Compiler transformation at LLVM IR level

DEADLINE: May 21, 23:59.
IMPORTANT: You should hand in your solution on the Minerva Dropbox. Create an archive using the `make_tarball.sh` script, and make sure you send it to the teaching assistants.
For any questions, contact us at `compilers@lists.ugent.be`.

1 Goal

The goal of this lab is to gain experience with the tooling and infrastructure of an important and widely-used compiler framework, LLVM. Instead if going straight from the AST to assembly as in the previous lab, you will be working at the LLVM intermediate representation (IR), a form of code that is designed to facilitate compiler analyses and transformations.

Specifically, you will need to (1) create a compiler pass to checks if arrays are addressed within their bounds[1], any (2) analyse the effect is has on the run time of generated binaries.

2 Set-up

Because of the LLVM API quickly evolving, it is important to use a specific version. For this lab you have to develop against version 7.1 of the LLVM APIs. We recommend you use the provided Docker images, because your distribution might not provide a compatible build of LLVM 7.1.

The API documentation accompanying this version of LLVM is available at `http://users.elis.ugent.be/~tbesard/compilers/llvm/doxygen/`, including a search engine. More general documentation, covering subsystems at a higher level, is available at `http://users.elis.ugent.be/~tbesard/compilers/llvm/sphinx/` (for example, see the *Writing an LLVM Pass* section). Although you can and should search on the internet[2] for help and inspiration, you should always refer to this documentation for the actual implementation details.

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` (a tool invoking the LLVM back ends), and subsequently generate a binary from using `gcc` (this involves assembling the source and linking with the C standard library):

```
./cheetah test/$PROGRAM.c -o test/$PROGRAM.ll
llc -march=x86-64 -relocation-model=pic test/$PROGRAM.ll -o test/helloworld.s
gcc test/$PROGRAM.s -o test/$PROGRAM.exe

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

[1] See course notes: p. 164 “A sermon on safety”, p. 425 “Array-bounds checks”.
[2] Useful sources include the LLVM developers mailing list, the many blog posts on writing LLVM transformations, the LLVM sources on GitHub.com, ...

You can enter these commands manually, which gives you full control over the compilation process. 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; 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 *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 directing control flow; these are all derived from the `TerminatorInst` super class.

In order to modify the LLVM IR you need to use the accompanying APIs. For example, the `Value` class has a method `replaceAllUsesWith`, and you can use `BasicBlock::Create` to add new basic blocks to a function. For more complex operations, it is recommended to instantiate a `IRBuilder` object, which you point at a certain point in the IR by means of its constructor or one of the `setInsertionPoint` methods. Afterwards, you can insert code using ony of the many `IRBuilder::Create` methods.

Every object in LLVM is a member of one or more classes. The documentation visualizes this hierarchy, for example, have a look at the documentation for `TerminatorInst`. You can see that `TerminatorInst` inherits from the `Instruction` class, which in turn indirectly extends `Value`. This last class is extremely important: almost every value in LLVM IR is a `Value`, from simple numbers to complex calculations, and the class offers a convenience `dump()` method to write information from said value to the screen. As a result, you can invoke the `dump()` method on almost every LLVM object[3]; this is very handy for debugging purposes!

[3] If developing on your own system, do know that this method might only be available on a debug build of LLVM. The Docker container provides such a build.

```
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.

4 Array accesses in LLVM

For this lab we will pay attention to array accesses, and check if 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
```

The goal of this lab is to prevent such crashes, and display a message to the user instead before terminating the execution 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

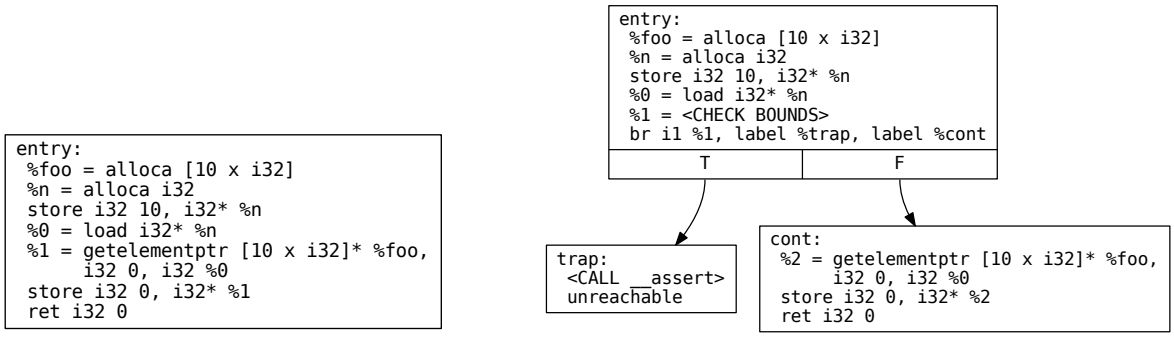
There are multiple paths toward analyzing and protecting array accesses. For this lab, you will target the `getelementptr` instruction by prefixing it with code checking the array indices.

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[4] and an index stepping in terms of pointer elements. See http://users.elis.ugent.be/~tbesard/compilers/llvm/sphinx/LangRef.html#getelementptr-instruction for a more detailed description, and http://users.elis.ugent.be/~tbesard/compilers/llvm/sphinx/GetElementPtr.html for a list of common misunderstandings.

You can deduce the requested array index from the 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

[4] In the case of arrays, where the type is `[10 x i32]` (and not a pointer to an element of type `i32`), the first index that steps in terms of the base pointer 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.

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 check's 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
```

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 a bitcode module. You can use it to analyze, transform, or even completely remove a function.

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

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/$PROGRAM.ll
opt -load BoundsCheck/libLLVMBoundsCheck.so -cheetah-bc \
    -debug-only="cheetah::boundscheck" -o test/$PROGRAM.checked.bc test/$PROGRAM.ll
llvm-dis test/$PROGRAM.checked.bc -o test/$PROGRAM.checked.ll
```

Note that `opt` outputs 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:

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

7 Assignment

Summarizing the assignment, you need to (1) implement a bounds checking pass at the LLVM IR level, and (2) analyse the performance of the resulting binaries.

7.1 Implementation

You need to implement the following features in `Pass.cpp`:

- **Decode array accesses**, deducing the indices and the size of the array.
- **Check bounds**, by testing the access indices against the array bounds.
- **Handle failure**, using debug information to display an appropriate error message that includes the source file and location of the failing access.

If possible, these steps should be performed at compile time and abort compilation. If this is not possible, code should be generated to dynamically perform the check and report failure.

You should test each of these features **thoroughly**: add test cases to the `test/` folder such that each situation is verified, and check the resulting program behavior. Do give these test cases relevant names, making clear what is being tested. You should also **program defensively**, eg. use `assert` statements when making assumptions such as casting polymorphic objects.

7.2 Analysis

The bounds check transformation is expected to lower the performance of generated binaries, even if they don't perform any out-of-bounds accesses. Since this is a common trade-off with transformations that improve correctness, we expect you to thoroughly analyze this effect:

- **Measure and analyze** the run-time performance overhead: use e.g. Linux `perf` to acquire accurate timings and use basic statistics to paint a fair picture of the performance of your binaries. Make sure you use realistic programs, such as `test/bubble_sort.c`.
- **Explain** the overhead: collect software or hardware metrics to explain the overhead. Do so in terms of the generated machine code and the underlying hardware.
- **Optimize**: try at least one optimization (as a suggestion, look at the `llvm.expect` intrinsic) that you *expect* to lower the overhead of bounds checks. Measure and analyze the impact of your optimization, and thoroughly explain why it does or does not perform as expected.

Your analysis should be between 2 and 5 pages, named `REPORT.pdf` (see `make_tarball.sh`). You should implement the above optimization in the `Pass.cpp` file you hand in, but make sure you guard it with some flag (`if (optimize)`, or using a `llvm::cl::opt` CLI flag).

Do not neglect this part of the lab, as it accounts for a significant part of the total grade!