Title: Improved Loop Execution Modeling in Clang Static Analyzer

Student: Peter Szecsi
Email Address: szepet95 at gmail dot com

The short description of the project:

The Clang Static Analyzer has limited capabilities while simulating loops. In the current implementation the analyzer relies on a control number N which determines how many times to unroll a loop. After the first N steps are simulated, the analyzer does not have any information on how long it will be iterated on and it stops the simulation of that execution path (this N is 4 by default). This could result in a loss of coverage, and indicate false negatives.
Problem example:

#1
```
void foo(int NUM) {
      int x = 0;
      for(int i=0;i<NUM;++i)
      {
      ... // Don't change x.
      }
      int a = 2/x; // Will find the division by zero when analyzing
it as a top level function because we simulate the path where we
step in the loop only once.
}
```

#2
```
void foo() {
      int x = 0;
      for(int i=0;i<10;++i)
      {
            …
      }
int a = 2/x; // Won't find the division by zero since the loop
bound is known and no path exits the loop before the 4th step.
Then, while simulating the 4th step we cut the analysis of this
path because of the lack of information.
}
```

There is already an alternative method to continue the simulation that breaks the loop after N iterations.[1] However, it comes at the price that (almost) all of the MemRegions values will be invalidated. Hence, the false positive rate is relatively high in these paths.

My project would aim to improve the simulation of the loops in general.
Moreover, it would provide an extensible and incremental way of loop widening in which only the relevant regions are invalidated, and thus can be turned on by default.

Proposed Improvements:

0) Evaluation

Since the naive loop widening is already implemented, I would start with experimenting that. An important part of the project would be to measure and compare the analysis process with or without this widening turned on.
The following statistics will be collected and evaluated: coverage, performance, false positive rate, false negative rate.
I will categorize the false positives depending on if they are coming from the imprecise invalidation or not. The number of this kind of bugs can determine how impactful the proposed improvements on widening can be. On the other hand, the false positive reports which found after a simple, easily modellable loop could influence the importance order of the proposed features.
Once the evaluation is done, it would be clear what specific benefits would the widening bring and what would be the benefits of modeling complete/precise execution of loops with known bounds. That could determine the order and the priority of the listed proposed features.

1) Known bounds modeling:

The control number described above leads to the loss of coverage, most likely in cases of known bound. However, in cases when the body of the loop is simple (easily modellable) and bound is not extremely large we could efficiently simulate the loop to the end.

```
int arr[100];
int x = 0;
for(int i=1; i<100; ++i) // Not too large bound.
{
     arr[i] = f(i); // Quite simple body.
}
int a = 1/x; // We could find this bug.
```

On the other hand there are cases when we cannot model the loop so easily.

```
for(customInt i=0; i<6e23; ++i)
{
                if(complexfun1(i)) …
                if(complexfun2(i)) …
                …
                /*insert a lot of complex stmt here*/
                …
}// In a case like this the loop’s body is complex and the bound
is way too large to simulate the whole loop. But because of the
known bound, we possibly could improve the analysis with some
minor heuristics. E.g. model the last iteration of the loop (as it
is listed in the optional work).
```

This is the motivation for the first milestone which could be summarized in the following way: Design and implement a solution that would work for a loop that has a known bound of iterations. Specifically, consider completely unrolling such loops when they are not too complex.

2) Loop widening improvements

I suggest to invalidate only the regions that have their values possibly modified.
So let's consider the next example:

```
...
int a = 4;
int x, maxval = 0;
for(int i = 1; i<100;++i)
{
      x = f(i);
      if(x < maxval)
      {
      maxval = x;
}
} // When we decide to cut the analysis of the loop then we can
clearly see that we need to invalidate only the values of the
variables "maxval" and "x".
std::cout << maxval << std::endl;
int c = 3/(a-4); // Regardless the loop widening we can spot the
division by zero here since we don't invalidate the information
about the variable "a".
...
```

To reach results like above, I suggest iterating over the statements of the loop and handle them one by one. An implementation of this approach could be done with an ASTVisitor. It could handle the statements via its callbacks and collect the regions which should be invalidated. We could not implement all of the callbacks at once but the visitor would provide an incremental way for that. Whenever we encounter any statement which is not handled at that time then we would fallback to the conservative method and not widening the loop. Moreover, this improvement should contain an additional check for not invalidating too much value. In case we successfully handle all the cases but the invalidation would affect a significant portion of the values, then again, the widening should not be continued. The above restrictions are important because too few information would result in a high false positive rate.

To summarize these changes, the widening should work with the following restrictions after this milestone:

- Invalidate the entire store if there are any pointer mutations or escaping.
- If there is no pointer operation, widen only the locals that are modified by the loop.
- Use ASTVisitor to go through the loop and find changed locals as well as determine if there are pointer operations. If there is an AST operation we do not have support for, conservatively, assume that it might be a pointer operation.

3) Pointer operation case enhancements in loop widening

Additionally to the previous milestone I would implement some important ASTVisitor callbacks in order to deal with simple pointer operations.
So the first restriction listed above would change to the following:

- Use ASTVisitor to identify syntactically which locals are used as the base for a pointer operation. For example, for “a[i]”, the base will be “a”.
- In RegionStore, invalidate only the memory regions transitively reachable via the locals, in our example, it will be “a” and everything reachable from it. For example, if there was a statement “int *b = a”, both regions pointed to by “b” and “a” will be invalidated. Note, that here we rely on the aliasing relations already captured by the RegionStore to determine what needs to be invalidated.
- Note that this will only work for local regions that do not escape. If we encounter an operation of any other region, we should be conservative and invalidate everything.

4) Nested Loops

As I see, these changes should scale with nested loops. However, it will be very important to understand and document what the semantics are on nested loops and make sure they are well tested.

Optional work

1. Evaluate which other (non-implemented) callbacks are often called in the ASTVisitor and aim to implement them.
2. Investigate the coverage and report changes when we assume that a loop with unknown bound always will be executed at least once.
3. Create an additional path for loops with known bound on which simulate the last iteration of the loop after widening.

Timeline

- 29 May – 4 June: Get more familiar with the current loop simulation algorithm and the functioning of the analyzer core.
- 5 June – 11 June: Run the current widening implementation on various projects and evaluate its results.
- 12 June – 25 June: Determine the heuristics used on „Known bounds modeling” and implement them.
- 26 June – 2 July: Adding test cases to „Known bounds modeling” and write proper documentation about it.
- 3 July - 9 July: Implementation of loop widening improvements stage I.
- 10 July – 18 July: Adding test cases to the initial widening implementation.
- 19 July - 31 July: Improving simple pointer operations in loop widening.
- 1 August - 8 August: Vacation.
- 9 August - 13 August: Adding test cases to the stage II widening.
- 14 August - 30 August: Testing (primarily nested loops), fixing bugs, documenting the project.

About me

I am a third year BSc student at Eötvös Loránd University, Budapest. It would be my first Summer of Code project but I have already contributed to clang:
During the last summer I have uploaded 2 patches:
- An accepted and merged Clang-Tidy checker [2]
- An accepted and merged Clang SA checker [3]

Since then I have been working on cross translational unit analysis (and because of that I have uploaded some patches about the ASTImporter [4][5][6]). Furthermore, I participated in the preparations of a talk that was accepted at the EuroLLVM conference. [7]
I like working on algorithmic challenges and enjoy participating in programming competitions. I found SA interesting because there is a lot of algorithms in the Static Analyzer which could be optimized/made more precise by heuristics.
I will be available to work full time (~40 hours per week) on this project.

References

[1] https://reviews.llvm.org/D12358
[2] https://reviews.llvm.org/D22507
[3] https://reviews.llvm.org/D24246
[4] https://reviews.llvm.org/D29612
[5] https://reviews.llvm.org/D30876
[6] https://reviews.llvm.org/D30877
[7] http://llvm.org/devmtg/2017-03//2017/02/20/accepted-sessions.html#7