CMSC 22600	**Compilers for Computer Languages**	**Handout 4**
Autumn 2020		**November 25, 2020**

LLVM Code Generation

1 Introduction

This document supplements the Project 4 description with additional information about generating and compiling LLVM assembly code.

LLVM is a large and complicated system for generating optimized machine code for a variety of target architectures. For this project, you will be using a very limited subset of its features and to ease its use, we have defined a simple interface to generating LLVM code, which you can find in the `llvm` directory of the sample code. Detailed information about the LLVM assembly language can be found in the *LLVM Language Reference*, which is available at `https://llvm.org/docs/LangRef.html`. While you do not need to read the whole document, which is quite long, it is a good reference if you are having trouble understanding how some LLVM instruction works.

2 The LLVM API

The `llvm` directory in the sample code contains a library to support the generation of LLVM assembly code. This section gives an overview of the main components of the library and how to use them, but you should look at the interfaces, which are well documented, to fully understand the library.

2.1 Names

LLVM uses a variety of different kinds of names for things. These include

- *Globals*, which are prefixed with a "`@`" symbol. These are used to name functions and string constants.
- *Registers*, which are prefixed with a "`%`" symbol, are *pseudo registers* and are used to hold local values. Since LLVM is an SSA representation, registers can only be assigned to once and ϕ instructions must be introduced to merge their values across multiple control paths.
- *Variables*, which are an abstraction of the various entities that can occur as an argument to a LLVM instruction. These include integer constants, globals, and registers. Variables roughly correspond to the values of the SimpleAST and CFG IRs.
- *Labels* are local to a function and are used to name the entry points of basic blocks.

2.2 Modules

An LLVM module is the container that holds the generated code. Its contents includes the functions that make up the program, external declarations for runtime-system functions, and string literals.

2.3 Functions

All executable code is contained in functions. Functions are named by globals and consist of one or more blocks. Function parameters are represented as LLVM registers, so the first step for generating code for a function is to map its parameters to registers.

2.4 Blocks

LLVM blocks are the containers for instructions and most of the operations for generating instructions require a block as an argument. An LLVM block has a unique label and starts with ϕ instructions followed by a sequence of register assignments. Blocks must terminated by either a branch, conditional branch or return instruction. Note that LLVM blocks are *basic blocks* and have no internal control flow. Thus, there is a one-to-one correspondence between the CFG fragments and the LLVM blocks of a function.

2.5 Types

LLVM has a type system that is somewhat similar to C's, with a few additional features. Like C, LLVM allows one to cast a value from one type to another. Mostly, you should not have to worry about casts; the one exception is discussed below in Section 4.4.

2.6 Instructions

Instructions are generated using the `LLVMBlock` module. The LLVM Library does not provide a public representation of instructions. Instead, the functions that generate instructions return a variable that can be supplied as an argument to some other instruction. For example, the following SML expression generates LLVM instructions for the expresion `(1 + 2) * 3` into the block `blk`:

```
LLVMBlock.emitMul (blk,
  LLVMBlock.emitAdd (blk, LLVMVar.i64const 1, LLVMVar.i64const 2),
  LLVMVar.i64const 3)
```

The generated LLVM assembly will be something like the following, but with different register names:

```
%r23 = add i64 1, 2
%r24 = mul i64 %r23, 3
```

One important form of LLVM instructions are the ϕ instructions that describe the incoming data flow at join points (*i.e.*, blocks with multiple predecessors). The LLVM library supports incremental definition of these instructions via an abstract type and two functions.

```
type phi
val emitPhi : t * LLVMReg.t -> phi
```

```
val addPhiDef : phi * LLVMVar.t * t -> unit
```

When setting up a block that is known to have multiple predecessors in the control-flow graph, we can allocate fresh ϕ instructions for each incoming live variable (*i.e.*, the parameters of the corresponding fragment) using the `emitPhi` function, where the first argument is the block, the second is the register assigned to the parameter, and the result is an abstract value representing the ϕ instruction. We can then add information about incoming data flow by calling the `addPhiDef` function on a ϕ instruction, where we specify the LLVM value flowing into the instruction and the block from whence it came.

2.7 Roadmap

The public interface to the LLVM library is organized into eight modules (listed in alphabetical order).

`structure LLVMBlock`
: Defines the representation of LLVM basic blocks. This module defines the functions for generating instructions.

`structure LLVMFunc`
: Defines the type and operations for LLVM functions.

`structure LLVMGlobal`
: Defines the type and operations for LLVM global names (*e.g.*, function names).

`structure LLVMLabel`
: Defines the type and operations for LLVM labels, which are used to name basic blocks.

`structure LLVMModule`
: Defines the type and operations for LLVM modules.

`structure LLVMReg`
: Defines the type and operations for LLVM pseudo registers.

`structure LLVMType`
: Defines the type and operations for LLVM types.

`structure LLVMVar`
: Defines the type and operations for LLVM variables, which describe the arguments and results of instructions.

3 Runtime Data Representations

As described in previous documents, **LangF** values are uniformly represented as 64-bit machine words. These words are either pointers to heap-allocated objects or else are tagged integers.

Because **LangF** is a polymorphic language, the type of variables in expressions is not always statically determined. Thus, we use a *uniform* representation of **LangF** values as a single 64-bit machine word. These words are either pointers to heap-allocated objects or integers. Since the

garbage collector must be able to distinguish between pointers and integers at runtime, we use a tagged representation for integers. Specifically, the integer n is represented as $[\![n]\!] = 2n + 1$. We discuss the implementation of arithmetic on tagged values below in Section 4.2.

The **LangF** runtime supports two kinds of heap objects: tuples and strings. Tuple objects are sequences of one or more **LangF** values (*i.e.*, 64-bit words). String objects have an initial length field, which is a word holding a tagged integer value specifying the number of characters in the string, followed by a sequence of bytes representing the string. Thus, the first character of a string is 8 bytes off of the base address of the string object.

4 Code generation

Since CFG is already a low-level IR, the translation to LLVM is mostly direct. CFG functions map to LLVM functions and fragments map to LLVM blocks. Your code generator will have to track the mapping of CFG variables to LLVM variables and the ϕ nodes associated with fragment parameters. In addition to converting to SSA and introducing ϕ instructions, the main complications are function calls, integer arithmetic, and booleans. We discuss each of these below, as well as other aspects of the translation.

4.1 Function calls

Function calls, which include both calls to **LangF** functions as well as calls to runtime-system functions (*e.g.*, for allocation) are the most complicated LLVM instruction to generate. The LLVM library supports two flavors of calls: tail calls and non-tail calls. Tail calls are fairly straight forward, since they do not involve any recording of garbage-collector meta data.

The non-tail-call case is more complicated. The garbage collector must examine the frames on the stack to find pointers into the heap. This activity requires generating *stack maps* that allow the collector to parse the stack. The LLVM library provides a function for generating function calls and the necessary stack-map information. The `emitCall` operation has the form

```
val {ret, live} = emitCall (blk, {func = f, args = vs, live = lvs})
```

where `blk` is the current LLVM block, `f` is a variable that names the function (it might be a global or a register), `vs` is the list of variables that hold the function arguments, and `lvs` is a list f LLVM variables that are live immediately after the function call. The result of `emitCall` is an optional variable representing the runtime result of the function call (`ret`) and a list of renamed live variables (`live`). Your code generator will have to update the bindings of the CFG variables that were in the live list to reflect this renaming. Note that for a CFG function call of the form

```
let x = f (y, z)
```

the live set will **not** include `x`, since it is killed by the let binding. The liveness information is used by LLVM to track potential GC roots in the stack. You will have to compute the liveness information as part of the `CodeGenInfo.analyze` function.

Calls that are compatible with garbage collection look ugly in LLVM, so the LLVM library hides the unimportant details. While debugging the output of your compiler, however, it is important to follow what is going on. Consider the following SML code fragment:

```
val res = emitCall (someBlk, {
        func = foo,
        args = [arg1, arg2],
        live = [val1, val2, val3]
      })
```

Where `val1` and `val3` are heap pointers. The LLVM code associated with this API call involves several LLVM *intrinsics*, which are functions used to access special features of the compiler. The simplified version of the code emitted by that example call follows.[1]

```
; performs the call to @foo, passing arg1 and arg2 as arguments.
%tok = call token @llvm.experimental.gc.statepoint(
    _, _,
    @foo,              ; function
    2,                 ; the arity of @foo
    _,
    arg1, arg2         ; list of args to @foo
    _, _,
    val1, val3         ; pointer values that are live
                       ; after the call. val2 was not a pointer.
  )

; retrieves the return value of the call to @foo
%retV = call @llvm.experimental.gc.result(token %tok)

; retrieves the (possibly updated) live heap pointers
%new.val1 = call @llvm.experimental.gc.result(token %tok)
%new.val3 = call @llvm.experimental.gc.relocate(token %tok, _, _)
```

It is important to note that after the call to `@foo` the live values `val1` and `val3` are considered updated, with the new values after the call being `new.val1` and `new.val3`, and you should not reuse `val1` and `val3`. This renaming will also affect the placement of phi nodes at the current join point. The example API call will return

```
{ ret = SOME retV, live = [val1', val2', val3']}
```

where the live-variable list contains LLVM variables

```
%new.val1, %val2, %new.val3
```

4.2 Integer Arithmetic

Recall from Section 3 that we represent the integer n as $[\![n]\!] = 2n + 1$. This representation must be accounted for when generating integer arithmetic instructions. For example, consider the addition of two tagged integers:

$$\begin{aligned} [\![n]\!] + [\![m]\!] &= (2n + 1) + (2m + 1) \\ &= 2(n + m) + 2 \\ &= [\![n + m]\!] + 1 \end{aligned}$$

[1] Underscores were added in places where the details are not important.

This reasoning shows that we can implement the tagged addition of two integers by adding their tagged representation and then subtracting one. Of course, if one of the integers is a constant, then we can subtract one from its literal representation at compile time. The following table shows how tagged versions of the various integer operations are implemented:

Addition	$[\![n + m]\!]$	$\Rightarrow$	$([\![n]\!] - 1) + [\![m]\!]$
Subtraction	$[\![n - m]\!]$	$\Rightarrow$	$[\![n]\!] - [\![m]\!] + 1$
Multiplication	$[\![n * m]\!]$	$\Rightarrow$	$([\![n]\!] - 1) * \lfloor [\![m]\!]/2 \rfloor + 1$
Division	$[\![n/m]\!]$	$\Rightarrow$	$2(\lfloor [\![n]\!]/2 \rfloor \ / \ \lfloor [\![m]\!]/2 \rfloor) + 1$
Remainder	$[\![n\%m]\!]$	$\Rightarrow$	$2(\lfloor [\![n]\!]/2 \rfloor \ \% \ \lfloor [\![m]\!]/2 \rfloor) + 1$

Note that the expression $\lfloor [\![m]\!]/2 \rfloor$ can be implemented using an arithmetic-right-shift instruction and multiplying by 2 can be implemented by shifting left by one bit. Integer comparison operations work correctly on the tagged representation of integers. The sample code includes a module `ArithGen` that generates tagged arithmetic operations.

4.3 String Operations

The two string primitive operators (`StrSize` and `StrSub`) are fairly easy to implement. The key thing to remember is that the first word of the string object is its length, so the first character in the string is 8 bytes from the start of the object. The length is stored in tagged representation, so `StrSize` just needs to load it. Be careful to use an `i8*` pointer for accessing the character data so that the address arithmetic is correct.

4.4 Booleans and Comparisons

Booleans, like all **LangF** values, are represented at runtime by a 64-bit quantity that is either `1` (for `false`) or `3` (for `true`). LLVM, however, uses 1-bit integers for the boolean results of conditional tests and as the arguments to conditional branches. Thus, it is necessary to convert between these representations. The conversions are fairly simple. To go from a **LangF** boolean to a 1-bit boolean, we need only emit an equality comparison with `true` (or `3`). To go the other way is a bit more complicated. We first need to cast the type to 64-bits and then shift left by one and add one. In C code, this process would be implemented as

```
(((int64_t)b) << 1) + 1
```

We also want to avoid unnecessary conversions. For example, if we have the **LangF** code

```
if (x == 0) then 1 else 2
```

we do not want to generate code that converts the result of the equality test to a **LangF** boolean and then tests that for equality with `true`. The trick is to be lazy about generating conversions. Since the LLVM API annotates vars with their type, we can write two functions (these are located in the `ArithGen` structure)

```
val toBool : LLVMBlock.t * LLVMVar.t -> LLVMVar.t
val toBit  : LLVMBlock.t * LLVMVar.t -> LLVMVar.t
```

that implement these coercions and then just use them when necessary. Specifically, when we store a 1-bit value into memory, pass it as an argument to a function, or return it as a result, we need to

ensure that it has the 64-bit representation. Likewise, when we use a 64-bit value as an argument to a conditional branch, then we need to ensure that it has the 1-bit representation. The conversion functions only inject code when necessary, so you just need to include them it the appropriate places.

4.5 Tuple Allocation

The **LangF** runtime provides a function for allocating an uninitialized tuple objects

```
int64_t *_langf_alloc (int32_t n);
```

This function takes the number of words in the tuple and returns a pointer to uninitialized memory for the tuple. Note that it is important to initialize the fields of the object immediately, since the garbage collector will be confused by uninitialized objects.

5 A Complete Example

We revisit the example of the iterative factorial function from the *Closure Conversion and Code Generation* document. Recall that the CFG code is

```
fun ifact_lab (ifact_clos, i, acc) {
    let n = #1(ifact_clos)
    goto frag_hdr (n, i, acc)
}
and ifact_hdr (n1, i1, acc1) {
    let t1 = IntLte(i1, n1)
    if t1 then goto frag1 (n1, i1, acc1) else goto frag2 (acc1)
}
and frag1 (n2, i2, acc2) {
    let t2 = IntAdd(i2, 1)
    let t3 = IntMul(i2, acc2)
    goto ifact_hdr (n2, t2, t3)
}
and frag2 (acc3) { ret acc3 }
```

The generated LLVM code for this function looks something like the following.

```
define i64 (i64*,i64,i64)* @ifact(i64* %clos1,i64 %i2,i64 %acc3)
    gc "statepoint-example"
{
entry_0001:
    %r7 = load i64, i64* %clos1
    br label %ifact_hdr_0002
ifact_hdr_0002:
    %n14 = phi i64 [ %n14, %frag1_0003 ], [ %r7, %entry_0001 ]
    %i15 = phi i64 [ %r9, %frag1_0003 ], [ %i2, %entry_0001 ]
    %acc16 = phi i64 [ %r13, %frag1_0003 ], [ %acc3, %entry_0001 ]
    %r8 = icmp sle i64 %i15, %n14
    br i1 %r8, label %entry_0001, label %frag1_0003
frag1_0003:
    %r9 = add i64 %i15, 2
    %r10 = sub i64 %i15, 1
    %r11 = ashr i64 %acc16, 1
```

```
    %r12 = mul i64 %r10, %r11
    %r13 = add i64 %r12, 1
    br label %ifact_hdr_0002
frag2_0004:
    ret i64 %acc16
}
```

6 Compiling the Generated LLVM Code

The code generator that you are writing produces LLVM assembly code to a "`.ll`" file. The compiler then takes three more steps to produce an executable program that you can run.

1. It translates the LLVM assembly code to x86-64 assembly code using `llc` (the LLVM compiler).
2. It patches the x86-64 assembly file to make the stack map accessible to the runtime system.
3. It compiles the patched assembly file and links it with the **LangF** runtime system to produce an executable.

In the end, a successful run of the command "`lfc.sh fool.lf`" will produce three output files: "`foo.ll`,""`foo.s`" (the x86-64 assembly), and "`foo`" (the executable).

In order for these steps to work, you must have the `llc` and `cc` commands in your path, and you must have the compiled version of the runtime system in the `runtime` directory. The 'cc' command is the standard name for the C compiler and should be in your path on any machine with command-line developer tools installed. The `Makefile` in the sample code has been modified to compile the runtime library as part of the build process, so the compiler should be able to find it.

6.1 The `llc` command

The LLVM tools that you need for this project have been installed on the Department Linux machines. To add them to your path, you need to run the following shell command:

```
module load clang-llvm/10.0.1
```

Note that you will need to run this command every time that you start a new shell. You can test if the correct version of `llc` is in your path by running the command

```
llc --version
```

which should report

```
LLVM (http://llvm.org/):
  LLVM version 10.0.1
  ...
```

(plus a lot of additional information).

If you are working on your own personal Linux or macOS machine, then you can install the LLVM system locally, but be aware that it is a large system that takes a long time to build. Please contact the instructor for more information on how to install the LLVM tools that you need for the project.

7 Document history

November 25, 2020 Original version.