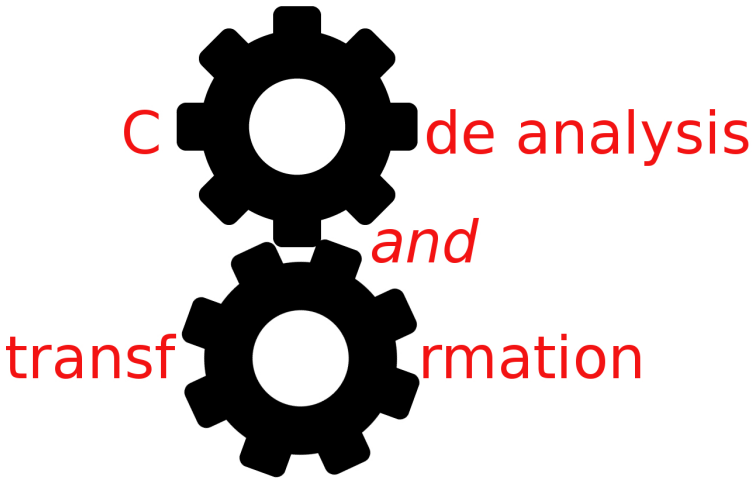

Homework H0

Contents

1 Description **3**

2 The CAT language **4**
2.1 The CAT API 4
2.2 Code example 4
2.3 Computation cost 5

3 Homework **6**
3.1 Develop 6
3.2 Assignment 6
3.3 Assumptions 6
3.4 Examples of Output of Your Work 6
3.5 Testing your work 7
3.6 Advice 8
3.7 LLVM API and Friends 8
3.8 What to submit 9

1 Description

Write an LLVM pass starting from the `cat-c` compiler (`download`) using its branch `v14`.

Your work needs to analyze programs written using the CAT language. First we describe the CAT language, and then we describe your first assignment.

2 The CAT language

The CAT language is embedded in the C language and it is composed by operations performed by invoking the CAT API. The CAT API is defined in CAT.h (and described next), which is available in your tests. For example, `CAT_add` is the add operation of the CAT language.

Variables in the CAT language can only be created by invoking `CAT_new` and they are called "CAT variables". `CAT_new` returns a reference to the CAT variable just created. In other words, the return value **is not** a CAT variable; instead, it includes the memory address where such new CAT variable has been allocated.

The CAT function `CAT_get` takes a reference to a CAT variable as input and it returns its value stored within.

The language used to invoke CAT functions is C.

2.1 The CAT API

All homework assignments will need to analyze (and transform in later assignments) C programs that invoke a known set of functions called CAT API. Next are the C functions that your homework needs to consider: `CAT_new`, `CAT_add`, `CAT_sub`, `CAT_get`, `CAT_set`, `CAT_destroy`.

Your work will have to understand how these functions are invoked by the C program given as input. The semantics of these functions are the following:

1. `CATData CAT_new (int64_t value)`: Create a new memory object called "CAT variable". This object includes an integer 64 bits value, which is initialized by this function before returning the reference to the new object allocated. The initial value is set to `value`. The CAT variable is stored in the memory heap.

2. `void CAT_add (CATData result, const CATData v1, const CATData v2)`: It adds the values within `v1` and `v2` and it stores the result of this operation within the CAT variable `result`.

3. `void CAT_sub (CATData result, const CATData v1, const CATData v2)`: It subtracts the value within `v2` from the value within `v1` (e.g., `v1 - v2`) and it stores the result of this operation within the CAT variable `result`.

4. `const int64_t CAT_get (const CATData v)`: It returns the value within the CAT variable `v`.

5. `void CAT_set (CATData v, int64_t value)`: It stores `value` within the CAT variable `v`.

6. `void CAT_destroy (CATData v)`: It destroys the CAT variable `v`. Accessing a CAT variable after being destroyed is not allowed (it is an undefined behavior).

You can find the CAT API in CAT.h available in the distributed tests.

2.2 Code example

The following program prints `5`.

```
#include <stdint.h>
#include <stdio.h>
#include <CAT.h>

int main (){
  CATData d1;
  CATData d2;
  CATData d3;

  d1  = CAT_new(2);
  d2  = CAT_new(3);
```

```
  d3  = CAT_new(0);

  CAT_add(d3, d1, d2);

  int64_t valueComputed = CAT_get(d3);
  printf("%ld", valueComputed);

  CAT_destroy(d1);
  CAT_destroy(d2);
  CAT_destroy(d3);

  return 0;
}
```

2.3 Computation cost

Next is the computation cost paid for each dynamic invocation of each CAT API. The cumulative cost paid for all dynamic invocations of all CAT APIs executed at run-time is the computation cost of a program.

- Cost of `CAT_new`: 4
- Cost of `CAT_destroy`: 1
- Cost of `CAT_add` and `CAT_sub`: 3
- Cost of `CAT_get`: 2
- Cost of `CAT_set`: 1

3 Homework

3.1 Develop

To develop your compiler, extend `cat-c/src/CatPass.cpp`.

Follow the instructions in `cat-c/README` to build and run your compiler.

To test your compiler has been properly installed:

```
$ cat-c --version
```

The output needs to include

```
clang version 14.0.6
```

3.2 Assignment

Write an LLVM pass to print the set of invocations (call instructions) of the CAT APIs that allocate or modify a CAT variable. A CAT variable is printed as the call instruction that allocates it (i.e., a call to `CAT_new`).

Specifically, for each program function, you must print in a single line

- the name of a program's function that contains definitions of CAT variables
- the CAT variable
- the call instruction that modifies the CAT variable

The order of the lines you print is not important. Your output will be automatically sorted by our scripts.

Constraints Next are the constraints for your solution:

- H0 has to be intra-procedural.
- You cannot modify the IR code.

No other constraints exist.

3.3 Assumptions

For the H0 homework, you can take advantage of the following assumptions about the C code that invokes the CAT API.

1. A C variable used to store the return value of `CAT_new` (i.e., reference to a CAT variable) is defined statically not more than once in the C function where it has been allocated.
2. A C variable that includes a reference to a CAT variable cannot be copied to other C variables (no aliasing).
3. A C variable that includes a reference to a CAT variable cannot be copied into a data structure.
4. A C variable that includes a reference to a CAT variable cannot be given as argument to a call to a function.
5. Finally, all CAT APIs are available as declaration in the IR files your pass will analyze.

3.4 Examples of Output of Your Work

Next we show an example of output that your pass will need to produce.

`H0.tar.bz2` includes a few programs you can use to test your work.

Example Consider the following program:

```
#include <stdio.h>
#include "CAT.h"

int CAT_execution1 (void){
  CATData d1;
  CATData d2;
  CATData d3;
  d1  = CAT_new(5);
  d2  = CAT_new(8);
  d3  = CAT_new(0);
  CAT_add(d3, d1, d2);
  CAT_sub(d3, d3, d1);
  CAT_set(d1, 3);
  return CAT_get(d3);
}

int CAT_execution2 (void){
  CATData d;
  d  = CAT_new(5);
  return CAT_get(d);
}

int main (int argc, char *argv[]){
  int v = CAT_execution1() + CAT_execution2();
  printf("CAT variables = %ld\n", CAT_variables());
  printf("CAT cost = %ld\n", CAT_cost());
  return 0;
}
```

your pass must generate the following output (stored in `output/oracle_output`):

```
CAT_execution1   %1 = call i8* @CAT_new(i64 5) #3   %1 = call i8* @CAT_new(i64 5) #3
CAT_execution1   %1 = call i8* @CAT_new(i64 5) #3   call void @CAT_set(i8* %1, i64 3) #3
CAT_execution1   %2 = call i8* @CAT_new(i64 8) #3   %2 = call i8* @CAT_new(i64 8) #3
CAT_execution1   %3 = call i8* @CAT_new(i64 0) #3   call void @CAT_add(i8* %3, i8* %1, i8* %2) #3
CAT_execution1   %3 = call i8* @CAT_new(i64 0) #3   call void @CAT_sub(i8* %3, i8* %3, i8* %1) #3
CAT_execution1   %3 = call i8* @CAT_new(i64 0) #3   %3 = call i8* @CAT_new(i64 0) #3
CAT_execution2   %1 = call i8* @CAT_new(i64 5) #3   %1 = call i8* @CAT_new(i64 5) #3
```

3.5 Testing your work

Run all tests. Go to `H0/tests` and run `make` to test your work.

The following output means you passed all tests:

```
./misc/run_tests_parallel.sh
CAT: Run all tests
CAT:    Spawn all tests in parallel
CAT:    Wait all threads to finish
CAT:    All tests are done
CAT: Check the results
```

```
CAT:    SUMMARY: 7 tests passed out of 7
CAT:    SUMMARY: Total execution cost = 0
CAT:    Your homework both passed all tests and it optimized the code enough, congratulations!
```

If you didn't pass a test, then the output will include all tests that have failed.

Run a test. At some point, you may want to figure out why you failed a test. To do so, go to the directory of such test.

Let us assume `test0` has failed. Follow the next steps to understand why you failed such test:

1. Go to the test directory:

   ```
   $ cd H0/tests/test0
   ```

2. Compile the C program included in the failed test:

   ```
   $ make clean ; make
   ```

3. Check the output generated by your pass against the oracle output:

   ```
   $ make check
   ```

   and follow the instructions printed in the output to debug your work.

3.6 Advice

`H0.tar.bz2` includes a few programs you can use to test your work. This archive also includes Makefiles that show how to invoke `cat-c` to generate the output file `compiler_output`. Read these makefiles to become familiar with `cat-c` and `llvm` tools.

The correct output of a test is stored in its subdirectory called `output`. When you will debug a test, read these output files to understand what you should generate and compare it with your current output.

3.7 LLVM API and Friends

This section lists the set of LLVM APIs and headers I have used in my (multiple) solutions (this is the union of all APIs across solutions). You can choose whether or not using these APIs.

- Method `getName` of the class `Function`
- Method `getFunction` of the class `Module`
- `isa<LLVM CLASS>(LLVM OBJECT)`. For example, `isa<CallInst>(i)` where `i` is an instance of the class `Instruction`
- `cast<LLVM CLASS>(LLVM OBJECT)`. For example, `CallInst *callInst = cast<CallInst>(i)` where `i` is an instance of the class `Instruction`
- `getCalledFunction` of the class `CallInst`
- `getArgOperand` of the class `CallInst`
- `errs()` to get the standard error stream

Next are some headers that you did not see in slides yet, and you might find them useful:

```
#include "llvm/Pass.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/Function.h"
#include "llvm/IR/Instructions.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/Transforms/IPO/PassManagerBuilder.h"
```

3.8 What to submit

Submit via Canvas the file `sources.tar.bz2` generated by `collect_src.sh` script of the middleend git repository you have cloned.

Good luck with your work!