CMSC 22600 **Compilers for Computer Languages** **Project 3**
Autumn 2016 **November 22, 2016**

Code Generation

Due: December 6, 2016

1 Introduction

The final project is to generate executable code for SOOL. There are basically three aspects to this project:

1. Liveness analysis to support garbage collection.
2. Mapping SOIR local variables to LLVM's SSA conventions.
3. Mapping SOIR statements and expressions to LLVM instructions.

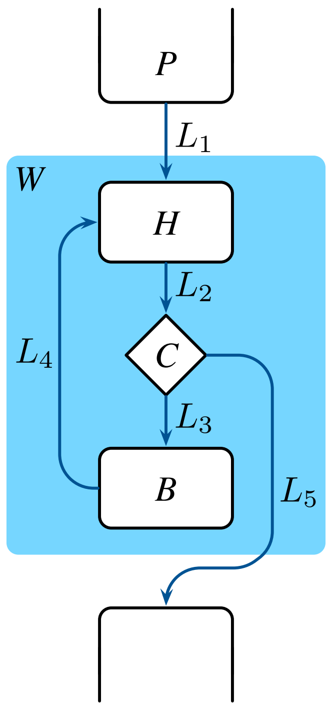
For this project, you will need to annotate the SOIR representation with information at various program points. To support this need, we have modified the SOIR representation to include *program points*, which are unique labels that can be used as keys in set, finite maps, and hash tables. Each basic block in the modified representation has its own program-point label. Furthermore, we have added a `LabelStm` statement constructor that wraps a statement with a program-point label, which is used in the liveness phase. We have also changed the expression forms that correspond to function calls (*e.g.*, `NewExp`) to be statement forms. This change is because of the need to annotate such statements with liveness information.

2 Liveness analysis

In order to support garbage collection, we will need information about which local variables are live at function call sites. Liveness information can also be used to filter out unnecessary ϕ instructions. Therefore, you will need to implement a liveness analysis for SOIR functions. You will implement this analysis in the `Liveness` structure that is provided in the sample code. The analysis has a pre-pass that adds labels to statements, such as call statements and conditionals. These labels (along with the labels on blocks) will be used as keys in a hash table that maps to sets of live variables.

Recall that liveness analysis is a backward-flow analysis and has the following transfer equations:

$$\begin{aligned}
\text{LiveIn}[s] &= \text{Gen}[s] \cup (\text{LiveOut}[s] \setminus \text{Kill}[s]) \\
\text{LiveOut}[s] &= \bigcup_{s' \in \text{succ}(s)} \text{LiveIn}[s'] \\
\text{LiveOut}[\mathit{exit}] &= \emptyset
\end{aligned}$$



$$\begin{aligned}
\text{LiveOut}[P] &= \text{LiveIn}[W] \\
\text{LiveIn}[W] &= L_1 \\
L_1 &= L_4 \\
L_2 &= \text{Gen}[C] \cup L_3 \cup L_5 \\
L_3 &= \text{Gen}[B] \cup (L_4 \setminus \text{Kill}[B]) \\
L_4 &= \text{Gen}[H] \cup (L_2 \setminus \text{Kill}[H]) \\
L_5 &= \text{LiveOut}[W]
\end{aligned}$$

Figure 1: Liveness analysis for loops

The main complication in computing liveness for SOIR is the `LoopStm` and `ExitIfStm` forms. While the body of a loop consists of a single SOIR block, it will map onto two LLVM blocks: one for the header, which will terminate with the `exit_if` test, and one for the body of the loop, which will terminate with a jump back to the loop header. Figure 1 shows this control-flow structure along with the liveness equations. When analyzing the loop W, we record the following liveness information for the loop and exit statements:

1. $\text{LiveIn}[W] = L_1$
2. $\text{LiveOut}[W] = L_5$
3. $\text{LiveIn}[C] = L_2$
4. $\text{LiveOut}[C] = L_3 \cup L_5$

3 The LLVM generator API

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 in the sample code.

3.1 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, class metadata, and index tables.
- *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.
- *Labels* are local to a function and are used to name the entry points of basic blocks.

3.2 Modules

An LLVM module is the container that holds the generated code. Its contents includes the functions that make up the program, the static data for the class metadata and interface index tables, and external declarations for runtime-system functions.

3.3 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.

3.4 Blocks

LLVM blocks are the containers for instructions. They have a unique label and start with ϕ instructions followed by a sequence of register assignments. Blocks are terminated by either a branch, conditional branch or return instruction. Not that LLVM blocks are *basic blocks* and have no internal control flow. Thus, a SOIR block can map to multiple LLVM blocks, since a SOIR block can contain conditional control flow and loops.

3.5 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 5.3.

3.6 Understanding LLVM function calls

The LLVM API uses the following interface for generating function calls:

```
val emitCall : t * {
        func : var,
        args : var list,
        live : var list
      } -> {
        ret : var option,
        live : var list
      }
```

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

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

Where `val1` and `val2` are heap pointers. The LLVM code associated with this API call involves several LLVM *intrinsics*, which are functions used by the program in order 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.val2 = 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
```

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

4 SSA conversion

We will use the approach for SSA conversion described in

> *Single-pass generation of static single-assignment form for structured languages*,
> by Marc Brandis and Hanspeter Mössenböck

(there is a link to this paper on the class website). This approach takes advantage of the fact that SOIR code is block structured and does not include unconstrained control flow. In particular, the dominator tree for SOIR code is directly determined by the syntax of the code and does not require any analysis to compute. For example, consider the following SOIR code fragment:

```
 1  var x = 1
 2  var w = 2
 3  if y {
 4    x := 2
 5  }
 6  else {
 7    var z = 1
 8    z := z+1
 9  }
10  w := x * 2
```

Because of the block structure, we know that any ϕ nodes for this code will have to be placed at the join point of the `if` statement (line 10). Thus, as we process the two arms of the `if` statement, when we see an assignment to a variable x that was in scope at line 3, we know that we will need a ϕ node for x at line 10. In this example, we need a ϕ node for `x`, but not for `w` (it is not assigned in the arms of the `if` statement) or for `z` (it is not in scope at the root of the `if` statement).

To implement this approach, we keep a stack of *pending joins*, where for each pending join we keep track of the environment at the immediate dominator of the join and a list of ϕ statements needed for the join. Whenever we see an assignment to a variable, we check if the variable was in scope at the immediate dominator of the topmost join in the stack. If so, we add the ϕ instruction for that variable (or we update the existing ϕ statement with the lhs of the assignment). Once we have processed both arms of the conditional, we can then extract the ϕ instructions from the top of the stack and insert them at the beginning of the next block. Since ϕ instruction are themselves assignments, we have to record them in the new topmost join.

As with liveness, generating ϕ instructions for a loop is slightly more difficult. Consider the loop shown in Figure 1. The join in this loop is the block H, which is where we need to place ϕ instructions. Assume that there is an assignment to a variable x in B. This assignment will force the creation of a ϕ instruction "$x_i = \phi(x_j, x_k)$," where x_j is the value of x that flows from P and x_k is the fresh LLVM variable for the left-hand-side of the assignment. The problem is that uses of x in H (or B) that occur before the assignment are dominated by the ϕ and thus must refer to x_i, but at the time that they were translated, the environment would map x to x_j. To address this problem, we need to do a simple analysis of the loop body to collect those variables that are both defined at the loop entry and are assigned to in the loop body.[2] For each variable x in this set, we add the instruction "$x_i = \phi(x_j, x_j)$" to the join for H andwe add the binding $x \mapsto x_i$ to the environment.

[2]You can either do this analysis as a prepass over the body of a function that records the assigned variables for each loop, or as pass over the loop body at the time that you encounter the loop.

We can then translate the loop body to LLVM. When we encounter a use of x before its assignment, we will map it to x_j, and when we encounter the assignment, we will update the ϕ instruction to refer to the new LLVM variable for x (*i.e.*, x_k).

The sample code includes the `JoinStack` module, which implements the stack of joins as described above.

Also note that one can filter out the ϕ instructions for any variables that are **not** live at the join point.

5 Code generation

Since SOIR is already a low-level IR, the translation to LLVM is mostly direct. 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.

5.1 Function calls

Function calls, which include both calls to SOOL functions as well as calls to runtime-system functions (*e.g.*, for allocation) are the most complicated LLVM instruction to generate. 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 SOIR variables that were in the live list to reflect ths renaming. Note that for a SOIR call of the form

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

it is important that this list **not** include the result of the function (*i.e.*, `x`), since, technically, it is killed by the assignment. The liveness information is used by LLVM to track potential GC roots in the stack.

For some runtime functions, such as the string equality tests, we are guaranteed that no garbage collection can happen during the call. In that case, we can emit an empty list for the live-variables.

5.2 Integer Arithmetic

Because the SOOL runtime does not have precise information about the types of class-member variables, we need a way for the garbage collector to distinguish between pointers and integers. The approach we take is to *tag* integers by setting their lowest bit to one. In other words, 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:

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

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	$[\![n/m]\!]$	$\Rightarrow$	$2(\lfloor [\![n]\!]/2 \rfloor / \lfloor [\![m]\!]/2 \rfloor)+1$

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.

5.3 Booleans and Comparisons

Booleans, like all SOOL 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 SOOL 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 SOOL code

```
if (x == 0) { return 1; }
```

we do not want to generate code that converts the result of the equality test to a SOOL 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 boolean 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 boolean as an argument to a conditional branch, then we need to ensure that it has the 1-bit representation.

The one other place where we need to be careful is in the treatment of boolean negation (`BoolNot`), which should be able to handle either representation. LLVM does not provide a logical negation operator, but the exclusive-or (xor) instructions can be used for this purpose ($\mathrm{not}(b) = \mathrm{xor}(b, \mathit{true})$).

6 Compiling the Generated LLVM Code

Once you producing an "`.ll`" file from your compiler, you will want to convert that to native x86-64 code and run it. To do so, you will need a special version of the LLVM tools (see below) and a compiled version of the SOOL runtime system.

To produce x86-64 assembly code, you run the **llc** command on the generated "`.ll`" file. This command will produce a "`.s`" file, which you will then have to compile and link with the runtime. Running `llc` on your compiler's output is also a good way of checking for errors in your LLVM code generator.

One complications is that we need to modify the generated assembly code slightly to make the garbage-collection information visible to the runtime system. We can so this using a **perl** command. The following sequence of shell commands will take the SOOL source file "`prog.sool`" to to the executable "`prog`:"

```
bin/soolc.sh prog.sool
bin/llc prog.ll -o prog.s
perl -i -pe \
  "s/__LLVM_StackMaps:/.globl __LLVM_StackMaps\n__LLVM_StackMaps:/" prog.s
cc prog.s lib/sool_rt.o -o prog
```

Here we are assuming that the current root directory is the `proj4` directory that contains your compiler and that the LLVM tools have been placed in the `proj4/bin` directory.

The LLVM tools that you need for this project have been installed on both the Department Linux machines (in the directory `/usr/local/bin`) and on the MacLab machines (also in `/usr/local/bin`). If you wish to install a copy on your personal machine, we provide an installation script on the course website. Note that to run this script you will need an internet connection (since it downloads the required LLVM source code) and you will need to have **cmake** (`cmake.org`) installed. To build the tools, do the following (assuming that you are in the directory containing the `build-llvm.sh` script):

```
mkdir llvm-tools
cd llvm-tools
../build-llvm.sh
```

The build process takes a significant amount of time, but once it is finished you will have a directory `llvm-tools/bin` that contains both **llc** and **opt** (the LLVM optimizer). You can leave these where they are or copy them to some other location.

7 Submission

We will collect the projects at **11pm** on **Tuesday December 6** from the SVN repositories, so make sure that you have committed your final version before then.

Important note: You are expected to submit code that **compiles** and that is well documented. Remember that points for project code are assigned 30% for coding style (documentation, choice of variable names, and program structure), and 70% for correctness. Code that does not compile will **not** receive any points for correctness.

8 Document history

December 3, 2016 Fixed description of tagged division.

December 1, 2016 Fixed typo in Figure 1.

December 1, 2016 Fixed typo in shell code.

December 1, 2016 Added additional discussion of how LLVM calls work.

November 24, 2016 Added discussion of SSA conversion for loops.

November 22, 2016 Original version.