3, i32* %idx
  br label %loop.cond

loop.cond:                                        ; preds = %if.end7, %if.end4
  %5 = load i32* %idx
  %6 = load i32* %x1
  %7 = icmp slt i32 %5, %6
  br i1 %7, label %loop.body, label %loop.end

loop.body:                                        ; preds = %loop.cond
  %8 = load i32* %x1
  %9 = load i32* %idx
  %10 = srem i32 %8, %9
  %11 = icmp eq i32 %10, 0
  br i1 %11, label %if.then5, label %if.else6

loop.end:                                         ; preds = %loop.cond
  ret i32 1

if.then5:                                         ; preds = %loop.body
  ret i32 0

if.else6:                                         ; preds = %loop.body
  br label %if.end7

if.end7:                                          ; preds = %if.else6
  %12 = load i32* %idx
  %13 = add i32 %12, 2
  store i32 %13, i32* %idx
  br label %loop.cond
}
```

We'll setup a ctype function object to try it out:

In [6]:
```
isprime = exe.get_ctype_function(llvm_isprime, 'int, int')
prime_100 = filter(isprime, range(2, 100))
print(prime_100)
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79,
83, 89, 97]
```

Structures

It is possible to create structures in llvm_cbuilder. Beware that structures in LLVM are unlike those in C. LLVM type system allows structures to be literal or identified. Literal structures are equivalent iff they have the same elements. Identified structures are equivalent iff they have the same name. In llvm_cbuilder, all structures are, by default, literal types.

Here's an example:

In [7]:
```
class Vector2D(CStruct):
    _fields_ = [
        ('x', C.float),
        ('y', C.float),
    ]
```

That's all you need for a structure. Very much like defining structures with ctypes.

We can also bind methods to structures which inline code to the caller.

In [8]:
```
class Vector2D(CStruct):
    _fields_ = [
        ('x', C.float),
        ('y', C.float),
    ]

    def add(self, other):
        self.x += other.x
        self.y += other.y
```

Setup a a test function:

In [9]:
```
class TestVector(CDefinition):
    _name_ = 'testvector'
    _retty_ = C.float
    _argtys_ = [('x', C.float),
                ('y', C.float)]

    def body(self, x, y):
        va = self.var(Vector2D)
        vb = self.var(Vector2D)
        va.x.assign(self.constant(C.float, 10))
        va.y.assign(self.constant(C.float, 20))
        vb.x.assign(x)
        vb.y.assign(y)

        va.add(vb)

        return self.ret(va.x * va.y)
```

In [10]:
```
llvm_testvec = TestVector()(m)
testvec = exe.get_ctype_function(llvm_testvec, "float, float, float")
print(testvec(-8, 5))  # should output 50
```

```
50.0
```

Generic Programming

llvm_cbuilder supports generic function (or template function for C++). When a `CDefinition` defines the `specialize` class method, its constructor will invoke `specialize`. All parameters passed to the constructor are forwarded to `specialize`.

The constructor of a generic CDefinition creates a new dynamic class with the original CDefinition subclass as the parent. Thus, `specialize` can modify class attributes without affecting the parent.

In [11]:

```
class GenericSquare(CDefinition):
    @classmethod
    def specialize(cls, data_type):
        cls._name_ = '.'.join(['square', str(data_type)])   # set function name
        cls._retty_ = data_type                             # set return type
        cls._argtys_ = [ ('x', data_type) ]                 # set argument type

    def body(self, x):
        y = x * x                    # just write out the expression
        self.ret(y)

llvm_square_int = GenericSquare(data_type=C.int)(m)
llvm_square_float = GenericSquare(data_type=C.float)(m)

print(llvm_square_int)
print(llvm_square_float)
```

```
define i32 @square.i32(i32 %x) {
decl:
  %x1 = alloca i32
  br label %body

body:                                             ; preds = %decl
  store i32 %x, i32* %x1
  %0 = load i32* %x1
  %1 = load i32* %x1
  %2 = mul i32 %0, %1
  ret i32 %2
}


define float @square.float(float %x) {
decl:
  %x1 = alloca float
  br label %body

body:                                             ; preds = %decl
  store float %x, float* %x1
  %0 = load float* %x1
  %1 = load float* %x1
  %2 = fmul float %0, %1
  ret float %2
}
```

External Functions

`CExternal` is a convenient class to help accessing externally-defined functions.

We define an external interface to `sqrtf()` in libm.

In [12]:
```
class LibM(CExternal):
    sqrtf = Type.function(C.float, [C.float])
```

All class attributes that are `llvm.core.FunctionType` in `CExternal` are converted to a `CFunc` instance. `CFunc` instances are callable. `CFunc.__call__` generates code that perform the corresponding LLVM function call.

The following example demonstrates the usage of `CExternal` and `CFunc`.

In [13]:
```
class TestSqrtf(CDefinition):
    _name_ = 'test_sqrtf'
    _retty_ = C.float
    _argtys_ = [('x', C.float)]

    def body(self, x):
        libm = LibM(self)   # init the API
        y = libm.sqrtf(x)   # call sqrtf
        self.ret(y)

llvm_sqrtf = TestSqrtf()(m)
sqrtf = exe.get_ctype_function(llvm_sqrtf, 'float, float')
print(sqrtf(144))
```

```
12.0
```

Casting

Unlike C, llvm_cbuilder does not automatically cast variables. It is up to the user to explicit cast types. All binary operations require both operands to be the same type. All values in llvm_cbuilder has a `cast(destty)` method.

For example:

In [14]:
```
class CastExample(CDefinition):
    _name_ = 'cast_example'
    _retty_ = C.float
    _argtys_ = [('x', C.int)]

    def body(self, x):
        y = x.cast(C.float)   # cast integer x to float
        self.ret(y)
```

More... (TODO)