llvmlite Documentation

Release 0.5.0

Continuum Analytics

January 06, 2017

Contents

CHAPTER 1

Introduction

1.1 Overview

llvmlite is born of the desire to have a new Python binding to LLVM for use in Numba. Numba used to rely on llvmpy, but llvmpy became harder and harder to maintain, because of its cumbersome architecture and also because the C++11 requirement of recent LLVM versions don't go well with the compiler and runtime ABI requirements of some Python versions (especially under Windows).

Moreover, llvmpy proved to be responsible for a sizable chunk of Numba's compilation times, because of its inefficient layering and object encapsulation. Fixing this issue inside the llvmpy codebase seemed a time-consuming and uncertain task.

Therefore, the Numba developers decided to start a new binding from scratch, with an entire different architecture, centered around the specific requirements of a JIT compiler.

1.2 Philosophy

While llvmpy exposed large parts of the LLVM C++ API for direct calls into the LLVM library, llvmlite takes an entirely different approach. llvmlite starts from the needs of a JIT compiler and splits them into two decoupled tasks:

1. Construction of a *module*, function by function, *instruction* by instruction
2. Compilation and optimization of the module into machine code

The construction of a LLVM module doesn't call the LLVM C++ API; rather, it constructs the LLVM *Intermediate Representation* in pure Python. This is the part of the *IR layer*.

The compilation of a LLVM module takes the IR in textual form and feeds it into LLVM's parsing API. It then returns a thin wrapper around LLVM's C++ module object. This is the part of the *binding layer*.

Once parsed, the module's source code cannot be modified anymore; this is where llvmlite loses in flexibility compared to the direct mapping of C++ APIs into Python that was provided by llvmpy. The saving in maintenance effort, however, is large.

1.3 LLVM compatibility

Despite minimizing the API surface with LLVM, llvmlite is impacted by changes to LLVM's C++ API (which can occur at every feature release). Therefore, each llvmlite version is targetted to a specific LLVM feature version. It should work accross all given bugfix releases of that version (for example, llvmlite 0.6.0 would work with LLVM 3.6.0 and 3.6.1, but with neither LLVM 3.5.0 nor 3.7.0).

Which LLVM version is supported is driven by Numba's requirements.

1.4 API stability

At this time, we reserve ourselves the possibility to slightly break the llvmlite API at each release. This is necessary because of changes in LLVM behaviour (for example differences in the IR accross versions), and because llvmlite as a young library still has room for improvement or fixes to the existing APIs.

CHAPTER 2

Installing

2.1 Pre-built binaries

For your own convenience, we recommend you install the pre-built binaries provided for the Conda package manager. Official binaries are available in the Anaconda distribution; to install them, simply type:

```
$ conda install llvmlite
```

If you want a more recent version, you can also fetch the automatic builds available on Numba's binstar channel:

```
$ conda install --channel numba llvmlite
```

If don't want to use Conda, or modified llvmlite yourself, you will need to build it.

2.2 Building manually

2.2.1 Prerequisites (UNIX)

You must have a LLVM 3.6 build (libraries and header files) available somewhere.

Under a recent Ubuntu or Debian system, you may install the `llvm-3.6-dev` package if available.

If building LLVM on Ubuntu, the linker may report an error if the development version of `libedit` is not installed. Install `libedit-dev` if you run into this problem.

2.2.2 Prerequisites (Windows)

You must have Visual Studio 2012 or later (the free "Express" edition is ok) in order to compile LLVM and llvmlite. In addition, you must have cmake installed, and LLVM should have been built using cmake, in Release mode. Be careful to use the right bitness (32- or 64-bit) for your Python installation.

2.2.3 Compiling

Run `python setup.py build`. This builds the llvmlite C wrapper, which embeds a statically-linked copy of the required subset of LLVM.

If your LLVM is installed in a non-standard location, first point the `LLVM_CONFIG` environment variable to the path of the corresponding `llvm-config` (or `llvm-config.exe`) executable.

2.2.4 Installing

Validate your build by running the test suite: run `python runtests.py` or `python -m llvmlite.tests`. If everything works fine, install using `python setup.py install`.

CHAPTER 3

llvmlite.ir – The IR layer

The *llvmlite.ir* module contains classes and utilities to build the LLVM *Intermediate Representation* of native functions. The provided APIs may sometimes look like LLVM's C++ APIs, but they never call into LLVM (unless otherwise noted): they construct a pure Python representation of the *IR*.

See also:

To make use of this module, you should be familiar with the concepts presented in the LLVM Language Reference.

3.1 Types

All *values* used in a LLVM module are explicitly typed. All types derive from a common base class *Type*. Most of them can be instantiated directly. Once instantiated, a type should be considered immutable.

class `llvmlite.ir.`**`Type`**

The base class for all types. You should never instantiate it directly. Types have the following methods in common:

`as_pointer`(*addrspace=0*)

Return a *PointerType* pointing to this type. The optional *addrspace* integer allows you to choose a non-default address space (the meaning is platform-dependent).

`get_abi_size`(*target_data*)

Get the ABI size of this type, in bytes, according to the *target_data* (a *llvmlite.binding.TargetData* instance).

`get_abi_alignment`(*target_data*)

Get the ABI alignment of this type, in bytes, according to the *target_data* (a *llvmlite.binding.TargetData* instance).

Note: *get_abi_size()* and *get_abi_alignment()* call into the LLVM C++ API to get the requested information.

3.1.1 Atomic types

class `llvmlite.ir.`**`PointerType`**(*pointee, addrspace=0*)

The type of pointers to another type. *pointee* is the type pointed to. The optional *addrspace* integer allows you to choose a non-default address space (the meaning is platform-dependent).

Pointer types exposes the following attributes:

addrspace
The pointer's address space number.

pointee
The type pointed to.

class `llvmlite.ir.`**`IntType`**(*bits*)
The type of integers. *bits*, a Python integer, specifies the bitwidth of the integers having this type.

class `llvmlite.ir.`**`FloatType`**
The type of single-precision floating-point real numbers.

class `llvmlite.ir.`**`DoubleType`**
The type of double-precision floating-point real numbers.

class `llvmlite.ir.`**`VoidType`**
The class for void types; only used as the return type of a function without a return value.

3.1.2 Aggregate types

class `llvmlite.ir.`**`Aggregate`**
The base class for aggregate types. You should never instantiate it directly. Aggregate types have the following attribute in common:

elements
A tuple-like immutable sequence of element types for this aggregate type.

class `llvmlite.ir.`**`ArrayType`**(*element*, *count*)
The class for array types. *element* is the type of every element, *count* the number of elements (a Python integer).

class `llvmlite.ir.`**`LiteralStructType`**(*elements*)
The class for literal struct types. *elements* is a sequence of element types for each member of the structure.

3.1.3 Other types

class `llvmlite.ir.`**`FunctionType`**(*return_type*, *args*, *var_arg=False*)
The type of a function. *return_type* is the return type of the function. *args* is a sequence describing the types of argument to the function. If *var_arg* is true, the function takes a variable number of additional arguments (of unknown types) after the explicit *args*.

Example:

```
int32 = ir.IntType(32)
fnty = ir.FunctionType(int32, (ir.DoubleType(), ir.PointerType(int32)))
```

An equivalent C declaration would be:

```
typedef int32_t (*fnty)(double, int32_t *);
```

class `llvmlite.ir.`**`LabelType`**
The type for *labels*. You don't need to instantiate this type.

class `llvmlite.ir.`**`MetaData`**
The type for *metadata*. You don't need to instantiate this type.

3.2 Values

Values are what a *module* mostly consists of.

`llvmlite.ir.`**`Undefined`**
An undefined value (mapping to LLVM's "undef").

class `llvmlite.ir.`**`Constant`**(*typ*, *constant*)
A literal value. *typ* is the type of the represented value (a `Type` instance). *constant* is the Python value to be represented. Which Python types are allowed for *constant* depends on *typ*:

- All types accept `Undefined`, and turn it into LLVM's "undef".
- All types accept None, and turn it into LLVM's "zeroinitializer".
- `IntType` accepts any Python integer or boolean.
- `FloatType` and `DoubleType` accept any Python real number.
- Aggregate types (i.e. array and structure types) accept a sequence of Python values corresponding to the type's element types.
- In addition, `ArrayType` also accepts a single `bytearray` instance to initialize the array from a string of bytes. This is useful for character constants.

classmethod **`literal_struct`**(*elements*)
An alternate constructor for constant structs. *elements* is a sequence of values (`Constant` or otherwise). A constant struct containing the *elems* in order is returned

Note: You cannot define constant functions. Use a *function declaration* instead.

class `llvmlite.ir.`**`Value`**
The base class for non-literal values.

class `llvmlite.ir.`**`MetaDataString`**(*module*, *value*)
A string literal for use in *metadata*. *module* is the module the metadata belongs to. *value* is a Python string.

class `llvmlite.ir.`**`MDValue`**
A *metadata* node. To create an instance, call `Module.add_metadata()`.

class `llvmlite.ir.`**`NamedMetaData`**
A named metadata. To create an instance, call `Module.add_named_metadata()`. Named metadata has the following method:

`add`(*md*)
Append the given piece of metadata to the collection of operands referred to by the NamedMetaData. *md* can be either a `MetaDataString` or a `MDValue`.

class `llvmlite.ir.`**`Argument`**
One of a function's arguments. Arguments have the following method:

`add_attribute`(*attr*)
Add an argument attribute to this argument. *attr* is a Python string.

class `llvmlite.ir.`**`Block`**
A *basic block*. You shouldn't instantiate or mutate this type directly; instead, call the helper methods on `Function` and `IRBuilder`.

Basic blocks have the following methods and attributes:

replace (*old*, *new*)
: Replace the instruction *old* with the other instruction *new* in this block's list of instructions. All uses of *old* in the whole function are also patched. *old* and *new* are `Instruction` objects.

function
: The function this block is defined in.

is_terminated
: Whether this block ends with a *terminator instruction*.

terminator
: The block's *terminator instruction*, if any. Otherwise None.

class `llvmlite.ir.`**BlockAddress**
: A constant representing an address of a basic block.

Block address constants have the following attributes:

function
: The function the basic block is defined in.

basic_block
: The basic block. Must be a part of `function`.

3.2.1 Global values

Global values are values accessible using a module-wide name.

class `llvmlite.ir.`**GlobalValue**
: The base class for global values. Global values have the following writable attribute:

linkage
: A Python string describing the linkage behaviour of the global value (e.g. whether it is visible from other modules). Default is the empty string, meaning "external".

class `llvmlite.ir.`**GlobalVariable** (*module*, *typ*, *name*, *addrspace=0*)
: A global variable. *module* is where the variable will be defined. *typ* is the variable's type. *name* is the variable's name (a Python string). *addrspace* is an optional address space to store the variable in.

typ cannot be a function type. To declare a global function, use `Function`.

The returned Value's actual type is a pointer to *typ*. To read (respectively write) the variable's contents, you need to `load()` from (respectively `store()` to) the returned Value.

Global variables have the following writable attributes:

global_constant
: If true, the variable is declared a constant, i.e. its contents cannot be ever modified. Default is False.

unnamed_addr
: If true, the address of the variable is deemed insignificant, i.e. it will be merged with other variables which have the same initializer. Default is False.

initializer
: The variable's initialization value (probably a `Constant` of type *typ*). Default is None, meaning the variable is uninitialized.

class `llvmlite.ir.`**Function** (*module*, *typ*, *name*)
: A global function. *module* is where the function will be defined. *typ* is the function's type (a `FunctionType` instance). *name* is the function's name (a Python string).

If a global function has any basic blocks, it is a *function definition*. Otherwise, it is a *function declaration*.

Functions have the following methods and attributes:

append_basic_block(*name=''*)
Append a *basic block* to this function's body. If *name* is non empty, it names the block's entry *label*.

A new `Block` is returned.

insert_basic_block(*before*, *name=''*)
Similar to `append_basic_block()`, but inserts it before the basic block *before* in the function's list of basic blocks.

args
The function's arguments as a tuple of `Argument` instances.

attributes
A set of function attributes. Each optional attribute is a Python string. By default this is empty. Use the `add()` method to add an attribute:

```
fnty = ir.FunctionType(ir.DoubleType(), (ir.DoubleType(),))
fn = Function(module, fnty, "sqrt")
fn.attributes.add("alwaysinline")
```

calling_convention
The function's calling convention (a Python string). Default is the empty string.

is_declaration
Whether the global function is a declaration (True) or a definition (False).

3.2.2 Instructions

Every *instruction* is also a value: it has a name (the recipient's name), a well-defined type, and can be used as an operand in other instructions or in literals.

Instruction types should usually not be instantiated directly. Instead, use the helper methods on the `IRBuilder` class.

class `llvmlite.ir.`**Instruction**
The base class for all instructions. Instructions have the following method and attributes:

set_metadata(*name*, *node*)
Add an instruction-specific metadata named *name* pointing to the given metadata *node* (a `MDValue`).

function
The function this instruction is part of.

module
The module this instruction's function is defined in.

class `llvmlite.ir.`**PredictableInstr**
The class of instructions for which we can specificy the probabilities of different outcomes (e.g. a switch or a conditional branch). Predictable instructions have the following method:

set_weights(*weights*)
Set the weights of the instruction's possible outcomes. *weights* is a sequence of positive integers, each corresponding to a different outcome and specifying its relative probability compared to other outcomes (the greater, the likelier).

class `llvmlite.ir.`**SwitchInstr**
A switch instruction. Switch instructions have the following method:

add_case(*val*, *block*)
Add a case to the switch instruction. *val* should be a `Constant` or a Python value compatible with the switch instruction's operand type. *block* is a `Block` to jump to if, and only if, *val* and the switch operand compare equal.

class llvmlite.ir.**IndirectBranch**
An indirect branch instruction. Indirect branch instructions have the following method:

add_destination(*value*, *block*)
Add an outgoing edge. The indirect branch instruction must refer to every basic block it can transfer control to.

class llvmlite.ir.**PhiInstr**
A phi instruction. Phi instructions have the following method:

add_incoming(*value*, *block*)
Add an incoming edge. Whenever transfer is controlled from *block* (a `Block`), the phi instruction takes the given *value*.

class llvmlite.ir.**LandingPad**
A landing pad. Landing pads have the following method:

add_clause(*value*, *block*)
Add a catch or filter clause. Create catch clauses using `CatchClause`, and filter clauses using `FilterClause`.

3.2.3 Landing pad clauses

Landing pads have the following classes associated with them:

class llvmlite.ir.**CatchClause**(*value*)
A catch clause. Instructs the personality function to compare the in-flight exception typeinfo with *value*, which should have type *IntType(8).as_pointer().as_pointer()*.

class llvmlite.ir.**FilterClause**(*value*)
A filter clause. Instructs the personality function to check inclusion of the the in-flight exception typeinfo in *value*, which should have type *ArrayType(IntType(8).as_pointer().as_pointer(), ...)*.

3.3 Modules

A module is a compilation unit. It defines a set of related functions, global variables and metadata. In the IR layer, a module is representated by the `Module` class.

class llvmlite.ir.**Module**(*name=''*)
Create a module. The optional *name*, a Python string, can be specified for informational purposes.

Modules have the following methods and attributes:

add_global(*globalvalue*)
Add the given *globalvalue* (a `GlobalValue`) to this module. It should have a unique name in the whole module.

add_metadata(*operands*)
Add an unnamed *metadata* node to the module with the given *operands* (a list of metadata-compatible values). If another metadata node with equal operands already exists in the module, it is reused instead. A `MDValue` is returned.

add_named_metadata(*name*)
Add a named metadata with the given *name*. A *NamedMetaData* is returned.

get_global(*name*)
Get the *global value* (a *GlobalValue*) with the given *name*. `KeyError` is raised if it doesn't exist.

get_named_metadata(*name*)
Return the named metadata with the given *name*. `KeyError` is raised if it doesn't exist.

get_unique_name(*name*)
Return a unique name accross the whole module. *name* is the desired name, but a variation can be returned if it is already in use.

data_layout
A string representing the data layout in LLVM format.

functions
The list of functions (as *Function* instances) declared or defined in the module.

global_values
An iterable of global values in this module.

triple
A string representing the target architecture in LLVM "triple" form.

3.4 IR builders

Contents

IRBuilder is the workhorse of LLVM *IR* generation. It allows you to fill the *basic blocks* of your functions with LLVM instructions.

A *IRBuilder* internally maintains a current basic block, and a pointer inside the block's list of instructions. When adding a new instruction, it is inserted at that point and the pointer is then advanced after the new instruction.

3.4.1 Instantiation

class `llvmlite.ir.`**`IRBuilder`**(*block=None*)

Create a new IR builder. If *block* (a *`Block`*) is given, the builder starts right at the end of this basic block.

3.4.2 Properties

`IRBuilder` has the following attributes:

`IRBuilder.`**`block`**

The basic block the builder is operating on.

`IRBuilder.`**`function`**

The function the builder is operating on.

`IRBuilder.`**`module`**

The module the builder's function is defined in.

3.4.3 Utilities

`IRBuilder.`**`append_basic_block`**(*name='‘*)

Append a basic block, with the given optional *name*, to the current function. The current block is not changed. A *`Block`* is returned.

3.4.4 Positioning

The following *`IRBuilder`* methods help you move the current instruction pointer around:

`IRBuilder.`**`position_before`**(*instruction*)

Position immediatly before the given *instruction*. The current block is also changed to the instruction's basic block.

`IRBuilder.`**`position_after`**(*instruction*)

Position immediatly after the given *instruction*. The current block is also changed to the instruction's basic block.

`IRBuilder.`**`position_at_start`**(*block*)

Position at the start of the basic *block*.

`IRBuilder.`**`position_at_end`**(*block*)

Position at the end of the basic *block*.

The following context managers allow you to temporarily switch to another basic block, then go back where you were:

`IRBuilder.`**`goto_block`**(*block*)

A context manager which positions the builder either at the end of the basic *block*, if it is not terminated, or just before the *block*‘s terminator:

```
new_block = builder.append_basic_block('foo')
with builder.goto_block(new_block):
   # Now the builder is at the end of *new_block*
   # ... add instructions

# Now the builder has returned to its previous position
```

IRBuilder.**goto_entry_block**()

Just like `goto_block()`, but with the current function's entry block.

3.4.5 Flow control helpers

The following context managers make it easier to create conditional code.

IRBuilder.**if_then**(*pred*, *likely=None*)

A context manager which creates a basic block whose execution is conditioned on predicate *pred* (a value of type `IntType(1)`). Another basic block is created for instructions after the conditional block. The current basic block is terminated with a conditional branch based on *pred*.

When the context manager is entered, the builder positions at the end of the conditional block. When the context manager is exited, the builder positions at the start of the continuation block.

If *likely* is not None, it indicates whether *pred* is likely to be true, and metadata is emitted to specify branch weights in accordance.

IRBuilder.**if_else**(*pred*, *likely=None*)

A context manager which sets up two basic blocks whose execution is condition on predicate *pred* (a value of type `IntType(1)`). *likely* has the same meaning as in if_then().

A pair of context managers is yield'ed. Each of them acts as a `if_then()` context manager: the first one for the block to be executed if *pred* is true, the second one for the block to be executed if *pred* is false.

When the context manager is exited, the builder is positioned on a new continuation block which both conditional blocks jump into.

Typical use:

```
with builder.if_else(pred) as (then, otherwise):
    with then:
        # emit instructions for when the predicate is true
    with otherwise:
        # emit instructions for when the predicate is false
# emit instructions following the if-else block
```

3.4.6 Instruction building

The following methods all insert a new instruction (a `Instruction` instance) at the current index in the current block. The new instruction is returned.

An instruction's operands are most always *values*.

Many of these methods also take an optional *name* argument, specifying the local *name* of the result value. If not given, a unique name is automatically generated.

Arithmetic

The *flags* argument in the methods below is an optional sequence of strings serving as modifiers of the instruction's semantics. Examples include the fast-math flags for floating-point operations, or whether wraparound on overflow can be ignored on integer operations.

IRBuilder.**shl**(*lhs*, *rhs*, *name=''*, *flags=()*)

Left-shift *lhs* by *rhs* bits.

IRBuilder.**lshr**(*lhs*, *rhs*, *name=''*, *flags=()*)

Logical right-shift *lhs* by *rhs* bits.

`IRBuilder.`**`ashr`**(*lhs*, *rhs*, *name=''*, *flags=()*)
Arithmetic (signed) right-shift *lhs* by *rhs* bits.

`IRBuilder.`**`add`**(*lhs*, *rhs*, *name=''*, *flags=()*)
Integer add *lhs* and *rhs*.

`IRBuilder.`**`fadd`**(*lhs*, *rhs*, *name=''*, *flags=()*)
Floating-point add *lhs* and *rhs*.

`IRBuilder.`**`sub`**(*lhs*, *rhs*, *name=''*, *flags=()*)
Integer subtract*rhs* from *lhs*.

`IRBuilder.`**`fadd`**(*lhs*, *rhs*, *name=''*, *flags=()*)
Floating-point subtract *rhs* from *lhs*.

`IRBuilder.`**`mul`**(*lhs*, *rhs*, *name=''*, *flags=()*)
Integer multiply *lhs* with *rhs*.

`IRBuilder.`**`fmul`**(*lhs*, *rhs*, *name=''*, *flags=()*)
Floating-point multiply *lhs* with *rhs*.

`IRBuilder.`**`sdiv`**(*lhs*, *rhs*, *name=''*, *flags=()*)
Signed integer divide *lhs* by *rhs*.

`IRBuilder.`**`udiv`**(*lhs*, *rhs*, *name=''*, *flags=()*)
Unsigned integer divide *lhs* by *rhs*.

`IRBuilder.`**`fdiv`**(*lhs*, *rhs*, *name=''*, *flags=()*)
Floating-point divide *lhs* by *rhs*.

`IRBuilder.`**`srem`**(*lhs*, *rhs*, *name=''*, *flags=()*)
Signed integer remainder of *lhs* divided by *rhs*.

`IRBuilder.`**`urem`**(*lhs*, *rhs*, *name=''*, *flags=()*)
Unsigned integer remainder of *lhs* divided by *rhs*.

`IRBuilder.`**`frem`**(*lhs*, *rhs*, *name=''*, *flags=()*)
Floating-point remainder of *lhs* divided by *rhs*.

`IRBuilder.`**`and_`**(*lhs*, *rhs*, *name=''*, *flags=()*)
Bitwise AND *lhs* with *rhs*.

`IRBuilder.`**`or_`**(*lhs*, *rhs*, *name=''*, *flags=()*)
Bitwise OR *lhs* with *rhs*.

`IRBuilder.`**`xor`**(*lhs*, *rhs*, *name=''*, *flags=()*)
Bitwise XOR *lhs* with *rhs*.

`IRBuilder.`**`not_`**(*value*, *name=''*)
Bitwise complement *value*.

`IRBuilder.`**`neg`**(*value*, *name=''*)
Negate *value*.

Conversions

`IRBuilder.`**`trunc`**(*value*, *typ*, *name=''*)
Truncate integer *value* to integer type *typ*.

`IRBuilder.`**`zext`**(*value*, *typ*, *name=''*)
Zero-extend integer *value* to integer type *typ*.

`IRBuilder.`**`sext`**(*value*, *typ*, *name=''*)
    Sign-extend integer *value* to integer type *typ*.

`IRBuilder.`**`fptrunc`**(*value*, *typ*, *name=''*)
    Truncate (approximate) floating-point *value* to floating-point type *typ*.

`IRBuilder.`**`fpext`**(*value*, *typ*, *name=''*)
    Extend floating-point *value* to floating-point type *typ*.

`IRBuilder.`**`fptosi`**(*value*, *typ*, *name=''*)
    Convert floating-point *value* to signed integer type *typ*.

`IRBuilder.`**`fptoui`**(*value*, *typ*, *name=''*)
    Convert floating-point *value* to unsigned integer type *typ*.

`IRBuilder.`**`sitofp`**(*value*, *typ*, *name=''*)
    Convert signed integer *value* to floating-point type *typ*.

`IRBuilder.`**`uitofp`**(*value*, *typ*, *name=''*)
    Convert unsigned integer *value* to floating-point type *typ*.

`IRBuilder.`**`ptrtoint`**(*value*, *typ*, *name=''*)
    Convert pointer *value* to integer type *typ*.

`IRBuilder.`**`inttoptr`**(*value*, *typ*, *name=''*)
    Convert integer *value* to pointer type *typ*.

`IRBuilder.`**`bitcast`**(*value*, *typ*, *name=''*)
    Convert pointer *value* to pointer type *typ*.

`IRBuilder.`**`addrspacecast`**(*value*, *typ*, *name=''*)
    Convert pointer *value* to pointer type *typ* of different address space.

Comparisons

`IRBuilder.`**`icmp_signed`**(*cmpop*, *lhs*, *rhs*, *name=''*)
    Signed integer compare *lhs* with *rhs*. *cmpop*, a string, can be one of <, <=, ==, !=, >=, >.

`IRBuilder.`**`icmp_unsigned`**(*cmpop*, *lhs*, *rhs*, *name=''*)
    Unsigned integer compare *lhs* with *rhs*. *cmpop*, a string, can be one of <, <=, ==, !=, >=, >.

`IRBuilder.`**`fcmp_ordered`**(*cmpop*, *lhs*, *rhs*, *name=''*)
    Floating-point ordered compare *lhs* with *rhs*. *cmpop*, a string, can be one of <, <=, ==, !=, >=, >, `ord`, `uno`.

`IRBuilder.`**`fcmp_unordered`**(*cmpop*, *lhs*, *rhs*, *name=''*)
    Floating-point unordered compare *lhs* with *rhs*. *cmpop*, a string, can be one of <, <=, ==, !=, >=, >, `ord`, `uno`.

Conditional move

`IRBuilder.`**`select`**(*cond*, *lhs*, *rhs*, *name=''*)
    Two-way select: *lhs* if *cond* else *rhs*.

Phi

`IRBuilder.`**`phi`**(*typ*, *name=''*)
    Create a phi node. Add incoming edges and their values using the *`add_incoming()`* method on the return value.

Aggregate operations

`IRBuilder.`**`extract_value`**`(`*agg*, *index*, *name=''*`)`
Extract the element at *index* of the *aggregate value* *agg*. *index* may be an integer or a sequence of integers. If *agg* is of an array type, indices can be arbitrary values; if *agg* is of a struct type, indices have to be constant.

`IRBuilder.`**`insert_value`**`(`*agg*, *value*, *index*, *name=''*`)`
Build a copy of *aggregate value* *agg* by setting the new *value* at *index*. *index* can be of the same types as in `extract_value()`.

Memory

`IRBuilder.`**`alloca`**`(`*typ*, *size=None*, *name=''*`)`
Statically allocate a stack slot for *size* values of type *typ*. If *size* is not given, a stack slot for one value is allocated.

`IRBuilder.`**`load`**`(`*ptr*, *name=''*`)`
Load value from pointer *ptr*.

`IRBuilder.`**`store`**`(`*value*, *ptr*`)`
Store *value* to pointer *ptr*.

`IRBuilder.`**`gep`**`(`*ptr*, *indices*, *inbounds=False*, *name=''*`)`
The *getelementptr* instruction. Given a pointer *ptr* to an aggregate value, compute the address of the inner element given by the sequence of *indices*.

`llvmlite.ir.`**`cmpxchg`**`(`*ptr*, *cmp*, *val*, *ordering*, *failordering=None*, *name=''*`)`
Atomic compare-and-swap at address *ptr*. *cmp* is the value to compare the contents with, *val* the new value to be swapped into. Optional *ordering* and *failordering* specify the memory model for this instruction.

`llvmlite.ir.`**`atomic_rmw`**`(`*op*, *ptr*, *val*, *ordering*, *name=''*`)`
Atomic in-memory operation *op* at address *ptr*, with operand *val*. *op* is a string specifying the operation (e.g. `add` or `sub`). The optional *ordering* specifies the memory model for this instruction.

Function call

`IRBuilder.`**`call`**`(`*fn*, *args*, *name=''*, *cconv=None*, *tail=False*`)`
Call function *fn* with arguments *args* (a sequence of values). *cconc* is the optional calling convention. *tail*, if true, is a hint for the optimizer to perform tail-call optimization.

Branches

These instructions are all *terminators*.

`IRBuilder.`**`branch`**`(`*target*`)`
Unconditional jump to the *target* (a `Block`).

`IRBuilder.`**`cbranch`**`(`*cond*, *truebr*, *falsebr*`)`
Conditional jump to either *truebr* or *falsebr* (both `Block` instances), depending on *cond* (a value of type `IntType(1)`). This instruction is a `PredictableInstr`.

`IRBuilder.`**`ret`**`(`*value*`)`
Return the *value* from the current function.

`IRBuilder.`**`ret_void`**`()`
Return from the current function without a value.

`IRBuilder.`**`switch`**`(`*value*, *default*`)`
Switch to different blocks based on the *value*. *default* is the block to switch to if no other block is matched.

Add non-default targets using the `add_case()` method on the return value.

`IRBuilder.`**`indirectbr`**`(`*address*`)`
Jump to basic block with address *address* (a value of type *IntType(8).as_pointer()*). A block address can be obtained using the `BlockAddress` constant.

Add all possible jump destinations using the `add_destination()` method on the return value.

Exception handling

`IRBuilder.invoke(self, fn, args, normal_to, unwind_to, name='', cconv=None, tail=False)`
Call function *fn* with arguments *args* (a sequence of values). *cconc* is the optional calling convention. *tail*, if true, is a hint for the optimizer to perform tail-call optimization.

If the function *fn* returns normally, control is transferred to *normal_to*. Otherwise, it is transferred to *unwind_to*, the first non-phi instruction of which must be `LandingPad`.

`IRBuilder.`**`landingpad`**`(`*typ*, *personality*, *name=''*, *cleanup=False*`)`
Describe which exceptions this basic block can handle.

typ specifies the return type of the landing pad. It is a structure with two pointer-sized fields. *personality* specifies an exception personality function. *cleanup* specifies whether control should be always transferred to this landing pad, even when no matching exception is caught.

Add landing pad clauses using the `add_clause()` method on the return value.

There are two kinds of landing pad clauses:

- A `CatchClause`, which specifies a typeinfo for a single exception to be caught. The typeinfo is a value of type *IntType(8).as_pointer().as_pointer()*;
- A `FilterClause`, which specifies an array of typeinfos.

Every landing pad must either contain at least one clause, or be marked for cleanup.

The semantics of a landing pad are entirely determined by the personality function. See Exception handling in LLVM for details on the way LLVM handles landing pads in the optimizer, and Itanium exception handling ABI for details on the implementation of personality functions.

`IRBuilder.`**`resume`**`(`*landingpad*`)`
Resume an exception caught by landing pad *landingpad*. Used to indicate that the landing pad did not catch the exception after all (perhaps because it only performed cleanup).

Miscellaneous

`IRBuilder.`**`assume`**`(`*cond*`)`
Let the LLVM optimizer assume that *cond* (a value of type `IntType(1)`) is true.

`IRBuilder.`**`unreachable`**`()`
Mark an unreachable point in the code.

3.5 Example

3.5.1 A trivial function

Define a function adding two double-precision floating-point numbers.

```
"""
This file demonstrates a trivial function "fpadd" returning the sum of
two floating-point numbers.
"""

from llvmlite import ir

# Create some useful types
double = ir.DoubleType()
fnty = ir.FunctionType(double, (double, double))

# Create an empty module...
module = ir.Module(name=__file__)
# and declare a function named "fpadd" inside it
func = ir.Function(module, fnty, name="fpadd")

# Now implement the function
block = func.append_basic_block(name="entry")
builder = ir.IRBuilder(block)
a, b = func.args
result = builder.fadd(a, b, name="res")
builder.ret(result)

# Print the module IR
print(module)
```

The generated LLVM *IR* is printed out at the end, and should look like this:

```
; ModuleID = "examples/ir_fpadd.py"
target triple = "unknown-unknown-unknown"
target datalayout = ""

define double @"fpadd"(double %".1", double %".2")
{
entry:
  %"res" = fadd double %".1", %".2"
  ret double %"res"
}
```

To learn how to compile and execute this function, refer to the *binding layer* documentation.

CHAPTER 4

llvmlite.binding – The LLVM binding layer

The *llvmlite.binding* module provides classes to interact with functionalities of the LLVM library. They generally mirror concepts of the C++ API closely. A small subset of the LLVM API is mirrored, though: only those parts that have proven useful to implement Numba's JIT compiler.

4.1 Initialization and finalization

These functions need only be called once per process invocation.

`llvmlite.binding.`**`initialize`**`()`
    Initialize the LLVM core.

`llvmlite.binding.`**`initialize_native_target`**`()`
    Initialize the native (host) target. Calling this function once is necessary before doing any code generation.

`llvmlite.binding.`**`initialize_native_asmprinter`**`()`
    Initialize the native assembly printer.

`llvmlite.binding.`**`shutdown`**`()`
    Shutdown the LLVM core.

4.2 Dynamic libraries and symbols

These functions tell LLVM how to resolve external symbols referred from compiled LLVM code.

`llvmlite.binding.`**`add_symbol`**`(`*name*, *address*`)`
    Register the *address* of global symbol *name*, for use from LLVM-compiled functions.

`llvmlite.binding.`**`address_of_symbol`**`(`*name*`)`
    Get the in-process address of symbol named *name*. An integer is returned, or None if the symbol isn't found.

`llvmlite.binding.`**`load_library_permanently`**`(`*filename*`)`
    Load an external shared library. *filename* should be the path to the shared library file.

4.3 Target information

Target information allows you to inspect and modify aspects of the code generation, such as which CPU is targetted or what optimization level is desired.

Minimal use of this module would be to create a *TargetMachine* for later use in code generation:

```
from llvmlite import binding
target = binding.Target.from_default_triple()
target_machine = target.create_target_machine()
```

4.3.1 Functions

llvmlite.binding.**get_default_triple**()

Return the default target triple LLVM is configured to produce code for, as a string. This represents the host's architecture and platform.

llvmlite.binding.**get_host_cpu_name**()

Get the name of the host's CPU as a string. You can use the return value with *Target.create_target_machine()*.

llvmlite.binding.**create_target_data**(*data_layout*)

Create a *TargetData* representing the given *data_layout* (a string).

llvmlite.binding.**create_target_library_info**(*triple*)

Create a *TargetLibraryInfo* for the given *triple* string.

4.3.2 Classes

class llvmlite.binding.**TargetData**

A class providing functionality around a given data layout. It specifies how the different types are to be represented in memory. Use *create_target_data()* to instantiate.

add_pass(*pm*)

Add an optimization pass based on this data layout to the *PassManager* instance *pm*.

get_abi_size(*type*)

Get the ABI-mandated size of the LLVM *type* (as returned by *ValueRef.type*). An integer is returned.

get_pointee_abi_size(*type*)

Similar to *get_abi_size()*, but assumes *type* is a LLVM pointer type, and returns the ABI-mandated size of the type pointed to by. This is useful for global variables (whose type is really a pointer to the declared type).

get_pointee_abi_alignment(*type*)

Similar to *get_pointee_abi_size()*, but return the ABI-mandated alignment rather than the ABI size.

class llvmlite.binding.**Target**

A class representing a compilation target. The following factories are provided:

classmethod from_triple(*triple*)

Create a new *Target* instance for the given *triple* string denoting the target platform.

classmethod from_default_triple()

Create a new *Target* instance for the default platform LLVM is configured to produce code for. This is equivalent to calling `Target.from_triple(get_default_triple())`

The following methods and attributes are available:

description

A description of the target.

name
The name of the target.

triple
The triple (a string) uniquely identifying the target, for example `"x86_64-pc-linux-gnu"`.

create_target_machine(*cpu='', features='', opt=2, reloc='default', codemodel='jitdefault'*)
Create a new *`TargetMachine`* instance for this target and with the given options. *cpu* is an optional CPU name to specialize for. *features* is a comma-separated list of target-specific features to enable or disable. *opt* is the optimization level, from 0 to 3. *reloc* is the relocation model. *codemodel* is the code model. The defaults for *reloc* and *codemodel* are appropriate for JIT compilation.

Tip: To list the available CPUs and features for a target, execute `llc -mcpu=help` on the command line.

class `llvmlite.binding.`**TargetMachine**
A class holding all the settings necessary for proper code generation (including target information and compiler options). Instantiate using *`Target.create_target_machine()`*.

add_analysis_passes(*pm*)
Register analysis passes for this target machine with the *`PassManager`* instance *pm*.

emit_object(*module*)
Represent the compiled *module* (a *`ModuleRef`* instance) as a code object, suitable for use with the platform's linker. A bytestring is returned.

emit_assembly(*module*)
Return the compiled *module*'s native assembler, as a string.

`initialize_native_asmprinter()` must have been called first.

target_data
The *`TargetData`* associated with this target machine.

class `llvmlite.binding.`**TargetLibraryInfo**
This class provides information about what library functions are available for the current target. Instantiate using *`create_target_library_info()`*.

add_pass(*pm*)
Add an optimization pass based on this library info to the *`PassManager`* instance *pm*.

disable_all()
Disable all "builtin" functions.

get_libfunc(*name*)
Get the library function *name*. `NameError` is raised if not found.

set_unavailable(*libfunc*)
Mark the library function *libfunc* (as returned by *`get_libfunc()`*) unavailable.

4.4 Modules

While they conceptually represent the same thing, modules in the *IR layer* and modules in the *binding layer* don't have the same roles and don't expose the same API.

While modules in the IR layer allow to build and group functions together, modules in the binding layer give access to compilation, linking and execution of code. To distinguish between the two, the module class in the binding layer is called *`ModuleRef`* as opposed to *`llvmlite.ir.Module`*.

To go from one realm to the other, you must use the `parse_assembly()` function.

4.4.1 Factory functions

A module can be created from the following factory functions:

`llvmlite.binding.`**`parse_assembly`**`(`*llvmir*`)`

Parse the given *llvmir*, a string containing some LLVM IR code. If parsing is successful, a new `ModuleRef` instance is returned.

llvmir can be obtained, for example, by calling `str()` on a `llvmlite.ir.Module` object.

`llvmlite.binding.`**`parse_bitcode`**`(`*bitcode*`)`

Parse the given *bitcode*, a bytestring containing the LLVM bitcode of a module. If parsing is successful, a new `ModuleRef` instance is returned.

bitcode can be obtained, for example, by calling `ModuleRef.as_bitcode()`.

4.4.2 The ModuleRef class

class `llvmlite.binding.`**`ModuleRef`**

A wrapper around a LLVM module object. The following methods and properties are available:

`as_bitcode``()`

Return the bitcode of this module as a bytes object.

`get_function``(`*name*`)`

Get the function with the given *name* in this module. If found, a `ValueRef` is returned. Otherwise, `NameError` is raised.

`get_global_variable``(`*name*`)`

Get the global variable with the given *name* in this module. If found, a `ValueRef` is returned. Otherwise, `NameError` is raised.

`link_in``(`*other*, *preserve=False*`)`

Link the *other* module into this module, resolving references wherever possible. If *preserve* is true, the other module is first copied in order to preserve its contents; if *preserve* is false, the other module is not usable after this call anymore.

`verify``()`

Verify the module's correctness. `RuntimeError` is raised on error.

`data_layout`

The data layout string for this module. This attribute is settable.

`functions`

An iterator over the functions defined in this module. Each function is a `ValueRef` instance.

`global_variables`

An iterator over the global variables defined in this module. Each global variable is a `ValueRef` instance.

`name`

The module's identifier, as a string. This attribute is settable.

`triple`

The platform "triple" string for this module. This attribute is settable.

4.5 Value references

A value reference is a wrapper around a LLVM value for you to inspect. You can’t create one yourself; instead, you’ll get them from methods of the *ModuleRef* class.

4.5.1 Enumerations

class `llvmlite.binding.`**`Linkage`**

The different linkage types allowed for global values. The following values are provided:

`external`

`available_externally`

`linkonce_any`

`linkonce_odr`

`linkonce_odr_autohide`

`weak_any`

`weak_odr`

`appending`

`internal`

`private`

`dllimport`

`dllexport`

`external_weak`

`ghost`

`common`

`linker_private`

`linker_private_weak`

4.5.2 The ValueRef class

class `llvmlite.binding.`**`ValueRef`**

A wrapper around a LLVM value. The following properties are available:

`is_declaration`

True if the global value is a mere declaration, False if it is defined in the given module.

`linkage`

The linkage type (a *Linkage* instance) for this value. This attribute is settable.

`module`

The module (a *ModuleRef* instance) this value is defined in.

`name`

This value’s name, as a string. This attribute is settable.

type
This value's LLVM type. An opaque object is returned. It can be used with e.g. *TargetData.get_abi_size()*.

4.6 Execution engine

The execution engine is where actual code generation and execution happens. The currently supported LLVM version (LLVM 3.6) exposes one execution engine, named MCJIT.

4.6.1 Functions

llvmlite.binding.**create_mcjit_compiler**(*module*, *target_machine*)
Create a MCJIT-powered engine from the given *module* and *target_machine*. A *ExecutionEngine* instance is returned. The *module* need not contain any code.

llvmlite.binding.**check_jit_execution**()
Ensure the system allows creation of executable memory ranges for JIT-compiled code. If some security mechanism (such as SELinux) prevents it, an exception is raised. Otherwise the function returns silently.

Calling this function early can help diagnose system configuration issues, instead of letting JIT-compiled functions crash mysteriously.

4.6.2 The ExecutionEngine class

class llvmlite.binding.**ExecutionEngine**
A wrapper around a LLVM execution engine. The following methods and properties are available:

add_module(*module*)
Add the *module* (a *ModuleRef* instance) for code generation. When this method is called, ownership of the module is transferred to the execution engine.

finalize_object()
Make sure all modules owned by the execution engine are fully processed and "usable" for execution.

get_pointer_to_function(*gv*)

Warning: This function is deprecated. User should use *ExecutionEngine.get_function_address()* and *ExecutionEngine.get_global_value_address* instead

Return the address of the function value *gv*, as an integer. The value should have been looked up on one of the modules owned by the execution engine.

Note: This method may implicitly generate code for the object being looked up.

Note: This function is formerly an alias to `get_pointer_to_global()`, which is now removed because it returns an invalid address in MCJIT when given a non-function global value.

get_function_address(*name*)
Return the address of the function named *name* as an integer.

get_global_value_address(*name*)
Return the address of the global value named *name* as an integer.

remove_module(*module*)
Remove the *module* (a `ModuleRef` instance) from the modules owned by the execution engine. This allows releasing the resources owned by the module without destroying the execution engine.

set_object_cache(*notify_func=None*, *getbuffer_func=None*)
Set the object cache callbacks for this engine.

notify_func, if given, is called whenever the engine has finished compiling a module. It is passed two arguments `(module, buffer)`. The first argument *module* is a `ModuleRef` instance. The second argument *buffer* is a bytes object of the code generated for the module. The return value is ignored.

getbuffer_func, if given, is called before the engine starts compiling a module. It is passed one argument, *module*, a `ModuleRef` instance of the module being compiled. The function can return `None`, in which case the module will be compiled normally. Or it can return a bytes object of native code for the module, which will bypass compilation entirely.

target_data
The `TargetData` used by the execution engine.

4.7 Optimization passes

LLVM gives you the possibility to fine-tune optimization passes. llvmlite exposes several of these parameters. Optimization passes are managed by a pass manager; there are two kinds thereof: `FunctionPassManager`, for optimizations which work on single functions, and `ModulePassManager`, for optimizations which work on whole modules.

To instantiate any of those pass managers, you first have to create and configure a `PassManagerBuilder`.

class `llvmlite.binding.`**PassManagerBuilder**
Create a new pass manager builder. This object centralizes optimization settings. The following method is available:

populate(*pm*)
Populate the pass manager *pm* with the optimization passes configured in this pass manager builder.

The following writable properties are also available:

disable_unroll_loops
If true, disable loop unrolling.

inlining_threshold
The integer threshold for inlining a function into another. The higher, the more likely inlining a function is. This attribute is write-only.

loop_vectorize
If true, allow vectorizing loops.

opt_level
The general optimization level as an integer between 0 and 3.

size_level
Whether and how much to optimize for size. An integer between 0 and 2.

slp_vectorize
If true, enable the “SLP vectorizer”, which uses a different algorithm from the loop vectorizer. Both may be enabled at the same time.

class `llvmlite.binding.`**`PassManager`**
The base class for pass managers.

class `llvmlite.binding.`**`ModulePassManager`**
Create a new pass manager to run optimization passes on a module. Use `PassManagerBuilder.populate()` to add optimization passes.

The following method is available:

`run`(*module*)
Run optimization passes on the *module* (a `ModuleRef` instance). True is returned if the optimizations made any modification to the module, False instead.

class `llvmlite.binding.`**`FunctionPassManager`**(*module*)
Create a new pass manager to run optimization passes on a function of the given *module* (an `ModuleRef` instance). Use `PassManagerBuilder.populate()` to add optimization passes.

The following methods are available:

`finalize`()
Run all the finalizers of the optimization passes.

`initialize`()
Run all the initializers of the optimization passes.

`run`(*function*)
Run optimization passes on the *function* (a `ValueRef` instance). True is returned if the optimizations made any modification to the module, False instead.

4.8 Examples

4.8.1 Compiling a trivial function

Compile and execute the function defined in *A trivial function*. The function is compiled with no specific optimizations.

```
from __future__ import print_function

from ctypes import CFUNCTYPE, c_double

import llvmlite.binding as llvm


# All these initializations are required for code generation!
llvm.initialize()
llvm.initialize_native_target()
llvm.initialize_native_asmprinter()  # yes, even this one

llvm_ir = """
   ; ModuleID = "examples/ir_fpadd.py"
   target triple = "unknown-unknown-unknown"
   target datalayout = ""

   define double @"fpadd"(double %".1", double %".2")
   {
   entry:
     %"res" = fadd double %".1", %".2"
     ret double %"res"
   }
```

```
    """

def create_execution_engine():
    """
    Create an ExecutionEngine suitable for JIT code generation on
    the host CPU.  The engine is reusable for an arbitrary number of
    modules.
    """
    # Create a target machine representing the host
    target = llvm.Target.from_default_triple()
    target_machine = target.create_target_machine()
    # And an execution engine with an empty backing module
    backing_mod = llvm.parse_assembly("")
    engine = llvm.create_mcjit_compiler(backing_mod, target_machine)
    return engine


def compile_ir(engine, llvm_ir):
    """
    Compile the LLVM IR string with the given engine.
    The compiled module object is returned.
    """
    # Create a LLVM module object from the IR
    mod = llvm.parse_assembly(llvm_ir)
    mod.verify()
    # Now add the module and make sure it is ready for execution
    engine.add_module(mod)
    engine.finalize_object()
    return mod


engine = create_execution_engine()
mod = compile_ir(engine, llvm_ir)

# Look up the function pointer (a Python int)
func_ptr = engine.get_function_address("fpadd")

# Run the function via ctypes
cfunc = CFUNCTYPE(c_double, c_double, c_double)(func_ptr)
res = cfunc(1.0, 3.5)
print("fpadd(...) =", res)
```

CHAPTER 5

Contributing

llvmlite originated to fulfill the needs of the Numba project. It is still mostly maintained by the Numba team. As such, we tend to prioritize the needs and constraints of Numba over other conflicting desires. However, we welcome any contributions, under the form of *bug reports* or *pull requests*.

5.1 Communication

5.1.1 Mailing-list

For now, we use the Numba public mailing-list, which you can e-mail at numba-users@continuum.io. If you have any questions about contributing to llvmlite, it is ok to ask them on this mailing-list. You can subscribe and read the archives on Google Groups, and there is also a Gmane mirror allowing NNTP access.

5.1.2 Bug tracker

We use the Github issue tracker to track both bug reports and feature requests. If you report an issue, please include specifics:

- what you are trying to do;
- which operating system you have and which version of llvmlite you are running;
- how llvmlite is misbehaving, e.g. the full error traceback, or the unexpected results you are getting;
- as far as possible, a code snippet that allows full reproduction of your problem.

5.2 Pull requests

If you want to contribute code, we recommend you fork our Github repository, then create a branch representing your work. When your work is ready, you should submit it as a pull request from the Github interface.

5.3 Development rules

5.3.1 Coding conventions

All Python code should follow **PEP 8**. Our C++ code doesn't have a well-defined coding style (would it be nice to follow **PEP 7**?). Code and documentation should generally fit within 80 columns, for maximum readability with all existing tools (such as code review UIs).

5.3.2 Platform support

llvmlite is to be kept compatible with Python 2.6, 2.7, 3.3 and 3.4 under at least Linux, OS X and Windows. It only needs to be compatible with the currently supported LLVM version (currently, the 3.6 series).

We don't expect contributors to test their code on all platforms. Pull requests are automatically built and tested using Travis-CI. This takes care of Linux compatibility. Other operating systems are tested on an internal continuous integration platform at Continuum Analytics.

5.4 Documentation

This documentation is maintained in the `docs` directory inside the llvmlite repository. It is built using Sphinx.

You can edit the source files under `docs/source/`, after which you can build and check the documentation:

```
$ make html
$ open _build/html/index.html
```

CHAPTER 6

Glossary

basic block A sequence of instructions inside a function. A basic block always starts with a *label* and ends with a *terminator*. No other instruction inside the basic block can transfer control out of the block.

function declaration The specification of a function's prototype (including the argument and return types, and other information such as the calling convention), without an associated implementation. This is like a `extern` function declaration in C.

function definition A function's prototype (like in a *function declaration*) plus a body implementing the function.

getelementptr A LLVM *instruction* allowing to get the address of a subelement of an aggregate data structure.

See also:

Official documentation: 'getelementptr' Instruction

global value A named value accessible to all members of a module.

global variable A variable whose value is accessible to all members of a module. Under the hood, it is a constant pointer to a module-allocated slot of the given type.

All global variables are global values. However, the converse is not true: a function declaration or definition is not a global variable; it is only a *global value*.

instruction The fundamental element(s) used in implementing a LLVM function. LLVM instructions define a procedural assembly-like language.

IR, Intermediate Representation A high-level assembly language describing to LLVM the code to be compiled to native code.

label A branch target inside a function. A label always denotes the start of a *basic block*.

metadata Optional ancillary information which can be associated with LLVM instructions, functions, etc. Metadata is used to convey certain information which is not critical to the compiling of LLVM *IR* (such as the likelihood of a condition branch or the source code location corresponding to a given instruction).

module A compilation unit for LLVM *IR*. A module can contain any number of function declarations and definitions, global variables, and metadata.

terminator, terminator instruction A kind of *instruction* which explicitly transfers control to another part of the program (instead of simply going to the next instruction after it is executed). Examples are branches and function returns.

CHAPTER 7

Release Notes

7.1 v0.7.0

- PR #88: Provide hooks into the MCJIT object cache
- PR #87: Add indirect branches and exception handling APIs to ir.Builder.
- PR #86: Add ir.Builder APIs for integer arithmetic with overflow
- Issue #76: Fix non-Windows builds when LLVM was built using CMake
- Deprecate .get_pointer_to_global() and add .get_function_address() and .get_global_value_address() in ExecutionEngine.

7.2 v0.6.0

Enhancements:

- Switch from LLVM 3.5 to LLVM 3.6. The generated IR for metadata nodes has slightly changed, and the “old JIT” engine has been remove (only MCJIT is now available).
- Add an optional flags argument to arithmetic instructions on IRBuilder.
- Support attributes on the return type of a function.

7.3 v0.5.1

Fixes:

- Fix implicit termination of basic block in nested if_then()

7.4 v0.5.0

New documentation hosted at http://llvmlite.pydata.org

Enhancements:

- Add code-generation helpers from numba.cgutils
- Support for memset, memcpy, memmove intrinsics

Fixes:

- Fix string encoding problem when round-triping parse_assembly()

7.5 v0.4.0

Enhancements: * Add Module.get_global() * Renamd Module.global_variables to Module.global_values * Support loading library parmanently * Add Type.get_abi_alignment()

Fixes: * Expose LLVM version as a tuple

Patched LLVM 3.5.1: Updated to 3.5.1 with the same ELF relocation patched for v0.2.2.

7.6 v0.2.2

Enhancements: * Support for addrspacescast * Support for tail call, calling convention attribute * Support for IdentifiedStructType

Fixes: * GEP addrspace propagation * Various installation process fixes

Patched LLVM 3.5: The binaries from the numba binstar channel use a patched LLVM3.5 for fixing a LLVM ELF relocation bug that is caused by the use of 32-bit relative offset in 64-bit binaries. The problem appears to occur more often on hardened kernels, like in CentOS. The patched source code is available at: https://github.com/numba/llvm-mirror/releases/tag/3.5p1

7.7 v0.2.0

This is the first official release. It contains a few feature additions and bug fixes. It meets all requirements to replace llvmpy in numba and numbapro.

7.8 v0.1.0

This is the first release. This is released for beta testing llvmlite and numba before the official release.

Indices

- genindex
- modindex

Python Module Index