llvmlite Documentation

Release 0.24.0

Continuum Analytics

Nov 28, 2018

Contents

A lightweight LLVM-Python binding for writing JIT compilers

llvmlite provides a Python binding to LLVM for use in Numba. Numba previously relied on llvmpy.

Llvmpy became hard to maintain because:

- It has a cumbersome architecture.
- The C++11 requirement of recent LLVM versions does not go well with the compiler and runtime ABI requirements of some Python versions, especially under Windows.

Llvmpy also proved to be responsible for a sizable part 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.

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

CHAPTER 1

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:

- Construction of a *Module*, function by function, *Instruction* by instruction.
- Compilation and optimization of the module into machine code.

The construction of an LLVM module does not call the LLVM C++ API. Rather, it constructs the LLVM *intermediate representation* (IR) in pure Python. This is the role of the *IR layer*.

The compilation of an 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 role of the *binding layer*.

Once parsed, the module's source code cannot be modified, which loses the flexibility of the direct mapping of C++ APIs into Python that was provided by llvmpy but saves a great deal of maintenance.

CHAPTER 2

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 targeted to a specific LLVM feature version and works across all given bugfix releases of that version.

EXAMPLE: Llvmlite 0.12.0 works with LLVM 3.8.0 and 3.8.1, but it does not work with LLVM 3.7.0 or 3.9.0.

Numba's requirements determine which LLVM version is supported.

CHAPTER 3

API stability

At this time, we reserve the possibility of slightly breaking the llvmlite API at each release, for the following reasons:

- Changes in LLVM behaviour, such as differences in the IR across versions.
- As a young library, llvmlite has room for improvement or fixes to the existing APIs.

3.1 Installation

Contrary to what you might expect, llvmlite does *not* use any LLVM shared libraries that may be present on the system, or in the conda environment. The parts of LLVM required by llvmlite are statically linked at build time. As a result, installing llvmlite from a binary package does not also require the end user to install LLVM. (For more details on the reasoning behind this, see: *Why Static Linking to LLVM?*)

3.1.1 Pre-built binaries

Building LLVM for llvmlite is challenging, so we *strongly* recommend installing a binary package where we have built and tested everything for you. Official conda packages are available in the Anaconda distribution:

```
conda install llvmlite
```

Development releases are built from the Git master branch and uploaded to the Numba channel on Anaconda Cloud:

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

Binary wheels are also available for installation from PyPI:

```
pip install llvmlite
```

3.1.2 Building manually

Building llvmlite requires first building LLVM. Do not use prebuilt LLVM binaries from your OS distribution or the LLVM website! There will likely be a mismatch in version or build options, and LLVM will be missing certain patches that are critical for llvmlite operation.

Prerequisites

Before building, you must have the following:

- On Windows:
  - Visual Studio 2015 (Update 3) or later, to compile LLVM and llvmlite. The free Express edition is acceptable.
  - CMake installed.
- On Linux:
  - g++ (>= 4.8) and CMake
  - If building LLVM on Ubuntu, the linker may report an error if the development version of `libedit` is not installed. If you run into this problem, install `libedit-dev`.
- On Mac:
  - Xcode for the compiler tools, and CMake

Compiling LLVM

If you can build llvmlite inside a conda environment, you can install a prebuilt LLVM binary package and skip this step:

```
conda install -c numba llvmdev
```

The LLVM build process is fully scripted by conda-build, and the llvmdev recipe is the canonical reference for building LLVM for llvmlite. Please use it if at all possible!

The manual instructions below describe the main steps, but refer to the recipe for details:

#. Download the LLVM 6.0.0 source code. (Note that PPC64LE requires LLVM 6.0.1 for specific bug fixes.)

1. Download or git checkout the llvmlite source code.
2. Decompress the LLVM tar file and apply the following patches from the `llvmlite/conda-recipes/` directory:
   (a) `llvm-lto-static.patch`: Fix issue with LTO shared library on Windows
   (b) `D47188-svml.patch`: Add support for vectorized math functions via Intel SVML
   (c) `0001-Transforms-Add-missing-header-for-InstructionCombini.patch`: Fix release bug with LLVM 6.0.0, skip on LLVM 6.0.1.
   (d) `twine_cfg_undefined_behavior.patch`: Fix obscure memory corruption bug in LLVM that hasn't been fixed in master yet
3. **For Linux/macOS:**
   (a) `export PREFIX=desired_install_location CPU_COUNT=N` (`N` is number of parallel compile tasks)

(b) Run the build.sh script in the llvmdev conda recipe from the LLVM source directory

4. **For Windows:**

   (a) `set PREFIX=desired_install_location`

   (b) Run the bld.bat script in the llvmdev conda recipe from the LLVM source directory.

Compiling llvmlite

1. To build the llvmlite C wrapper, which embeds a statically linked copy of the required subset of LLVM, run the following from the llvmlite source directory:

```
python setup.py build
```

2. If your LLVM is installed in a nonstandard location, set the `LLVM_CONFIG` environment variable to the location of the corresponding `llvm-config` or `llvm-config.exe` executable. This variable must persist into the installation of llvmlite—for example, into a Python environment.

   EXAMPLE: If LLVM is installed in `/opt/llvm/` with the `llvm-config` binary located at `/opt/llvm/bin/llvm-config`, set `LLVM_CONFIG=/opt/llvm/bin/llvm-config`.

Installing

1. To validate your build, run the test suite by running:

```
python runtests.py
```

   or:

```
python -m llvmlite.tests
```

2. If the validation is successful, install by running:

```
python setup.py install
```

3.1.3 Why Static Linking to LLVM?

The llvmlite package uses LLVM via ctypes calls to a C wrapper that is statically linked to LLVM. Some people are surprised that llvmlite uses static linkage to LLVM, but there are several important reasons for this:

1. *The LLVM API has not historically been stable across releases* - Although things have improved since LLVM 4.0, there are still enough changes between LLVM releases to cause compilation issues if the right version is not matched with llvmlite.
2. *The LLVM shipped by most Linux distributions is not the version llvmlite needs* - The release cycles of Linux distributions will never line up with LLVM or llvmlite releases.
3. *We need to patch LLVM* - The binary packages of llvmlite are built against LLVM with a handful of patches to either fix bugs or to add features that have not yet been merged upstream. In some cases, we've had to carry patches for several releases before they make it into LLVM.
4. *We don't need most of LLVM* - We are sensitive to the install size of llvmlite, and a full build of LLVM is quite large. We can dramatically reduce the total disk needed by an llvmlite user (who typically doesn't need the rest of LLVM, ignoring the version matching issue) by statically linking to the library and pruning the symbols we do not need.

5. *Numba can use multiple LLVM builds at once* - Some Numba targets (AMD GPU, for example) may require different LLVM versions or non-mainline forks of LLVM to work. These other LLVMs can be wrapped in a similar fashion as llvmlite, and will stay isolated.
6. *We need to support Windows + Python 2.7* - Python 2.7 extensions on Windows needs to be built with Visual Studio 2008 for ABI compatibility reasons. This presents a serious issue as VS2008 can no longer build LLVM. The best workaround we have found (until the sunset of Python 2.7) is to build LLVM and the llvmlite C wrapper with VS2015 and call it through ctypes. This is not ideal, but experience has shown it seems to work.

Static linkage of LLVM was definitely not our goal early in Numba development, but seems to have become the only workable solution given our constraints.

3.2 User guide

3.2.1 IR layer—llvmlite.ir

The `llvmlite.ir` module contains classes and utilities to build the LLVM *intermediate representation* (IR) of native functions.

The provided APIs may sometimes look like LLVM's C++ APIs, but they never call into LLVM, unless otherwise noted. Instead, they construct a pure Python representation of the IR.

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

Types

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

class llvmlite.ir.**Type**

The base class for all types. 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*—an `llvmlite.binding.TargetData` instance.
- **get_abi_alignment**(*target_data*)
  Get the ABI alignment of this type, in bytes, according to the *target_data*—an `llvmlite.binding.TargetData` instance.

  NOTE: `get_abi_size()` and `get_abi_alignment()` call into the LLVM C++ API to get the requested information.
- **__call__**(*value*)
  A convenience method to create a `Constant` of this type with the given *value*:

```
>>> int32 = ir.IntType(32)
>>> c = int32(42)
>>> c
<ir.Constant type='i32' value=42>
>>> print(c)
i32 42
```

Atomic types

class llvmlite.ir.**PointerType**(*pointee*, *addrspace=0*)
The type of pointers to another type.

Pointer types expose the following attributes:

- **addrspace**
  The pointer's address space number. This optional integer allows you to choose a non-default address space—the meaning is platform dependent.
- **pointee**
  The type pointed to.

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

width
The width in bits.

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. Used only as the return type of a function without a return value.

Aggregate types

class llvmlite.ir.**Aggregate**
The base class for aggregate types. Never instantiate it directly. Aggregate types have the elements 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* is a Python integer representing the number of elements.

class llvmlite.ir.**LiteralStructType**(*elements*[, *packed=False*])
The class for literal struct types.

- *elements* is a sequence of element types for each member of the structure.
- *packed* controls whether to use packed layout.

class llvmlite.ir.**IdentifiedStructType**
The class for identified struct types. Identified structs are compared by name. It can be used to make opaque types.

Users should not create new instance directly. Use the `Context.get_identified_type` method instead.

An identified struct is created without a body (thus opaque). To define the struct body, use the `.set_body` method.

set_body(**elems*)
Define the structure body with a sequence of element types.

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 do not need to instantiate this type.

class llvmlite.ir.**MetaDataType**
The type for *Metadata*. You do not need to instantiate this type.

NOTE: This class was previously called "MetaData," but it was renamed for clarity.

Values

A *Module* consists mostly of values.

llvmlite.ir.**Undefined**
An undefined value, mapping to LLVM's `undef`.

class llvmlite.ir.**Value**
The base class for all IR values.

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 convert it to LLVM's `undef`.
- All types accept `None` and convert it to LLVM's `zeroinitializer`.
- *IntType* accepts any Python integer or boolean.
- *FloatType* and *DoubleType* accept any Python real number.
- Aggregate types—array and structure types—accept a sequence of Python values corresponding to the type's element types.
- *ArrayType* accepts a single `bytearray` instance to initialize the array from a string of bytes. This is useful for character constants.

classmethod literal_array(*elements*)
An alternate constructor for constant arrays.

- *elements* is a sequence of values, *Constant* or otherwise.
- All *elements* must have the same type.
- Returns a constant array containing the *elements*, in order.

classmethod literal_struct(*elements*)
An alternate constructor for constant structs.

- *elements* is a sequence of values, *Constant* or otherwise. Returns a constant struct containing the *elements* in order.
- **bitcast**(*typ*)
  Convert this pointer constant to a constant of the given pointer type.
- **gep**(*indices*)
  Compute the address of the inner element given by the sequence of *indices*. The constant must have a pointer type.
- **inttoptr**(*typ*)
  Convert this integer constant to a constant of the given pointer type.

NOTE: You cannot define constant functions. Use a *Function declaration* instead.

class llvmlite.ir.**Argument**
One of a function's arguments. Arguments have the *add_attribute()* method.

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

class llvmlite.ir.**Block**
A *Basic block*. Do not 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 in which the basic block is defined.
- **basic_block**
  The basic block. Must be a part of *function*.

Metadata

There are several kinds of *Metadata* values.

class llvmlite.ir.**MetaDataString**(*module*, *value*)
A string literal for use in metadata.

- *module* is the module that 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.**DIValue**
A debug information descriptor, containing key-value pairs. To create an instance, call *Module.add_debug_info()*.

class llvmlite.ir.**DIToken**(*value*)
A debug information "token," representing a well-known enumeration value. *value* is the enumeration name.

EXAMPLE: `'DW_LANG_Python'`

class llvmlite.ir.**NamedMetaData**
A named metadata node. To create an instance, call *Module.add_named_metadata()*. *NamedMetaData* has the *add()* 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*.

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 attributes:

- **linkage**
  A Python string describing the linkage behavior of the global value—for example, whether it is visible from other modules. The default is the empty string, meaning "external."
- **storage_class**
  A Python string describing the storage class of the global value.
  - The default is the empty string, meaning "default."
  - Other possible values include `dllimport` and `dllexport`.

class `llvmlite.ir.`**GlobalVariable**(*module*, *typ*, *name*, *addrspace=0*)
A global variable.

- *module* is where the variable is defined.
- *typ* is the variable's type. It cannot be a function type. To declare a global function, use *Function*.

  The type of the returned Value is a pointer to *typ*. To read the contents of the variable, you need to *load()* from the returned Value. To write to the variable, you need to *store()* to the returned Value.
- *name* is the variable's name—a Python string.
- *addrspace* is an optional address space to store the variable in.

Global variables have the following writable attributes:

- **global_constant**
  - If `True`, the variable is declared a constant, meaning that its contents cannot be modified.
  - The default is `False`.
- **unnamed_addr**
  - If `True`, the address of the variable is deemed insignificant, meaning that it is merged with other variables that have the same initializer.
  - The default is `False`.
- **initializer**
  The variable's initialization value—probably a *Constant* of type *typ*. The default is `None`, meaning that the variable is uninitialized.
- **align**
  An explicit alignment in bytes. The default is `None`, meaning that the default alignment for the variable's type is used.

class `llvmlite.ir.`**Function**(*module*, *typ*, *name*)
A global function.

- *module* is where the function is 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 not empty, it names the block's entry *Label*.
  - Returns a new *Block*.
- **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.

- **set_metadata**(*name*, *node*)
  Add a function-specific metadata named *name* pointing to the given metadata *node*—an *MDValue*.
- **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. The default is the empty string.
- **is_declaration**
  Indicates whether the global function is a declaration or a definition.
  - If `True`, it is a declaration.
  - If `False`, it is a definition.

Instructions

Every *Instruction* is also a value:

- It has a name—the recipient's name.
- It has a well-defined type.
- It can be used as an operand in other instructions or in literals.

Usually, you should not instantiate instruction types directly. 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 *name* pointing to the given metadata *node*—an *MDValue*.
- **function**
  The function that contains this instruction.
- **module**
  The module that defines this instruction's function.

class llvmlite.ir.**PredictableInstr**
The class of instructions for which we can specify the probabilities of different outcomes—for example, a switch or a conditional branch. Predictable instructions have the *set_weights()* 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 number, the likelier the outcome.

class llvmlite.ir.**SwitchInstr**
A switch instruction. Switch instructions have the *add_case()* 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 val and the switch operand compare equal.

class llvmlite.ir.**IndirectBranch**
An indirect branch instruction. Indirect branch instructions have the `add_destination()` 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 `add_incoming()` 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 `add_clause()` method:

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

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(), ...)*.

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 represented by the `Module` class.

class llvmlite.ir.**Module**(*name=''*)
Create a module. For informational purposes, you can specify the optional *name*, a Python string.

Modules have the following methods and attributes:

- **add_debug_info**(*kind*, *operands*, *is_distinct=False*)
  Add debug information metadata to the module with the given *operands*—a mapping of string keys to values—or return a previous equivalent metadata. *kind* is the name of the debug information kind.

  EXAMPLE: `'DICompileUnit'`

  A `DIValue` instance is returned. You can then associate it to, for example, an instruction.

  EXAMPLE:

```
di_file = module.add_debug_info("DIFile", {
     "filename": "factorial.py",
     "directory": "bar",
})
di_compile_unit = module.add_debug_info("DICompileUnit", {
     "language": ir.DIToken("DW_LANG_Python"),
     "file": di_file,
     "producer": "llvmlite x.y",
     "runtimeVersion": 2,
     "isOptimized": False,
}, is_distinct=True)
```

- **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. Returns an *MDValue*.
- **add_named_metadata**(*name*, *element=None*)
  Return the metadata node with the given *name*. If it does not already exist, the named metadata node is created first. If *element* is not `None`, it can be a metadata value or a sequence of values to append to the metadata node's elements. Returns a *NamedMetaData*.

  EXAMPLE:

  ```
  module.add_named_metadata("llvm.ident", ["llvmlite/1.0"])
  ```

- **get_global**(*name*)
  Get the *Global value*—a *GlobalValue*—with the given name. `KeyError` is raised if the name does not exist.
- **get_named_metadata**(*name*)
  Return the metadata node with the given *name*. `KeyError` is raised if the name does not exist.
- **get_unique_name**(*name*)
  Return a unique name across 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.

IR builders

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

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

A `IRBuilder` also maintains a reference to metadata describing the current source location, which is attached to all inserted instructions.

Instantiation

class llvmlite.ir.**IRBuilder**(*block=None*)
Create a new IR builder. If *block*—a `Block`—is given, the builder starts at the end of this basic block.

Attributes

`IRBuilder` has the following attributes:

- IRBuilder.**block**
  The basic block that the builder is operating on.
- IRBuilder.**function**
  The function that the builder is operating on.
- IRBuilder.**module**
  The module that the builder's function is defined in.

- IRBuilder.**debug_metadata**
  If not None, the metadata that is attached to any inserted instructions as !dbg, unless the instruction already has !dbg set.

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.

Positioning

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

- IRBuilder.**position_before**(*instruction*)
  Position immediately before the given *instruction*. The current block is also changed to the instruction's basic block.
- IRBuilder.**position_after**(*instruction*)
  Position immediately 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 and then go back to where you were.

- IRBuilder.**goto_block**(*block*)
  Position 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**()
  The same as *goto_block()*, but with the current function's entry block.

Flow control helpers

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

- IRBuilder.**if_then**(*pred*, *likely=None*)
  Create 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 accordingly.

- `IRBuilder.`**`if_else`**(*pred*, *likely=None*)
  Set up 2 basic blocks whose execution is conditioned on predicate *pred*, a value of type `IntType(1)`. *likely* has the same meaning as in `if_then()`.

  A pair of context managers is yielded. Each of them acts as an *`if_then()`* context manager—the first for the block to be executed if pred is `True` and the second 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 that 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
```

Instruction building

The following methods insert a new instruction—an *`Instruction`* instance—at the current index in the current block. The new instruction is returned.

An instruction's operands are almost 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

In the methods below, the *flags* argument is an optional sequence of strings that modify the instruction's semantics. Examples include the fast-math flags for floating-point operations, and whether wraparound on overflow can be ignored on integer operations.

Integer

- `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.`**`sadd_with_overflow`**(*lhs*, *rhs*, *name=''*, *flags=()*)
  Integer add *lhs* and *rhs*. A `{ result, overflow bit }` structure is returned.
- `IRBuilder.`**`sub`**(*lhs*, *rhs*, *name=''*, *flags=()*)
  Integer subtract *rhs* from *lhs*.

- IRBuilder.**ssub_with_overflow**(*lhs*, *rhs*, *name=''*, *flags=()*)
  Integer subtract *rhs* from *lhs*. A `{ result, overflow bit }` structure is returned.
- IRBuilder.**mul**(*lhs*, *rhs*, *name=''*, *flags=()*)
  Integer multiply *lhs* with *rhs*.
- IRBuilder.**smul_with_overflow**(*lhs*, *rhs*, *name=''*, *flags=()*)
  Integer multiply *lhs* with *rhs*. A `{ result, overflow bit }` structure is returned.
- 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.**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.**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*.

Floating-point

- IRBuilder.**fadd**(*lhs*, *rhs*, *name=''*, *flags=()*)
  Floating-point add *lhs* and *rhs*.
- IRBuilder.**fsub**(*lhs*, *rhs*, *name=''*, *flags=()*)
  Floating-point subtract *rhs* from *lhs*.
- IRBuilder.**fmul**(*lhs*, *rhs*, *name=''*, *flags=()*)
  Floating-point multiply *lhs* by *rhs*.
- IRBuilder.**fdiv**(*lhs*, *rhs*, *name=''*, *flags=()*)
  Floating-point divide *lhs* by *rhs*.
- IRBuilder.**frem**(*lhs*, *rhs*, *name=''*, *flags=()*)
  Floating-point remainder of *lhs* divided by *rhs*.

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*. The string *cmpop* can be one of <, <=, ==, !=, >= or >.
- `IRBuilder.`**`icmp_unsigned`**(*cmpop*, *lhs*, *rhs*, *name=''*)
  Unsigned integer compare *lhs* with *rhs*. The string *cmpop* can be one of <, <=, ==, !=, >= or >.
- `IRBuilder.`**`fcmp_ordered`**(*cmpop*, *lhs*, *rhs*, *name=''*, *flags=[]*)
  Floating-point ordered compare *lhs* with *rhs*.
  - The string *cmpop* can be one of <, <=, ==, !=, >=, >, `ord` or `uno`.
  - The *flags* list can include any of `nnan`, `ninf`, `nsz`, `arcp` and `fast`, which implies all previous flags.
- `IRBuilder.`**`fcmp_unordered`**(*cmpop*, *lhs*, *rhs*, *name=''*, *flags=[]*)
  Floating-point unordered compare *lhs* with *rhs*.
  - The string *cmpop*, can be one of <, <=, ==, !=, >=, >, `ord` or `uno`.
  - The *flags* list can include any of `nnan`, `ninf`, `nsz`, `arcp` and `fast`, which implies all previous flags.

Conditional move

`IRBuilder.`**`select`**(*cond*, *lhs*, *rhs*, *name=''*)
A 2-way select—*lhs* if *cond*, else *rhs*.

Phi

IRBuilder.**phi**(*typ*, *name=”*)
Create a phi node. To add incoming edges and their values, use 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.
  - Indices must be constant.
- IRBuilder.**insert_value**(*agg*, *value*, *index*, *name=”*)
  Build a copy of *aggregate value* *agg* by setting the new *value* at *index*. The value for *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 1 value is allocated.
- IRBuilder.**load**(*ptr*, *name=”*, *align=None*)
  Load value from pointer *ptr*. If *align* is passed, it should be a Python integer specifying the guaranteed pointer alignment.
- IRBuilder.**store**(*value*, *ptr*, *align=None*)
  Store *value* to pointer *ptr*. If *align* is passed, it should be a Python integer specifying the guaranteed pointer alignment.
- 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* is 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*.
  - The string *op* specifies the operation—for example, `add` or `sub`.
  - The optional *ordering* specifies the memory model for this instruction.

Function call

IRBuilder.**call**(*fn*, *args*, *name=”*, *cconv=None*, *tail=False*, *fastmath=()*)
Call function *fn* with arguments *args*, a sequence of values.

- *cconv* is the optional calling convention.
- *tail*, if `True`, is a hint for the optimizer to perform tail-call optimization.
- *fastmath* is a string or a sequence of strings of names for fast-math flags.

Branches

The following methods 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.

  To add non-default targets, use the `add_case()` method on the return value.
- `IRBuilder.`**`indirectbr`**`(`*address*`)`
  Jump to the basic block with the address *address*, a value of type *IntType(8).as_pointer()*.

  To obtain a block address, use the `BlockAddress` constant.

  To add all possible jump destinations, use 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.
  - *cconv* 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*, whose first non-phi instruction 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 2 pointer-sized fields.
  - *personality* specifies an exception personality function.
  - *cleanup* specifies whether control should always be transferred to this landing pad, even when no matching exception is caught.

  To add landing pad clauses, use the `add_clause()` method on the return value.

  There are 2 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 1 clause or be marked for cleanup.

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

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

Inline assembler

- `IRBuilder.`**`asm`**(*ftype*, *asm*, *constraint*, *args*, *side_effect*, *name=''*)
  Add an inline assembler call instruction. For example, this is used in *load_reg()* and *store_reg()*.

  Arguments:

  - *ftype* is a function type specifying the inputs and output of the inline assembler call.
  - *asm* is the inline assembler snippet—for example, `"mov $2, $0\nadd $1, $0"`. x86 inline ASM uses the AT&T syntax.
  - *constraint* defines the input/output constraints—for example `=r,r,r`.
  - *args* is the list of inputs, as IR values.
  - *side_effect* is a boolean that specifies whether or not this instruction has side effects not visible in the constraint list.
  - *name* is the optional name of the returned LLVM value.

  For more information about these parameters, see the official LLVM documentation.

  EXAMPLE: Adding 2 64-bit values on x86:

```
fty = FunctionType(IntType(64), [IntType(64),IntType(64)])
add = builder.asm(fty, "mov $2, $0\nadd $1, $0", "=r,r,r",
                  (arg_0, arg_1), name="asm_add")
```

- `IRBuilder.`**`load_reg`**(*reg_type*, *reg_name*, *name=''*)
  Load a register value into an LLVM value.

  EXAMPLE: Obtaining the value of the `rax` register:

```
builder.load_reg(IntType(64), "rax")
```

- `IRBuilder.`**`store_reg`**(*value*, *reg_type*, *reg_name*, *name=''*)
  Store an LLVM value inside a register.

  EXAMPLE: Storing `0xAAAAAAAAAAAAAAAA` into the `rax` register:

```
builder.store_reg(Constant(IntType(64), 0xAAAAAAAAAAAAAAAA), IntType(64), "rax
↪")
```

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.

Example—defining a simple function

This example defines a function that adds 2 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 *intermediate representation* is printed at the end:

```
; 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, see *LLVM binding layer—llvmlite.binding*.

3.2.2 LLVM binding layer—llvmlite.binding

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

Initialization and finalization

You only need to call these functions once per process invocation.

- llvmlite.binding.**initialize**()
  Initialize the LLVM core.
- llvmlite.binding.**initialize_all_targets**()
  Initialize all targets. Must be called before targets can be looked up via the *Target* class.
- llvmlite.binding.**initialize_all_asmprinters**()
  Initialize all code generators. Must be called before generating any assembly or machine code via the *TargetMachine.emit_object()* and *TargetMachine.emit_assembly()* methods.
- llvmlite.binding.**initialize_native_target**()
  Initialize the native—host—target. Must be called once before doing any code generation.
- llvmlite.binding.**initialize_native_asmprinter**()
  Initialize the native assembly printer.
- llvmlite.binding.**initialize_native_asmparser**()
  Initialize the native assembly parser. Must be called for inline assembly to work.
- llvmlite.binding.**shutdown**()
  Shut down the LLVM core.
- llvmlite.binding.**llvm_version_info**
  A 3-integer tuple representing the LLVM version number.

  EXAMPLE: `(3, 7, 1)`

  Since LLVM is statically linked into the `llvmlite` DLL, this is guaranteed to represent the true LLVM version in use.

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 *name*. An integer is returned, or `None` if the symbol is not found.
- llvmlite.binding.**load_library_permanently**(*filename*)
  Load an external shared library. *filename* is the path to the shared library file.

Target information

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

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

EXAMPLE:

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

Functions

- `llvmlite.binding.`**`get_default_triple`**`()`
  Return a string representing the default target triple that LLVM is configured to produce code for. This represents the host's architecture and platform.
- `llvmlite.binding.`**`get_process_triple`**`()`
  Return a target triple suitable for generating code for the current process.

  EXAMPLE: The default triple from `get_default_triple()` is not suitable when LLVM is compiled for 32-bit, but the process is executing in 64-bit mode.
- `llvmlite.binding.`**`get_object_format`**`(`*triple=None*`)`
  Get the object format for the given *triple* string, or the default triple if `None`. Returns a string such as `"ELF"`, `"COFF"` or `"MachO"`.
- `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.`**`get_host_cpu_features`**`()`
  Return a dictionary-like object indicating the CPU features for the current architecture and whether they are enabled for this CPU.

  The key-value pairs contain the feature name as a string and a boolean indicating whether the feature is available.

  The returned value is an instance of the `FeatureMap` class, which adds a new method `.flatten()` for returning a string suitable for use as the `features` argument to *`Target.create_target_machine()`*.

  If LLVM has not implemented this feature or it fails to get the information, a `RuntimeError` exception is raised.
- `llvmlite.binding.`**`create_target_data`**`(`*data_layout*`)`
  Create a *`TargetData`* representing the given *data_layout* string.

Classes

`class` `llvmlite.binding.`**`TargetData`**
Provides functionality around a given data layout. It specifies how the different types are to be represented in memory. Use *`create_target_data()`* to instantiate.

- **`get_abi_size`**`(`*type*`)`
  Get the ABI-mandated size of a *`TypeRef`* object. Returns an integer.
- **`get_pointee_abi_size`**`(`*type*`)`
  Similar to *`get_abi_size()`*, but assumes that *type* is an LLVM pointer type and returns the ABI-mandated size of the type pointed to. This is useful for a global variable, whose type is really a pointer to the declared type.

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

- **get_element_offset**(*type*, *position*)
  Computes the byte offset of the struct element at position.

class llvmlite.binding.**Target**
Represents 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 that LLVM is configured to produce code for. This is equivalent to calling `Target.from_triple(get_default_triple())`.

The following attributes and methods are available:

- **description**
  A description of the target.

- **name**
  The name of the target.

- **triple**
  A string that uniquely identifies the target.

  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, run the command `llc -mcpu=help`.

class llvmlite.binding.**TargetMachine**
Holds 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 that is suitable for use with the platform's linker. Returns a bytestring.

- **set_asm_verbosity**(*is_verbose*)
  Set whether this target machine emits assembly with human-readable comments, such as those describing control flow or debug information.

- **emit_assembly**(*module*)
  Return a string representing the compiled *module*'s native assembler. You must first call *initialize_native_asmprinter()*.

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

class llvmlite.binding.**FeatureMap**
Stores processor feature information in a dictionary-like object. This class extends `dict` and adds only the `.flatten()` method.

flatten(*sort=True*)
Returns a string representation of the stored information that is suitable for use in the `features` argument of *`Target.create_target_machine()`*.

If the `sort` keyword argument is `True`—the default—the features are sorted by name to give a stable ordering between Python sessions.

Context

LLVMContext is an opaque context reference used to group modules into logical groups. For example, the type names are unique within a context, the name collisions are resolved by LLVM automatically.

LLVMContextRef

A wrapper around LLVMContext. Should not be instantiated directly, use the following methods:

class LLVMContextRef

- **create_context():**
  Create a new LLVMContext instance.
- **get_global_context():**
  Get the reference to the global context.

Modules

Although they conceptually represent the same thing, modules in the *IR layer* and modules in the *binding layer* do not have the same roles and do not expose the same API.

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

To go from the IR layer to the binding layer, use the *`parse_assembly()`* function.

Factory functions

You can create a module from the following factory functions:

- llvmlite.binding.**parse_assembly**(*llvmir*, *context=None*)
  Parse the given *llvmir*, a string containing some LLVM IR code. If parsing is successful, a new *`ModuleRef`* instance is returned.
  - context: an instance of *`LLVMContextRef`*.

    Defaults to the global context.

  EXAMPLE: You can obtain *llvmir* by calling `str()` on an *`llvmlite.ir.Module`* object.
- llvmlite.binding.**parse_bitcode**(*bitcode*, *context=None*)
  Parse the given *bitcode*, a bytestring containing the LLVM bitcode of a module. If parsing is successful, a new *`ModuleRef`* instance is returned.

- context: an instance of *LLVMContextRef*.

  Defaults to the global context.

EXAMPLE: You can obtain the *bitcode* by calling *ModuleRef.as_bitcode()*.

The ModuleRef class

class llvmlite.binding.**ModuleRef**

A wrapper around an 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.
- **get_struct_type**(*name*)
  Get the struct type with the given *name* in this module.

  If found, a *TypeRef* 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.
- **verify**()
  Verify the module's correctness. On error, raise RuntimeError.
- **data_layout**
  The data layout string for this module. This attribute can be set.
- **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.
- **struct_types**
  An iterator over the struct types defined in this module. Each type is a *TypeRef* instance.
- **name**
  The module's identifier, as a string. This attribute can be set.
- **triple**
  The platform "triple" string for this module. This attribute can be set.

Value references

A value reference is a wrapper around an LLVM value for you to inspect. You cannot create a value reference yourself. You get them from methods of the *ModuleRef* class.

Enumerations

class llvmlite.binding.**Linkage**
The linkage types allowed for global values are:

- **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**

class llvmlite.binding.**Visibility**
The visibility styles allowed for global values are:

- **default**
- **hidden**
- **protected**

class llvmlite.binding.**StorageClass**
The storage classes allowed for global values are:

- **default**
- **dllimport**
- **dllexport**

The ValueRef class

class llvmlite.binding.**ValueRef**

A wrapper around an LLVM value. The attributes available are:

- **is_declaration**
  - True—The global value is a mere declaration.
  - False—The global value is defined in the given module.
- **linkage**
  The linkage type—a *Linkage* instance—for this value. This attribute can be set.
- **module**
  The module—a *ModuleRef* instance—that this value is defined in.
- **name**
  This value's name, as a string. This attribute can be set.
- **type**
  This value's LLVM type as *TypeRef* object.
- **storage_class**
  The storage class—a *StorageClass* instance—for this value. This attribute can be set.
- **visibility**
  The visibility style—a *Visibility* instance—for this value. This attribute can be set.

Type references

A type reference wraps an LLVM type. It allows accessing type's name and IR representation. It is also accepted by methods like *TargetData.get_abi_size()*.

The TypeRef class

class llvmlite.binding.**TypeRef**

A wrapper around an LLVM type. The attributes available are:

- llvmlite.binding.**name**
  This type's name, as a string.
- llvmlite.binding.**is_pointer**
  - True—The type is a pointer type
  - False—The type is not a pointer type
- llvmlite.binding.**element_type**
  If the type is a pointer, return the pointed-to type. Raises a ValueError if the type is not a pointer type.
- llvmlite.binding.**__str__**(*self*)
  Get the string IR representation of the type.

Execution engine

The execution engine is where actual code generation and execution happen. The currently supported LLVM version—LLVM 3.8—exposes a single execution engine, named MCJIT.

Functions

- `llvmlite.binding.`**`create_mcjit_compiler`**(*module*, *target_machine*)
  Create a MCJIT-powered engine from the given *module* and *target_machine*.
  - *module* does not need to contain any code.
  - Returns a *ExecutionEngine* instance.
- `llvmlite.binding.`**`check_jit_execution`**()
  Ensure that 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.

The ExecutionEngine class

class `llvmlite.binding.`**ExecutionEngine**
A wrapper around an 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_function_address**(*name*)
  Return the address of the function *name* as an integer. It's a fatal error in LLVM if the symbol of *name* doesn't exist.
- **get_global_value_address**(*name*)
  Return the address of the global value *name* as an integer. It's a fatal error in LLVM if the symbol of *name* doesn't exist.
- **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 the `(module, buffer)` arguments:
    * *module* is a *ModuleRef* instance.
    * *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 an argument, *module*, a *ModuleRef* instance of the module being compiled.
    * It can return `None`, in which case the module is compiled normally.
    * It can return a bytes object of native code for the module, which bypasses compilation entirely.
- **target_data**
  The *TargetData* used by the execution engine.

Optimization passes

LLVM gives you the opportunity to fine-tune optimization passes. Optimization passes are managed by a pass manager. There are 2 kinds of pass managers:

- *FunctionPassManager*, for optimizations that work on single functions.
- *ModulePassManager*, for optimizations that work on whole modules.

To instantiate either of these pass managers, you first need to create and configure a *PassManagerBuilder*.

class llvmlite.binding.**PassManagerBuilder**

Create a new pass manager builder. This object centralizes optimization settings.

The `populate` method is available:

populate(*pm*)

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

The following writable attributes are available:

- **disable_unroll_loops**
  If `True`, disable loop unrolling.
- **inlining_threshold**
  The integer threshold for inlining one function into another. The higher the number, the more likely that inlining will occur. 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, as an integer between 0 and 2.
- **slp_vectorize**
  If `True`, enable the SLP vectorizer, which uses a different algorithm than the loop vectorizer. Both may be enabled at the same time.

class llvmlite.binding.**PassManager**

The base class for pass managers. Use individual `add_*` methods or *PassManagerBuilder.populate()* to add optimization passes.

- **add_constant_merge_pass**()
  See constmerge pass documentation.
- **add_dead_arg_elimination_pass**()
  See deadargelim pass documentation.
- **add_function_attrs_pass**()
  See functionattrs pass documentation.
- **add_function_inlining_pass**(*self*)
  See inline pass documentation.
- **add_global_dce_pass**()
  See globaldce pass documentation.
- **add_global_optimizer_pass**()
  See globalopt pass documentation.

- **add_ipsccp_pass**()
  See ipsccp pass documentation.
- **add_dead_code_elimination_pass**()
  See dce pass documentation.
- **add_cfg_simplification_pass**()
  See simplifycfg pass documentation.
- **add_gvn_pass**()
  See gvn pass documentation.
- **add_instruction_combining_pass**()
  See instcombine pass documentation.
- **add_licm_pass**()
  See licm pass documentation.
- **add_sccp_pass**()
  See sccp pass documentation.
- **add_sroa_pass**()
  See scalarrepl pass documentation.

  While the scalarrepl pass documentation describes the transformation performed by the pass added by this function, the pass corresponds to the `opt -sroa` command-line option and not to `opt -scalarrepl`.
- **add_type_based_alias_analysis_pass**()
  See tbaa metadata documentation.
- **add_basic_alias_analysis_pass**()
  See basicaa pass documentation.

class llvmlite.binding.**ModulePassManager**

Create a new pass manager to run optimization passes on a module.

The `run` method is available:

run(*module*)

Run optimization passes on the *module*, a *ModuleRef* instance.

Returns `True` if the optimizations made any modification to the module. Otherwise returns `False`.

class llvmlite.binding.**FunctionPassManager**(*module*)

Create a new pass manager to run optimization passes on a function of the given *module*, a *ModuleRef* instance.

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 *function*, a *ValueRef* instance.

  Returns `True` if the optimizations made any modification to the module. Otherwise returns `False`.

Analysis utilities

`llvmlite.binding.`**`get_function_cfg`**(*func*, *show_inst=True*)
Return a string of the control-flow graph of the function, in DOT format.

- If *func* is not a materialized function, the module containing the function is parsed to create an actual LLVM module.
- The *show_inst* flag controls whether the instructions of each block are `printed.functions`.

`llvmlite.binding.`**`view_dot_graph`**(*graph*, *filename=None*, *view=False*)
View the given DOT source. This function requires the graphviz package.

- If view is `True`, the image is rendered and displayed in the default application in the system. The file path of the output is returned.
- If view is `False`, a `graphviz.Source` object is returned.
- If view is `False` and the environment is in an IPython session, an IPython image object is returned and can be displayed inline in the notebook.

Example—compiling a simple function

Compile and execute the function defined in `ir-fpadd.py`. For more information on `ir-fpadd.py`, see *Example—Defining a simple 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
```

(continues on next page)

(continued from previous page)

```
    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()
    engine.run_static_constructors()
    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)
```

3.3 Contributing to llvmlite

llvmlite originated to fulfill the needs of the Numba project. It is maintained mostly by the Numba team. We tend to prioritize the needs and constraints of Numba over other conflicting desires.

We do welcome any contributions in the form of *bug reports* or *pull requests*.

3.3.1 Communication methods

Mailing list

Send email to the Numba users public mailing list at numba-users@anaconda.com. You are welcome to send any questions about contributing to llvmlite to this mailing list.

You can subscribe and read the archives on Google Groups. The Gmane mirror allows NNTP access.

Bug reports

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

- What you are trying to do.
- Your operating system.
- What version of llvmlite you are running.
- A description of the problem—for example, 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.

Pull requests

To contribute code:

1. Fork our Github repository.
2. Create a branch representing your work.
3. When your work is ready, submit it as a pull request from the Github interface.

3.3.2 Development rules

Coding conventions

- All Python code should follow PEP 8.
- Our C++ code does not have a well-defined coding style.
- Code and documentation should generally fit within 80 columns, for maximum readability with all existing tools, such as code review user interfaces.

Platform support

Llvmlite will be kept compatible with Python 2.7, 3.4 and later under at least Windows, macOS and Linux. It needs to be compatible only with the currently supported LLVM version—the 3.8 series.

We do not expect contributors to test their code on all platforms. Pull requests are automatically built and tested using Travis-CI, which addresses Linux compatibility. Other operating systems are tested on an internal continuous integration platform at Anaconda®.

3.3.3 Documentation

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

1. Edit the source files under `docs/source/`.
2. Build the documentation:

```
make html
```

3. Check the documentation:

```
open _build/html/index.html
```

3.4 Release Notes

3.4.1 v0.26.0

The primary new features in this release is support for generation of Intel JIT events, which makes profiling of JIT compiled code in Intel VTune possible. This release also contains some minor build improvements for ARMv7, and some small fixes.

LLVM 7 support was originally slated for this release, but had to be delayed after some issues arose in testing. LLVM 6 is still required for llvmlite.

- PR #409: Use native cmake on armv7l
- PR #407: Throttle thread count for llvm build on armv7l.
- PR #403: Add shutdown detection to ObjectRef __del__ method.
- PR #400: conda recipe: add make as build dep
- PR #399: Add get_element_offset to TargetData
- PR #398: Fix gep method call on Constant objects
- PR #395: Fix typo in irbuilder documentation
- PR #394: Enable IntelJIT events for LLVM for VTune support

3.4.2 v0.25.0

This release adds support for the FMA instruction, and has some documentation and build improvements. Starting with this release, we are including ARMv8 (AArch64) testig in our CI process.

- PR #391: Fix broken win32 py2.7 build.
- PR #387: protect against empty features in list
- PR #384: Read CMAKE_GENERATOR which conda-build sets
- PR #382: rewrite of install instructions, calling out LLVM build challenges
- PR #380: Add FMA intrinsic support
- PR #379: ARM aarch64 test on jetson tx2
- PR #378: add slack, drop flowdock

3.4.3 v0.24.0

This release adds support for Python 3.7 and fixes some build issues. It also contains an updated SVML patch for the llvmdev package that works around some vectorization issues. It also adds a selective LLVM 6.0.1 llvmdev build for the *ppc64le* architecture.

- PR #374: Fix up broken patch selector
- PR #373: Add long description from readme
- PR #371: LLVM 6.0.1 build based on RC and fixes for PPC64LE
- PR #369: Recipe fixes for Conda Build 3
- PR #363: Workaround for incorrect vectorization factor computed for SVML functions
- PR #356: fix build on OpenBSD.
- PR #351: Python 3.7 compat: Properly escape repl in re.sub

3.4.4 v0.23.2

This is a bug fix release to assist in addressing a critical Numba issue that can affect users who download llvmlite packages from sources other than PyPI (pip), Anaconda, or Intel Python: https://github.com/numba/numba/issues/3006

Support for SVML is now detected at compile time and baked into a function that is exposed by llvmlite. This function can be queried at runtime to find out if SVML is supported by the LLVM that llvmlite was compiled against, code generation paths can then be adjusted accordingly.

The following PRs are closed in this release:

- PR #361: Add SVML detection and a function to declare support.

3.4.5 v0.23.1

This is a minor patch release that includes no code changes. It is issued to fix a couple of problems with the build recipes for llvmdev (on which llvmlite relies).

The following PRs are closed in this release:

- PR #353: PR Fix llvmdev build recipe.
- PR #348: llvmdev: enhancements to conda recipe

3.4.6 v0.23.0

In this release, we upgrade to LLVM 6. Two LLVM patches are added:

1. A patch to fix LLVM bug (https://bugs.llvm.org/show_bug.cgi?id=37019) that causes undefined behavior during CFG printing.
2. A patch to enable Intel SVML auto-vectorization of transcendentals.

The following PRs are closed in this release:

- PR #343: Fix undefined behavior bug due to Twine usage in LLVM
- PR #340: This moves llvmlite to use LLVM 6.0.0 as its backend.
- PR #339: Add cttz & ctlz

- PR #338: Add 3 Bit Manipulation Intrinsics
- PR #330: Add support for LLVM fence instruction
- **PR #326: Enable Intel SVML-enabled auto-vectorization for all the** transcendentals

3.4.7 v0.22.0

In this release, we have changed the locking strategy that protects LLVM from race conditions. Before, the llvmlite user (like Numba) was responsible for this locking. Now, llvmlite imposes a global thread lock on all calls into the LLVM API. This should be significantly less error prone. Future llvmlite releases may manually exempt some functions from locking once they are determined to be thread-safe.

The following PRs are closed in this release:

- PR#318: Ensuring threadsafety in concurrent usage of LLVM C-API
- PR#221: Add all function/return value attributes from LLVM 3.9
- PR#304: Expose support for static constructors/destructors

3.4.8 v0.21.0

In this release, we upgrade to LLVM 5. Our build scripts now use conda-build 3. For our prebuilt binaries, GCC 7 toolchain is used on unix-like systems and the OSX minimum deployment target is 10.9.

The following PRs are closed in this release:

- PR #315: Updates for conda build 3.
- PR #307: Fixes for LLVM5.
- PR #306: Working towards LLVM 5.0 support.

3.4.9 v0.20.0

Beginning with this minor release, we support wheels for Linux, OSX and Windows. Pull requests related to enabling wheels are #294, #295, #296 and #297. There are also fixes to the documentation (#283 and #289).

3.4.10 v0.19.0

This is a minor release with the following fixes.

- PR #281, Issue #280: Fix GEP addrspace issue
- PR #279: Fix #274 addrspace in gep
- PR #278: add Readthedocs badge
- PR #275: Add variables to pass through when doing conda-build
- PR #273: Fix the behavior of module.get_global
- PR #272: cmpop contains comparison type, not lhs
- PR #268, Fix #267: Support packed struct

The following are CI build related changes:

- PR #277: Add pass through gcc flags for llvmdev

- PR #276: Remove jenkins build scripts

3.4.11 v0.18.0

This is a minor release that fixes several issues (#263, #262, #258, #237) with the wheel build. In addition, we have minor fixes for running on PPC64LE platforms (#261). And, we added CI testing against PyPy (#253).

3.4.12 v0.17.1

This is a bugfix release that addresses issue #258 that our LLVM binding shared library is missing from the wheel builds.

3.4.13 v0.17.0

In this release, we are upgrading to LLVM 4.0. We are also starting to provide wheel packages for 64-bit Linux platforms (manylinux).

Fixes:

- Issue #249, PR #250: Disable static linking of libstdc++ by default.

Enhancements:

- PR #246: Add requirements.txt for pip dependency resolving
- PR #238: LLVM 4.0
- PR #222: Enable wheel builds

3.4.14 v0.16.0

API changes:

- Switched from LLVM 3.8 to 3.9
- `TargetData.add_pass` is removed in LLVM 3.9.

Enhancements:

- PR #239: Enable fastmath flags
- PR #233: Updates for llvm3.9.1
- PR #199: Update for changes in LLVM 3.9

Fixes:

- PR #236: Fix metadata with long value
- PR #231: Fix setup.py for Python2.7 so that pip auto installs dependencies
- PR #226: Fix get_host_cpu_features() so that it fails properly

3.4.15 v0.15.0

Enhancements:

- PR #213: Add partial LLVM bindings for ObjectFile.
- PR #215: Add inline assembly helpers in the builder.
- PR #216: Allow specifying alignment in alloca instructions.
- PR #219: Remove unnecessary verify in module linkage.

Fixes:

- PR #209, Issue #208: Fix overly restrictive test for library filenames.

3.4.16 v0.14.0

Enhancements:

- PR #104: Add binding to get and view function control-flow graph.
- PR #210: Improve llvmdev recipe.
- PR #212: Add initializer for the native assembly parser.

3.4.17 v0.13.0

Enhancements:

- PR #176: Switch from LLVM 3.7 to LLVM 3.8.
- PR #191: Allow setting the alignment of a global variable.
- PR #198: Add missing function attributes.
- PR #160: Escape the contents of metadata strings, to allow embedding any characters.
- PR #162: Add support for creating debug information nodes.
- PR #200: Improve the usability of metadata emission APIs.
- PR #200: Allow calling functions with metadata arguments (such as `@llvm.dbg.declare`).

Fixes:

- PR #190: Suppress optimization remarks printed out in some cases by LLVM.
- PR #200: Allow attaching metadata to a `ret` instruction.

3.4.18 v0.12.1

New release to fix packages on PyPI. Same as v0.12.0.

3.4.19 v0.12.0

Enhancements:

- PR #179: Let llvmlite build on armv7l Linux.
- PR #161: Allow adding metadata to functions.

- PR #163: Allow emitting fast-math `fcmp` instructions.
- PR #159: Allow emitting verbose assembly in TargetMachine.

Fixes:

- Issue #177: Make setup.py compatible with `pip install`.

3.4.20 v0.11.0

Enhancements:

- PR #175: Check LLVM version at build time
- PR #169: Default initializer for non-external global variable
- PR #168: add ir.Constant.literal_array()

3.4.21 v0.10.0

Enhancements:

- PR #146: Improve `setup.py clean` to wipe more leftovers.
- PR #135: Remove some llvmpy compatibility APIs.
- PR #151: Always copy TargetData when adding to a pass manager.
- PR #148: Make errors more explicit on loading the binding DLL.
- PR #144: Allow overriding `-flto` in Linux builds.
- PR #136: Remove Python 2.6 and 3.3 compatibility.
- Issue #131: Allow easier creation of constants by making type instances callable.
- Issue #130: The test suite now ensures the runtime DLL dependencies are within a certain expected set.
- Issue #121: Simplify build process on Unix and remove hardcoded linking with LLVMOProfileJIT.
- Issue #125: Speed up formatting of raw array constants.

Fixes:

- PR #155: Properly emit IR for metadata null.
- PR #153: Remove deprecated uses of `TargetMachine::getDataLayout()`.
- PR #156: Move personality from LandingPadInstr to FunctionAttributes. It was moved in LLVM 3.7.
- PR #149: Implement LLVM scoping correctly.
- PR #141: Ensure no CMakeCache.txt file is included in sdist.
- PR #132: Correct constant in `llvmir.py` example.

3.4.22 v0.9.0

Enhancements:

- PR #73: Add get_process_triple() and get_host_cpu_features()
- Switch from LLVM 3.6 to LLVM 3.7. The generated IR for some memory operations has changed.

- Improved performance of IR serialization.
- Issue #116: improve error message when the operands of a binop have differing types.
- PR #113: Let Type.get_abi_{size,alignment} not choke on identified types.
- PR #112: Support non-alphanumeric characters in type names.

Fixes:

- Remove the libcurses dependency on Linux.

3.4.23 v0.8.0

- Update LLVM to 3.6.2
- Add an *align* parameter to IRBuilder.load() and IRBuilder.store().
- Allow setting visibility, DLL storageclass of ValueRef
- Support profiling with OProfile

3.4.24 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.

3.4.25 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.

3.4.26 v0.5.1

Fixes:

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

3.4.27 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()

3.4.28 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.

3.4.29 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

3.4.30 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.

3.4.31 v0.1.0

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

3.5 Glossary

3.5.1 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.

3.5.2 Function declaration

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

3.5.3 Function definition

A function's prototype, as in a *Function declaration*, plus a body implementing the function.

3.5.4 getelementptr

An LLVM *Instruction* that lets you get the address of a subelement of an aggregate data structure.

See 'getelementptr' Instruction in the official LLVM documentation.

3.5.5 Global value

A named value accessible to all members of a module.

3.5.6 Global variable

A variable whose value is accessible to all members of a module. It is a constant pointer to a module-allocated slot of the given type.

All global variables are global values. However, the opposite is not true—function declarations and function definitions are not global variables, they are only *global values*.

3.5.7 Instruction

The fundamental element used in implementing an LLVM function. LLVM instructions define a procedural, assembly-like language.

3.5.8 Intermediate representation (IR)

High-level assembly-language code describing to LLVM the program to be compiled to native code.

3.5.9 Label

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

3.5.10 Metadata

Optional information associated with LLVM instructions, functions and other code. Metadata provides information that is not critical to the compiling of an LLVM *intermediate representation*, such as the likelihood of a condition branch or the source code location corresponding to a given instruction.

3.5.11 Module

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

3.5.12 Terminator, terminator instruction

A kind of *Instruction* that explicitly transfers control to another part of the program instead of going to the next instruction after it is executed. Examples are branches and function returns.

Python Module Index

l

Index

Symbols

A

B

C

D

E

F

G

I

L

M

N

O

P

R

S

T

U

V

W

X

Z