Compiler Construction
Assignment 3: Bounds checker

Deadline: 23:59 Friday 1st February 2019

For this assignment you will write an LLVM IR pass to insert *bounds checks* into programs. FenneC, like C, is currently an *unsafe language*, where accesses to arrays are not checked. Consider the following code for instance:

```
int main(int argc, char[][] argv) {
    int[10] arr;
    arr[42] = 10; // Out-of-bounds!
    return 0;
}
```

Space for only 10 integers is allocated for array `arr`, but a write to (for example) the 42nd integer is still allowed. Such an out-of-bounds write (or buffer overflow) can corrupt other data in memory, and causes a serious security issue that is often abused by attackers.

A solution to this problem is to insert *bounds checks* before every read or write, checking whether the index or pointer is still in-bounds. This is also done in *safe languages* such as Python and Java. The goal of this assignment is to write similar checks for FenneC. An example of the *instrumented* version of the code above could be:

```
int main(int argc, char[][] argv) {
    int[10] arr;
    if (42 < 0 || 42 > 10) ERROR("out-of-bounds!");
    arr[42] = 10; // Out-of-bounds!
    return 0;
}
```

Your goal is to write a pass to automatically insert such checks in the IR, so that at runtime your program would report an error and crash, similar to Python and Java. In the simple case above it is easy to see this will go wrong during compilation, since both the size of the array (10) and the offset (42) are constant. However, this does not have to be the case, for instance when the index is a program argument. Languages such as Java dynamically associate each array with its size. For this assignment you only have to implement a simple version of bounds checking.

To see how this should be implemented in the LLVM IR, first consider the following code, which is the IR corresponding to our first example.

```
%arr = alloca i32, i32 10
... ; initialize arr to 0
%arr.idx.1 = getelementptr i32, i32* %arr, i32 42
store i32 10, i32* %arr.idx.1
```

The access to an array element consists of two instructions: a `getelementptr` (**GEP**) instruction to calculate a new pointer which points to the 42nd item, and a `store` instruction (for reads you would instead see a `load` instruction). Even though FenneC does not have pointers exposed, LLVM still uses these internally. Your task is to insert checks for every GEP instruction, meaning you will detect an error as soon as a pointer is created that is out-of-bounds. FenneC does not allow out-of-bound pointers since they can only be used for dereference (in contrast to C, where a pointer can be temporarily out-of-bounds).

The steps for accomplishing this are roughly: (1) identify all GEPs and determine their offset and (2) the size of the array (locally), (3) insert a call to a helper function that does the bounds check and error reporting. Finally, (4) when passing an array to another function, its size should be passed alongside, and (5) the size should be passed through PHI-nodes.

1 Finding GEPs and their offsets

Start by writing an LLVM module pass called `BoundsChecker` that iterates over all instructions. You can create a copy of `DummyModulePass` as a starting point. Bounds checks work at the function level, but sadly you cannot use a function pass here: for the last step of this assignment you will modify function signatures to pass bounds between function, something only a *module pass* can do. Make sure your pass runs when passing the `-coco-boundscheck` argument to `myopt`, which will automatically enable the tests. Make sure to run `git pull` before starting the assignment, since the tests for this assignment were added later.

For each instruction you want to see if it is a GEP[1]. A GEP instruction consists of a single pointer to the base of an array, and a number of indices. Each index takes an element in a specific dimension. In the IR generated for FenneC only a single index is used, except for global constants like string literals (which are pointers to an array), in which case all indices are simply zero and no bounds check is required. Write a function that returns a `Value*` that contains the *accumulated* offset of GEPs. Since multiple GEPs are possible on a single pointer, you should accumulate all these offsets recursively, adding them together. You can ignore GEPs with all-zero indices and assume non-zero GEPs only have a single index (which you can request with `GEP->getOperand(1)`. Make sure the returned value is of type `i32` (i.e., `Type::getInt32Ty(M.getContext())`). For this assignment you can assume each GEP is only executed once, and not in a loop.

2 Determining array allocation size

Once you can determine the accumulated offset of a GEP, you need to know the size of the array it is operating on. For this you will have to traverse the *use-def chain* of the pointer operand of the GEP. Write a function that, given a GEP instruction, finds the *origin* of that pointer (e.g., an `alloca`[2], a `load`[3], a function argument[4], or a constant[5]), and then from there determines its size.

Depending one the type of the origin you can then determine and return the size of the array. This size should be a `Value*` again, since the size of the array can be dynamic at runtime (e.g., allocating an array of size `argc`). Start by implementing the `alloca` origin. The returned value should be the number of elements in the `alloca`. Implement the same for constants for now, but leave function arguments and loads. The only exception is `argv` for the `main` function, which you should determine based on `argc`. If your code encounters any unknown type of origin, it should raise an error.

3 Inserting checks using helper functions

To implement the actual (runtime) checks you have to insert new code in the IR to perform these. The checking code will consist of a comparison of the offset with the size, and if the offset is greater than the bound (or smaller than 0) you print an error and terminate the program. There are two options of inserting these checks: manually constructing all the code as IR in the pass, or inserting a call to a helper function that does all this for you.

[1]`http://llvm5-docs.koenk.net/classllvm_1_1GetElementPtrInst.html`
[2]`http://llvm5-docs.koenk.net/classllvm_1_1AllocaInst.html`
[3]`http://llvm5-docs.koenk.net/classllvm_1_1LoadInst.html`
[4]`http://llvm5-docs.koenk.net/classllvm_1_1Argument.html`
[5]`http://llvm5-docs.koenk.net/classllvm_1_1Constant.html`

For this assignment you will write and use such a helper function. Recall our infrastructure contains a *runtime library*, containing utility functions written in C, which are compiled to LLVM bitcode and linked in with the program. This runtime is currently primarily used to implement functions that FenneC does not support (such as converting integers to floats). By writing a function in the runtime library that implements the check and its errors, you can simply call that function every time you want to check bounds. This same mechanism is used by the `DummyModulePass` example. Write a function (in C) `void __coco_check_bounds(long offset, long array_size)` that implements the check, and call it before every GEP.

4 Passing bounds to other functions

Your current implementation can only detect out-of-bounds accesses to arrays that are allocated in the same function. Consider the example below:

```
define void @print10th(i32* %arr) {
entry:
  %.str.0 = getelementptr inbounds [4 x i8], [4 x i8]* @.str.0, i32 0, i32 0
  %arr.ptr = getelementptr i32, i32* %arr, i32 10
  %arr.idx = load i32, i32* %arr.ptr
  %.3 = call i32 (i8*, ...) @printf(i8* %.str.0, i32 %arr.idx)
  ret void
}

define i32 @main(i32 %param.argc, i8** %argv) {
entry:
  %a = alloca i32, i32 8
  %b = alloca i32, i32 16
  ... ; initialize a and b to 0
  call void @print10th(i32* %a)
  call void @print10th(i32* %b)
  ret i32 0
}
```

Here `main` allocates two arrays, and calls `print10th` with both of these arrays. This function *might* access an out-of-bounds element, but cannot determine this now. To support these (common) cases you have to pass the size of the array alongside the array itself. You should automatically add a size parameter for every array argument to a function, and modify the function from part 2 to support `Argument` origins.

Sadly it is not possible to simply modify the function type of an existing function in LLVM, nor is it possible to add arguments to call instructions. Instead, you will have to *clone* the function that you want to modify, and replace each call instruction with an entirely new one to the new (cloned) function. Our framework contains a helper function, `addParamsToFunction`, to help you out with this process. See `llvm-passes/utils.h` for its implementation and documentation. It's important to remember this function makes a clone and returns a pointer to the new one. After modifying all functions you should create a new `CallInst` to the new function, and use `ReplaceInstWithInst`[6]. to replace the call to the old function. Afterwards make sure you delete the old functions.

5 Finding size through PHI nodes

Currently the frontend always generates GEPs that have the base pointer of an array. For instance, in FenneC it is not possible to create aliases to arrays, such as in the C code below:

```
    int a[n], b[m];
```

[6]`http://llvm5-docs.koenk.net/namespacellvm.html#a58cb353f6bb490b0c689f5f2a830414d`

```
    int *c;
    if (cond)
        c = a;
    else
        c = b;
    foo(c[i]);
```

Here the value of the "array" c depends on the branch. In IR, this looks like this:

```
entry:
  %a = alloca i32, i32 %n
  %b = alloca i32, i32 %m
  br i1 %cond, label %entry.if, label %entry.else

entry.if:                                         ; preds = %entry
  br label %entry.endif

entry.else:                                       ; preds = %entry
  br label %entry.endif

entry.endif:                                      ; preds = %entry.else, %entry.if
  %c = phi i32* [ %a, %entry.if ], [ %b, %entry.else ]
  %c.ptr = getelementptr i32, i32* %c, i32 %i
  %c.idx = load i32, i32* %c.ptr
  call void @foo(i32 *%c.idx)
```

Here you can see that the load uses a GEP that uses `%c`, which is a `phi` of the value `%a` or `%b` depending on which branch was taken. The array size of `%c` is thus either `%n` or `%m`. While the FenneC frontend does not not generate such IR directly, other optimization passes can transform FenneC IR to this. For example, compiling the following FenneC program with the `-gvn-sink` pass results in exactly the IR above. The frontend generates a GEP and load in each branch, but the `-gvn-sink` pass moves this common code out of the branch.

```
    int[n] a;
    int[m] b;
    int t = 0;
    if (cond)
        t = a[i];
    else
        t = b[i];
    foo(t);
```

The only way to know the size of `%c` is to determine this value at runtime. You can do this by creating another PHI-node[7] yourself that propagates the size alongside the array, like so:

```
entry:
  %a = alloca i32, i32 %n
  %b = alloca i32, i32 %m
  br i1 %cond, label %entry.if, label %entry.else

entry.if:                                         ; preds = %entry
  br label %entry.endif

entry.else:                                       ; preds = %entry
  br label %entry.endif

entry.endif:                                      ; preds = %entry.else, %entry.if
  %c = phi i32* [ %a, %entry.if ], [ %b, %entry.else ]
  %c.size = phi i32 [ %n, %entry.if ], [ %m, %entry.else ]
  call void @__coco_check_bounds(i32 %i, i32 %c.size)
```

[7] http://llvm5-docs.koenk.net/classllvm_1_1PHINode.html

```
  %c.ptr = getelementptr i32, i32* %c, i32 %i
  %c.idx = load i32, i32* %c.ptr
  call void @foo(i32 *%c.idx)
```

Extend your pass to duplicate every PHI-node that propagates an array and fill in the edges for every PHI-node with the size. When looking for the array size (in the function you implemented in part 2), if you encounter a PHI-node, you should return the corresponding PHI-node containing the array size. For a complete implementation you should also consider cases where the value of an incoming edge of a PHI-node is a PHI-node itself.