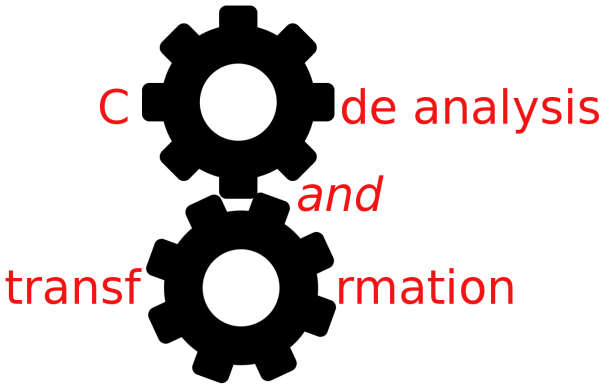

Homework H5

1 Description

Write an LLVM pass starting from the code you have developed for H4.

This homework is composed by three assignments. These assignments require to extend your work (H4) to remove an increasing number of assumptions you relied on to develop your prior work.

As it was the case for prior homework, the only variables you need to consider are the CAT variables.

1.1 First assignment

For this assignment, you need to remove the following assumption from the ones you relied on for H4:

- A variable that includes a reference to a CAT variable does not escape the function where it has been declared.

You need to extend your work to implement the constant propagation + reaching definition **without** relying on the above assumption.

Therefore, the only assumptions you can rely on for this assignment are:

1. A variable used to store the return value of `CAT_create_signed_value` (i.e., reference to a CAT variable) is defined statically not more than once in the function it has been declared.
2. A variable that includes a reference to a CAT variable does not get copied to other variables.
3. A variable that includes a reference to a CAT variable does not get copied in any data structure.

To understand the impact of this assumption you cannot rely on anymore, consider the following C function:

```
void a_generic_C_function (void){
  CATData d1 = CAT_create_signed_value(20);
  int64_t v = CAT_get_signed_value(d1);
  printf("%ld ", v);
  return ;
}
```

Your current pass (i.e., H4) is able to transform the above code to the following one:

```
void a_generic_C_function (void){
  int64_t v = 20;
  printf("%ld ", v);
  return ;
}
```

In the original code, the CAT variable `d1` is defined within the same function that is used (e.g., `a_generic_C_function`). Until H4 included, you could assume that this was the case for all CAT variables used in any C function. For H5, instead, you cannot assume this anymore: now you can have CAT variables as parameters of the function your pass analyzes.

For example, now your pass has to be able to handle the following C function:

```
void a_generic_C_function (CATData d1){
  int64_t v = CAT_get_signed_value(d1);
  printf("%ld ", v);
  return ;
}
```

Because your pass analyzes one function at a time (i.e., intra-procedural), you must assume that you do not know anything about the value stored inside `d1` at the entry point of the function `a_generic_C_function`. Therefore, your pass cannot transform the above code to set the variable `v` with a constant value.

Moreover, consider the following code:

```
void a_generic_C_function (CATData d1){
  int64_t v = CAT_get_signed_value(d1);
  if (v > 10){
    d1 = CAT_create_signed_value(50);
  }

  int64_t v2 = CAT_get_signed_value(d1);
  printf("%ld ", v2);

  return ;
}
```

Consider the instruction that sets the variable `v2`. This instruction has only one definition of `d1` that reaches it, which is the following one:

```
d1 = CAT_create_signed_value(50);
```

Therefore, your current constant propagation algorithm would (wrongly) transform the code to set the variable `v2` with the constant `50`. By doing this transformation, your pass changed the semantics of the original code. The reason is because `v2` might be set with the value stored in the CAT variable `d1` given as input of `a_generic_C_function` and not the one created by the above instruction that invokes `CAT_create_signed_value`.

1.2 Second assignment

For this assignment, you need to remove the following assumptions from the ones you relied on for H4:

- A variable used to store the return value of `CAT_create_signed_value` (i.e., reference to a CAT variable) is defined statically not more than once in the function it has been declared.
- A variable that includes a reference to a CAT variable does not get copied to other variables.
- A variable that includes a reference to a CAT variable does not escape the function where it has been declared.

You need to extend your work to implement the constant propagation + reaching definition **without** relying on the above assumptions.

Therefore, the only assumptions you can rely on for this assignment are:

1. A variable that includes a reference to a CAT variable does not get copied in any data structure.

Therefore, now your pass has to be able to analyze and transform the following C functions:

```
void a_generic_C_function (CATData d1){
  int64_t v = CAT_get_signed_value(d1);
  if (v > 10){
    d1 = CAT_create_signed_value(50);
  } else {
    d1 = CAT_create_signed_value(20);
  }

  int64_t v2 = CAT_get_signed_value(d1);
  printf("%ld ", v2);

  return ;
}
```

as well as the following one:

```
void a_generic_C_function (CATData d1){
  CATData d2;
  int64_t v = CAT_get_signed_value(d1);
  if (v > 10){
    d2 = CAT_create_signed_value(50);
  } else {
    d2 = d1;
  }

  int64_t v2 = CAT_get_signed_value(d2);
  printf("%ld ", v2);

  return ;
}
```

1.3 Third assignment

For this assignment, you need to remove the following assumptions from the ones you relied on for H4:

- A variable that includes a reference to a CAT variable does not get copied in any data structure.
- A variable that includes a reference to a CAT variable does not get copied to other variables.
- A variable used to store the return value of `CAT_create_signed_value` (i.e., reference to a CAT variable) is defined statically not more than once in the function it has been declared.
- A variable that includes a reference to a CAT variable does not escape the function where it has been declared.

You need to extend your work to implement the constant propagation + reaching definition **without** relying on the above assumptions.

Therefore, there is no assumptions about the C program given as input you can rely on for this assignment.

Therefore, now your pass has to be able to analyze and transform the following C functions:

```
void a_generic_C_function (void){
  CATData *pointer;
  (*pointer) = CAT_create_signed_value(50);

  int64_t v2 = CAT_get_signed_value(*pointer);
  printf("%ld ", v2);

  return ;
}
```

Correctness comes first (before the ability to improve code). Therefore, for this last assignment, you have to implement a correct solution by being overly conservative: if any CAT variable get stored in memory, do not consider them as candidates for the code transformation implemented by your constant propagation pass.

1.4 Final note

The above C code are just examples of functions your pass has to be able to handle for this homework.

Your solution of a given assignment needs to be able to handle any C program that respect (only) the assumptions of that assignment.

2 API

This section describes the set of LLVM APIs I have used in my H5 solution that I did not used in prior assignments. You can choose whether or not using these APIs.

The next APIs are related to the first assignment:

- To create the 32-bits integer constant `0` :

  ```
  Constant *zeroConst = ConstantInt::get(IntegerType::get(m->getContext(), 32), 0, true);
  ```

  where `m` is an instance of `Module`.

- To create a new instruction "add" and insert it just before another instruction called `inst`:

  ```
  Instruction *newInst = BinaryOperator::Create(Instruction::Add, zeroConst, zeroConst, "", inst)
  ```

  where `inst` is an instance of `Instruction`, and `zeroConst` is an instance of `Constant`.

- To check if an instance of `Value` is an argument of a function:

  ```
  isa<Argument>(v)
  ```

  where `v` is an instance of `Value`.

- To delete an instruction from the function it belongs to:

  ```
  i->eraseFromParent()
  ```

where `i` is an instance of `Instruction`.

The next APIs are related to the second assignment and they are new compared to the first assignment:

- To check if an instance of the class `Value` is an instance of the class `PHINode`:

  ```
  isa<PHINode>(v)
  ```

  where `v` is an instance of `Value`.

The next APIs are related to the third assignment and they are new compared to the second assignment:

- To fetch the variable stored in memory by a store instruction:

  ```
  Value *valueStored = storeInst->getValueOperand()
  ```

  where `storeInst` is an instance of `StoreInst`.

3 Testing your work

To test your work, go to a test you have available:

```
cd H5/tests/test0
```

set the path of your pass in `LLVMPASSPATH` of Makefile.
Now, invoke your pass

```
make
```

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

```
make check
```

If you've passed the test, you'll see the following output:

```
Test passed!
```

Otherwise, the output message tells you what went wrong. Output differences are stored in the `diff` directory .
Good luck with your work!

4 What to submit

Submit via Canvas

- The C++ file you've implemented (CatPass1.cpp) for the first assignment
- The C++ file you've implemented (CatPass2.cpp) for the second assignment
- The C++ file you've implemented (CatPass3.cpp) for the third assignment

For your information:

- my solution for H5 first assignment added 38 lines of C++ code to H4
- my solution for H5 second assignment added 3 lines of C++ code to the first assignment of H5
- my solution for H5 third assignment added 20 lines of C++ code to the second assignment of H5

The number of lines of code are computed by using `sloccount`.

5 Homework due

11/14 at 2am.