252-0210-00L Compiler Design HS2021

LLVMlite documentation

LLVMlite documentation

Syntax Semantics Provided Code

Overview

LLVMlite is a small subset of the LLVM IR that we will be using throughout the course as the intermediate representation in our compiler. Conceptually, it is either an abstract assembly-like language or an even lower-level C-like language that is convenient to manipulate programatically.

LLVMlite's features include:

- A C-like "weak type system" to statically rule out some malformed programs. More on this later.
- A variety of different kinds of integer values, pointers, function pointers, and structured data including strings, arrays, and structs.
- Top-level mutually-recursive function definitions and function calls as primitives.
- An infinite number of "locals" (also known as "pseudoregisters", "SSA variables", or "temporaries") to hold intermediate results of computations.
- An abstract memory model that doesn't constrain the layout of data in memory.
- Dynamically allocated memory associated with a function invocation (in C, the stack).
- Static and dynamically (heap) allocated structured data.
- A control-flow graph representation of function bodies.

This document explains the structure of well-formed LLVMlite programs, the semantics of LLVMlite in terms of an abstract machine, and the relevant parts of the code provided with the assignments. A description of the full LLVM intermediate representation can be found in the LLVM Language Reference.

Structure and Well-Formedness of Programs

To give you a sense of structure of LLVMlite programs and the most basic features, the following is our running example, the simple recursive factorial function written in the concrete syntax of the LLVMlite IR.

```
define i64 @fac(i64 %n) {              ; (1)
  %1 = icmp sle i64 %n, 0              ; (2)
  br i1 %1, label %ret, label %rec     ; (3)
ret:                                   ; (4)
  ret i64 1
rec:                                   ; (5)
  %2 = sub i64 %n, 1                   ; (6)
  %3 = call i64 @fac(i64 %2)           ; (7)
  %4 = mul i64 %n, %3
  ret i64 %4                           ; (8)
}

define i64 @main() {                   ; (9)
  %1 = call i64 @fac(i64 6)
  ret i64 %1
}
```

First, notice the function definition at (1). The `i64` annotations declare the return type and the type of the argument `n`. The argument is prefixed with "`%`" to indicate that it's an identifier local to the function, while `fac` is prefixed with " `@`" to indicate that it is global (i.e. in scope in the entire compilation unit).

Next, at (2) we have the first instruction of the body of `fac`, which performs a signed comparison of the argument `%n` and 0 and assigns the result to the temporary `%1`. The instruction at (3) is a "terminator", and marks the end of the current block. It will transfer control to either `ret at (4) or rec` at (5). The labels at (4) and (5) each indicate the beginning of a new block of instructions. Notice that the entry block starting at (2) is not labeled: in LLVM it is illegal to jump back to the entry block of a function body. Moving on, (6) performs a subtraction and names the result `%2. The i64` annotation indicates that both operands are 64-bit integers. The function `fac` is called at (7), and the result named `%3. Again, the i64` annotations indicate that the single argument and the returned value are 64-bit integers.

Finally, (8) returns from the function with the result named by `%4`, and (9) defines the main function of the program, which simply calls `fac` with a literal `i64` argument.

Program Structure

LLVMlite programs consist of three types of global definitions: function definitions, global data definitions, and named type definitions, which may be interleaved. These definitions are in scope for the entire compilation unit, may be mutually recursive, and need not be declared in order.

Types

Functions, global data definitions, and instruction are explicitly annotated with types. These are divided into "simple" types that may appear on the stack and as arguments to functions and "aggregate" types that may only appear in global and heap-allocated data. (Unlike full LLVM, LLVM lite does not allow locals to hold structured data.) There is also a "void" type that only appears in the return type of instructions and functions that do not return a value. This is essentially the ML unit type, but it has the additional restriction that it cannot appear as the type of an operand, so it is actually illegal to give it a name in the LLVM concrete syntax. In the following table we use `T` to range over simple and aggregate (non-void, non-function) types, `F to range over function types, and S` to range over simple types.

Concrete Syntax	Kind	Description
`void`	void	Indicates the instruction does not return a usable value.
`i1, i64`	simple	1-bit (boolean) and 64-bit integer values.
`T*`	simple	Pointer that can be dereferenced if its target is compatible with T
`i8*`	simple	Pointer to the first character in a null-terminated array of bytes. Note: `i8*` is a valid type, but just `i8` is not. LLVMlite programs do not operate over byte-sized integer values.
`F*`	simple	Function pointer
`S(S1, ..., SN)`	function	A function from S1, ..., SN to S
`void(S1, ..., SN)`	function	A function from S1, ..., SN to void
`{ T1, ..., TN }`	aggregate	Tuple of values of types T1, ..., TN
`[ N x T ]`	aggregate	Exactly N values of type T
`%NAME`	*	Abbreviation defined by a top-level named type definition

Named Types

Named type definitions

```
%IDENT = type T
```

define abbreviations for types in the scope of the entire compilation unit. The following specification assumes that these are replaced with their definitions whenever they are encountered. Note that recursive types, in which `T mentions %IDENT` are allowed, but for the type to be well formed, each such recursive occurrence must appear under a `*`. More generally, any collection of named types may be mutually recursive (i.e. the names may appear in the the definitions), but each cycle of such references must be broken by a `*`.

Global Definitions

The next kind of top-level definition is global data

```
@IDENT = global T G
```

where `G` ranges over global initializers, described in the following table, and `T` is the associated type. The global identifier `@IDENT, when used in the program, has type T*`. For example, the following program fragment has valid annotations:

```
     @foo = global i64 42
     @bar = global i64* @foo
     @baz = global i64** @bar
```

Concrete Syntax	**Type**	**Description**
`null`	`T*`	The null pointer constant.
`[0-9]+`	`i64`	64-bit integer literal.
`@IDENT`	`T*`	Global identifier. The type is always a pointer of the type associated with the global definition.
`c"[A-z]*\00"`	`[ N x i8 ]`	String literal. The size of the array N should be the length of the string in bytes, including the null terminator `\00`.
`[ T G1, ..., T GN ]`	`[ N x T ]`	Array literal.
`{ T1 G1, ..., TN GN }`	`{T1,...,TN}`	Struct literal.
`bitcast (T1* G1 to T2*)`	`T2*`	Bitcast.

Operands

We now turn to the parts of a function declaration. Each instruction in a function has zero or more operands which for the purposes of determining the well-formedness of programs, are restricted to the following types.

Concrete Syntax	**Type**	**Description**
`null`	`T*`	The null pointer constant
`[0-9]+`	`i64`	64-bit integer literal
`@IDENT`	`T*`	Global identifier. The type can always be determined from the global definitions and is always a pointer
`%IDENT`	`S`	Local identifier: can only name values of simple type. The type determined by an local definition of `%IDENT` in scope

Instructions and Terminators

The following table describes the restrictions on the types that may appear as parameters of well-formed instructions, and the constraints on the operands and result of the instruction for the purposes of type-checking. We assume that named types have been replace by their definitions.

For example, in the `call` instruction, each type parameter `S1, ..., SN` must be a simple type. When we type check a program containing this instruction, we must make sure that the operand `OP1 has exactly the function pointer type S1(S2, ..., SN)*`, and that the remaining operands `OP2, ..., OPN` have types `S2, ..., SN`.

Concrete Syntax	**Operand → Result Types**
`%L = BOP i64 OP1, OP2`	`i64 x i64 → i64`
`%L = alloca S`	`- → S*`
`%L = load S* OP`	`S* → S`
`store S OP1, S* OP2`	`S x S* → void`
`%L = icmp CND S OP1, OP2`	`S x S → i1`
`%L = call S1 OP1(S2 OP2, ..., SN OPN)`	`S1(S2, ..., SN)* x S2 x ... x SN → S1`
`call void OP1(S2 OP2, ... ,SN OPN)`	`void(S2, ..., SN)* x S2 x ... x SN → void`
`%L = getelementptr T1* OP1, i32 OP2, ..., i32 OPN`	`T1* x i64 x ... x i64 -> ` **GEPTY**`(T1, OP1, ..., OPN)*`

`%L = bitcast T1* OP to T2*`	`T1* → T2*`

Getelementptr Well-Formedness and Result Type

The `getelementptr` instruction has some additional well-formedness requirements. Operands after the first must all be constants, unless they are used to index into an array. LLVM actually requires the operands used to index into structs to be 32-bit integers. Rather than introducing 32-bit integers into our language, we will use our 64-bit constants and operands and assume the arguments of `getelementptr` always fall in the range [0, Int32.max_int].

In the table above, the result type of a `getelementptr` instruction described using the **GEPTY** function, which is defined in pseudocode as follows:

```
     GEPTY : T -> operand list -> T
     GEPTY : T operand::path' = GEPTY' T path'

     GEPTY' : T -> operand list -> T
     GEPTY' T                             [] = T
     GEPTY' { T1, ..., TN } (Const m)::path' = GEPTY' TM path' when m <= N
     GEPTY' [ _ x T ]         operand::path' = GEPTY'  T path'
```

Notice that `GEPTY` is a partial function. When `GEPTY` is not defined, the corresponding instruction is malformed. This happens when, for example:

- The list of index operands provided is empty
- An operand used to index a struct is not a constant
- The type is not an aggregate and the list of indices is not empty

Also notice that a GEP instruction that indexes beyond the size of an array is well-formed. The length information on array tags is only present to help the compiler lay out data in memory and is not verified statically.

Blocks, CFGs, and Function Definitions

A block (or "basic" block) is just a sequence of instructions followed by a terminator:

Concrete Syntax	**Operand → Result Types**
`ret void`	`– → –`
`ret S OP`	`S → –`
`br label %LAB`	`– → –`
`br i1 OP, label %LAB1, label %LAB2`	`i1 → –`

The body of a function is represented by a control flow graph (CFG). A CFG consists of a distinguished entry block and a sequence blocks of prefixed with a label `LAB:`. A function definition has a return type, the function name, a list of formal parameters and their types, and the body of the function. The full syntax of a function definition is then:

```
define [S|void] @IDENT(S1 OP, ... , SN OP) { BLOCK (LAB: BLOCK)...}
```

Like global data definitions, the type of the defined global identifier `@IDENT is S(S1, ... , SN)*` or `void(S1, ... , SN)*`, a function pointer.

There are some additional global well-formedness requirements for function definitions. Each label and local definition must be unique. In this way, a local identifier both names the result of an instruction and serves to identify the instruction within a function body. For the locals in a CFG to be well scoped, there must never be a path to a use of a local that does not pass through its definition. We will not go into the details here.

Abstract Machine

Like for X86lite, we define the semantics of LLVMlite by describing the execution of an abstract machine. One major difference between the LLVMlite machine and our x86 simulator is that we specify an explicit stack, heap, code, and global memory. While these structures were present in X86lite as conventions on how areas of memory and registers were used during program execution, legal programs were free to, for example, write over the return address on the stack and jump to arbitrary locations in memory. The definition of LLVMlite, on the other hand, enforces some of the abstractions present in C-like languages.

In general, a guiding principle behind LLVMlite and C-like languages is that the specification optimizes first for the ease of translation to assembly, as long as this does not constrain the underlying machine. This often leads to a more complicated semantics. We will provide the details here, but it is not necessary to understand them completely. For X86lite we gave an informal natural language specification and asked you to implement it. For LLVMlite, we will provide an interpreter that (assuming it is entirely free of bugs!) can serve as a formal definition of the language. This means that if you have some technical question about a detail of the semantics, you can simply run a program through the interpreter.

We will start with a high-level overview of the abstract machine, which should provide enough intuition for the assignments in the course. The details are presented in a later section.

Simple and Memory Values

The LLVMlite machine operates on dynamic values that, like the operand tags described in the previous section, include a subset of **simple values**. During program execution, operands can evaluate only to simple values and all other **memory values** must be manipulated indirectly through pointers. While this distinction makes the specification of LLVMlite more complicated, it results in a very straightforward compilation strategy: the simple values are those that can appear in X86lite registers.

Memory values will be represented as tree structures where the leaves are simple values (or strings) and finitely-branching nodes represent arrays and structs. The memory state of the LLVMlite machine is represented by a mapping between **block identifiers** and memory values. We will refer to a top-level memory value that is not a subtree of another as a **memory block**.

At this point, an illustration might be helpful:

```
    { bid0 -> node     bid1 -> node     bid2 -> node }
             /  |  \             |                |
            L   L   L          node               L
                              /    \
                             L      node
                                   / | | \
                                   L L L* L
```

The above diagram shows three memory values mapped to the block identifiers `bid0, bid1`, and `bid2`. One thing to notice is that every memory value contains at least one node. Even simple values, such as a single global i64 will be represented using a node with one leaf. The identifier `bid2` is an example of how non-aggregate data will be represented in memory, while `bid1` might be a structure having two fields. There's no deep reason for this, it's just a convenient invariant to represent the particular way LLVM computes pointers into structs.

In order to manipulate the simple values at the leaves of our memory blocks, we need specify a **path** to a leaf. For example, to uniquely identify the leaf marked * we might provide the **indices** `0,1,2` along with the block identifier `bid1`. This means that we're selecting the 2nd child of the 1st child of the 0th child of the root node.

This approach might seem a more complicated than our memory representation in X86lite, but it has some advantages as a specification. If instead, like in X86lite, we represented memory as an array of bytes, we would be forced us to make decisions about how large each value in the language is and the relative position of values in memory. Using an unordered set of trees for the language specification lets us define how operations on structured data work while leaving such details up to the compiler.

The simple values include:

- 1-bit (boolean) and 64-bit 2's complement signed integers
- Pointers to a subtree of a particular memory block containing a block identifier and path
- A special **undef** value that represents an unusable value

So, a real piece of LLVMlite memory might look like:

```
    { bid0 -> node       bid1 ->  node      bid2 -> node }
               |                    \                |
              node                   node           undef
            /  |  \                 /  |  \
       "foo" "bar" "baz"   (Ptr Null)  4  (Ptr bid0, 0, 1)
```

Here, the pointer in memory at `(bid1, 0, 2)` refers to the string "foo" in memory block `bid0`. In addition to the restriction that each memory block has a node, we require that every pointer have at least one index. In other words, it is not possible to refer directly to the top-level node of each memory block.

Machine Configurations

References to memory blocks will be split between three different address spaces: the **heap**, **stack**, and **globals**. Though all three are simply collections of memory values, they have different initialization and runtime behavior, as explained in the next section. We can think of machine configurations as having three separate memory components, mapping disjoint sets of identifiers to memory values.

In addition to memory, machine configurations need to keep track of the values assigned to temporaries by instructions. We will call this mapping of uids to simple values the **locals** of the machine state. Finally, the machine needs to keep track of the progress of execution of a function body. This is the **code** component of the state, and will consist of a currently executing block and the mapping from labels to blocks.

Machine Execution

We can describe how the machine executes a program by using two mutually-recursive functions, **interp_call** and **interp_cfg** that execute, respectively, an entire function call and the body of a function. This is a different approach than the one used in the X86lite machine, where we described a function that executes a single step of the machine, and then iterated the function until we reached a terminating state. The evaluation function approach used here will allow us to use properties of functions in the meta-language to avoid describing some details of the machine.

- **interp_call** takes the global identifier of a function in an LLVMlite program, a list of (simple) values to serve as arguments, and an initial memory state and returns the memory state after the function call has completed and the return value of the function.
  First, the machine looks up the function declaration associated with the global identifier. Then, it creates new `locals` that maps the formal parameters of the function declaration to the arguments supplied. It allocates a new frame by adding an empty memory block with a fresh frame identifier to the stack in the initial memory state. Finally, the machine evaluates the function body using **interp_cfg** and the cfg associated with the function identifier in the program text, and returns the result.

- **interp_cfg** does most of the work involved in evaluating an LLVMlite program. It takes a cfg, an initial locals map, and a memory state and evaluates the cfg, returning the resulting memory state and the return value of the function body.
  The LLVMlite machine examines the next instruction in the currently executing block, executes it, updating the locals, memory state, and currently executing block as necessary, and then calls itself again with the resulting configuration or returns. To interpret the call instruction, it uses the **interp_call** function above. A summary of the locals, cfg, and memory state passed to the next invocation is provided in the next section.

Instructions

Constant operands evaluate to the corresponding integer value, while global identifiers evaluate to a global pointer, and the Null constant evaluates to a pointer with the special null block identifier. Local ids are looked up in the locals map. In well-formed programs, execution will always pass through the definition of a local id before it is used as an operand.

Instructions are executed as follows:

Instruction/Terminator	**Behavior**
`%L = BOP i64 OP1, OP2`	Update locals(`%L`) with the result of the computation.
`%L = alloca S`	Allocate a slot in the current stack frame and return a pointer to it. This involves adding a subtree of undef to the root node of the memory block representing the frame at the next available index.
`%L = load S* OP`	OP must be a pointer or **undef**. Find the value referenced by the pointer in the current memory state. Update locals(`%L`) with the result. If OP is not a valid pointer, either because it evaluates to **undef**, no memory value is associated with its block identifier or its path does not identify a valid subtree, then the operation raises an error and the machine crashes. If the pointer is valid, but the value in memory is not a simple value of type S, the operation raises an error and the machine crashes.
`store S OP1, S* OP2`	Update the memory state by setting the target of OP2 to the value of OP1. If OP2 is not a valid pointer, or if the target of OP2 is not a simple value in memory of type S, the operation raises an error and the machine crashes.
`%L = icmp CND S OP1, OP2`	Update locals(`%L`) to 1 if the condition holds and 0 otherwise.
`%L = call S1 OP1(S2 OP2, ... ,SN OPN)`	Evaluate all of the operands and use them to recursively invoke the interpreter through **interp_call** with the current memory state. If OP1 does not evaluate to a function pointer that identifies a function with return type `S1` and argument types `S2, ... , SN`, then the operation raises an error and the machine crashes. Update the local (`%L`) to the result of **interp_call** and continue with the return memory state.
`call void OP1(S2 OP2, ... ,SN`	The same as a non-void call, but no locals are updated with the returned value.

`OPN)`	
`%L = getelementptr T1* OP1, i64 OP2, ... , i64 OPN`	Create a new pointer by *adding* the first index operand OP2 to the last index of the pointer value of OP1 and then *concatenating* the remaining indices onto the path. If the target of the resulting pointer is not a valid memory value *compatible* with the type %L, then update locals(`%L`) with the **undef** value. Otherwise, update locals(`%L`) with the new pointer. See the following section for a more detailed explanation.
`%L = bitcast T1* OP to T2*`	Update locals(`%L) with the value of OP`.
`ret void`	Pop the most recently allocated frame off the stack and return from **interp_cfg** with the **undef** value and the resulting memory state.
`ret S OP`	Pop the most recently allocated frame off the stack and return from **interp_cfg** with the value of OP and the resulting memory state.
`br label %LAB`	Look up the block associated with `%LAB` in the CFG set is as the current executing block.
`br i1 OP, label %LAB1, label %LAB2`	If `OP is 1, set the current block to %LAB1`, otherwise, set it to `%LAB2`.

Initial Configurations

Creating the initial machine state requires a few steps. First, global data declarations must be converted to a global memory state. This process is entirely straightforward and is implemented in the provided interpreter. Next, memory values for each string passed to the main function are added to an empty heap. Execution is started by invoking **interp_call** with the global identifier of the "main" function, passing in the number of arguments supplied and pointers to each string on the heap.

GEP Indexing

The semantics of GEP and exactly when the resulting pointer is valid is the most complicated part of LLVMlite. Here we walk though a slightly more complex example.

```
%t1 = type { A, B, C }
%t2 = type [ 2 x %t1 ]

@pn1 = global %t2 [ {a0, b0, c0}, {a1, b1, c1} ]

; Memory:
; { ... bid0 -> root ... }
;                 |
;                 n1
;               /    \
;             n2      n3
;           / | \    / | \
;          a0 b0 c0 a1 b1 c1

...
%pn2 = getelementptr %t2* pn1, i32 0, i32 0    ; %t1* -> n2
%pb1 = getelementptr %t1* pn2, i32 1, i32 1    ; B* -> b1
```

Suppose we start with the pointer pn1 = (bid0, 0) pointing to n1. The first GEP instruction above will compute the pointer (bid0, 0, 0), by first adding 0 to the last index of pn1 and then concatenating the rest of the indices to the end of the path. The next GEP instruction will compute the pointer (bid0, 0, 1, 1), which points to b1.

In LLVMlite, indexing into a sibling (rather than a child) of a node using GEP with a non-zero first index is only legal if sibling nodes are allocated as part of an array. In our example, n1 was allocated as `[ 2 x %t1 ]`, so this is the case. In addition to the restricted use of the first, the resulting path must target a subtree. If, instead, we tried to create a pointer off the end of the array, the resulting pointer would be undef. Similarly, if `B` is not an aggregate type and we computed a path into it, the result of the gep would be undef.

Lastly, since we do not preserve any metadata about the "shape" of memory values during compilation, we must check that there is enough information in the static annotation of the GEP instruction to actually compute the right index into memory. Since we can bitcast between any pointer values, there is no guarantee that the static annotation on a GEP instruction will match the target of the pointer value its operand will evaluate to at runtime.

For this, we have to define a notion of *compatible* LLVMlite types, which is defined in terms of a flattened version of our type annotations.

```
    FLATTEN : ty -> list ty
    FLATTEN i1          = [i1]
    FLATTEN i8          = [i8]
    FLATTEN i64         = [i64]
    FLATTEN Ptr t       = [Ptr I8]
    FLATTEN Array (n, t) = [Array (n, t)]
    FLATTEN Struct ts   = concat (map FLATTEN ts)

    PTRTOI8 : ty -> ty
    PTRTOI8 Ptr t       = Ptr I8
    PTRTOI8 Array (n, t) = Array (n, PTRTOI8 t)
    PTRTOI8 Struct ts   = Struct (map PTRTOI8 ts)
```

PTRTOI8 simply converts all pointers that appear in the type to Ptr I8. This is arbitrary, we just want all pointers to compare equal in the flattened type. FLATTEN then unnests all of the struct types that don't occur under an array constructor. A type t1 is compatible with t2 if FLATTEN (PTRTOI8 t1) is a list prefix of FLATTEN (PTRTOI8 t2). If, during execution, the annotation of a GEP instruction is not compatible with the actual type of the memory value that its operand was allocated with, the resulting pointer is undef.

Last modified: Monday, 12 October 2020, 12:20 AM

◀ X86lite documentation

Jump to...

The Often Misunderstood GEP Instruction ▶