Compiler transformation at LLVM IR level

Contact info:
Address *ELIS (floor -3), Technicum (building I), Sint-Pietersnieuwstraat 41*
Email *compiler@elis.ugent.be*

IMPORTANT: You should always hand in your solutions using the Dropbox functionality of Minerva, both an intermediate version at the end of the lab as well as a final one before the deadline. Use the `make_tarball.sh` script to generate an archive, and make sure you send it to the teaching assistants.

DEADLINE: April 7, 23:59.

1 Goal

The goal of this lab is to write a compiler transformation which checks whether arrays are addressed within their bounds[1]. For this, we will make use of the LLVM compiler infrastructure, a versatile set of libraries and applications to simplify the development of compilers.

2 Preparing the work environment

Because of the LLVM API quickly evolving, it is important to use a fixed version. For this lab you have to develop against version 3.5 of the LLVM APIs (minor releases such as 3.5.1 are fine too). In order to install the needed LLVM libraries on the VM or on your computer[2], execute the following commands:

```
cd ~/Downloads
wget http://users.elis.ugent.be/~tbesard/compilers/llvm-3.5.1.bin.tar.bz2
sudo tar -xvf llvm-3.5.1.bin.tar.bz2 -C /
```

The API documentation accompanying this version of LLVM is available at `http://users.elis.ugent.be/~tbesard/compilers/llvm/`. You can also use the official documentation at `http://llvm.org/docs/doxygen/html/`, but beware that it tracks the current development version of LLVM and hence could contain some incompatibilities.

You will use the `cheetah` front-end compiler, a very simple compiler which only supports a minimal subset of C. The compiler emits LLVM bitcode, which you can lower to assembly using `llc`, and subsequently generate a binary from using `gcc` (this involves assembling the source and linking against the C standard library):

```
./cheetah test/$PROGRAM.c -o test/$PROGRAM.ll
llc -march=x86 test/$PROGRAM.ll -o test/helloworld.s
gcc -m32 test/$PROGRAM.s -o test/$PROGRAM.exe
```

[1] see course notes: p. 164 "A sermon on safety", p. 425 "Array-bounds checks"

[2] only Linux is supported, and make sure that if you're running a 64-bit OS you have installed the necessary packages for 32-bit compilation (such as `gcc-multilib`)

```
# Alternatively, using pipes:
./cheetah test/$PROGRAM.c \
  | llc -march=x86 \
  | gcc -x assembler -m32 -o test/$PROGRAM.exe -
```

You can enter these commands manually, which gives you full control over the compilation process (do mind that you use the LLVM binaries at `/opt/llvm-3.5/bin`). Most of the times however, you can simply rely on the `Makefile` we provide:

```
# Given: test/$PROGRAM.c
make test/$PROGRAM.ll       # generate textual LLVM bitcode
make test/$PROGRAM.s        # generate x86-assembly
make test/$PROGRAM.exe      # generate an executable
```

You don't have to enter all of these `make` commands each time, ie. if you run `make test/helloworld.exe` the files `helloworld.ll` and `helloworld.s` will be generated automatically.

3 Introduction to LLVM bitcode

The LLVM intermediate representation is a low-level SSA representation well suited for compiler analyses and transformations. A simple `for` would be represented as in Table 1.

Code is grouped in so-called *basic blocks*, and control flow within the application can only change at the boundaries of basic blocks. This means that each basic block has to be terminated with an instruction which directs control flow (or lets it continue by branching to the next basic block); such instructions are all derived from the `llvm::TerminatorInst` super class.

In order to modify the LLVM IR you will always use the accompanying APIs. For example, the `llvm::Value` class has a method `replaceAllUsesWith`. You could also instantiate a `llvm::IRBuilder` object, point it before a certain instruction using its `setInsertionPoint` method, and subsequently insert code using one of the `Create` methods.

4 Array accesses in LLVM

For this lab we will pay attention to array accesses, and check whether they address memory within the array bounds. Without such protective measures a program could crash when it is executed:

```
$ cat test/overflow.c
  int foo[10]; int n = 10;
  foo[n] = 0;
$ make test/overflow.exe && test/overflow.exe
  Segmentation fault
```

```
int n = 10;

int sum = 0;


for (
      int i = 0;


      i < n;


      i = i+1

    )

    {
      sum = sum + i;


    }
```

```
entry:
  %n = alloca i32
  store i32 10, i32* %n
  %sum = alloca i32
  store i32 0, i32* %sum
  br label %for.init

for.init:
  %i = alloca i32
  store i32 0, i32* %i
  br label %for.cond

for.cond
  %0 = load i32* %i
  %1 = load i32* %n
  %2 = icmp slt i32 %0, %1
  br i1 %2, label %for.body,
            label %for.end

for.inc:
  %6 = load i32* %i
  %7 = add i32 %6, 1
  store i32 %7, i32* %i
  br label %for.cond

for.body:
  %3 = load i32* %sum
  %4 = load i32* %i
  %5 = add i32 %3, %4
  store i32 %5, i32* %sum
  br label %for.inc

for.end:
  ret i32 0
```

Table 1: A `for` loop and its LLVM bitcode counterpart.

The goal of this lab is to prevent such crashes, and display a message to the user instead before terminating the executing with `SIGABRT`:

```
$ make test/overflow.exe && test/overflow.exe
  overflow.checked.exe: overflow.c:2: Assertion
                        'out-of-bounds array access' failed.
  Aborted
```

If you have a look at the LLVM bitcode of the `test/overflow.c` example, you can see that the array access in question consists of a `getelementptr` instruction calculating the elements memory location, and a subsequent memory operation (in this case a `store`) performing the actual operation:

```
%foo = alloca [10 x i32]
%0 = ...
%1 = getelementptr [10 x i32]* %foo, i32 0, i32 %0
store i32 0, i32* %1
```

5 Protecting array accesses

There are multiple paths toward analysing and protecting array accesses. For this lab, you will target the `getelementptr` instruction by prefixing it with code checking the array indices.

The operands of the `getelementptr` instruction are in our case as follows: a base pointer containing the memory address of the array, an index stepping in terms of the base pointer[3] and an index stepping in terms of pointer elements. See `http://llvm.org/docs/GetElementPtr.html` for more information.

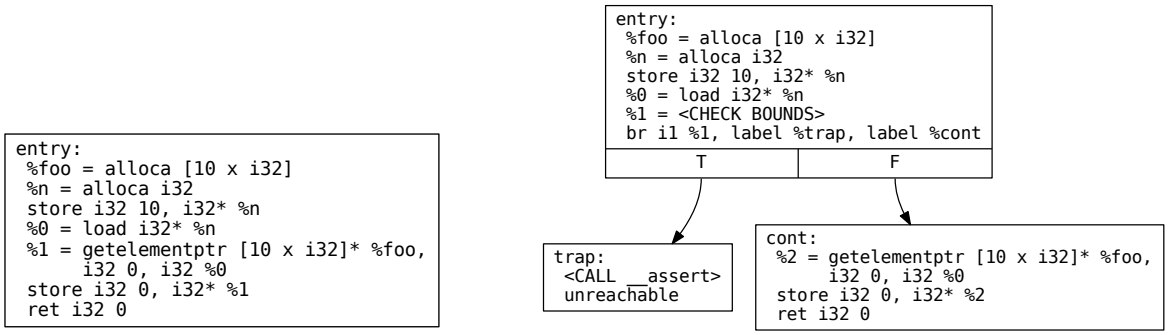
You can deduce the requested array index from these operands. Note that this index will be represented by a symbolic `llvm::Value` object, which doesn't necessarily represent a known value at compile time.

If you can check the array access statically (ie. at compile time), your pass should error straight away (use the `llvm::report_fatal_error` function for this). If the check needs to be performed during execution, insert the appropriate code to do so, and conditionally call out to the `__assert` function (part of the Sys V ABI) to display an error message and abort program execution. In both cases, use debug information emitted by the `cheetah` front end (these are the `!dbg` metadata structures in the bitcode) to generate a useful error message.

If the check happens at run time, you will need to split control flow depending on the checks outcome. You can visualize the control flow by executing the `opt` tool with the `-dot-cfg` flag:

```
make test/$PROGRAM.ll
opt -dot-cfg /test/$PROGRAM.ll
xdot cfg.$FUNCTION.dot
```

[3] as LLVM knows here that the pointer is of type `[10 x i32]`, this index will always be 0, or the resulting address would point *past* the entire array



(a) Original code (b) With bounds checking

Figure 1: Control flow before and after adding bounds checking.

In case of a simple memory access this leads to the control flow in Figure 1a (without bounds checking) and Figure 1b (with bounds checking). You can also use the `llvm-diff` tool to quickly compare two bitcode files.

6 Developing an LLVM pass

To implement the protections described above, you will create an LLVM *function pass*. This is a piece of code which is called for every function in an object file. You can use it to analyse, transform, or even completely remove a function.

The lab assignment already contains a file `BoundsCheck/Pass.cpp` which forms the basis of a bounds checking pass. In its current state the pass only lists `getelementptr` instructions, and initialises some objects which you will need throughout the lab.

As to compile the function pass, you can use the supplied `Makefile`, after which you can use the `opt` tool to load and execute the pass:

```
make BoundsCheck/libLLVMBoundsCheck.so

make test/$PROGRAM.ll
opt -load BoundsCheck/libLLVMBoundsCheck.so -cheetah-boundscheck \
    -o test/$PROGRAM.checked.bc test/$PROGRAM.ll
llvm-dis test/$PROGRAM.checked.bc -o test/$PROGRAM.checked.ll
```

Note that `opt` output binary bitcode (file extension `bc`), hence the additional call to `llvm-dis` to convert it to the textual representation.

Once again, you can automate the above commands using `make`, by adding the `.checked` prefix to the targets extension. This even recompiles the pass, if needed:

```
# (Re)compile the function pass if necessary,
# and generate the necessary .ll, .s and .exe
# files with bounds checking enabled:
make test/$PROGRAM.checked.exe
```

7 Assignment

More concretely, you will have to implement the following features in `Pass.cpp`:

- Deduce array size and index dimensions (if possible during compilation, otherwise at run time).
- Check bounds, and generate appropriate control flow in function thereof.
- Provide failure basic blocks, using debug information to display an appropriate error message.

You should test each of these features thoroughly: add necessary test cases to the `test/` folder such that each situation is verified, and check the resulting program behaviour. You should also program defensively, eg. use `assert` statements when making assumptions.

Finally, you should make an analysis (in `VERSLAG.txt`) on the cost of bounds checking: it is (statistically) significant, how would you explain that (both in terms of the generated code and the underlying hardware), and how would you minimise the slowdown?