GSoC 2023 Extending gcc -fanalyzer C++ support for self-analysis.

PRIOUR Benjamin
priour.be@gmail.com

Synopsis

Currently, the static analyzer offers little dedicated support for C++ [1]. Even when projecting the most straightforward valid C test cases to C++, the checkers output incorrect diagnostics, either by their absence or imprecision [2]. The latter is partly due to the complexity of the generated GIMPLE of C++ code. Yet with the added support of understanding dynamic dispatch [3] as part of GSoC 2021, a solid infrastructure is already in place to extend the existing checkers to C++ programs. The issues this project aims to tackle are all prerequisites to further extensions of C++ support, like supporting iteration of C++ containers [PR106392] or supporting use-after-move. Note that this project is not part of GCC 'official' projects list for GSoC 2023, but it has been and is discussed with David Malcolm.

Project Goals

The aim of this project is to enable the analyzer to self-analyze itself. To do so, the following items should be implemented (**m**: minor, **M**: Major feature)

1. Generalize *gcc.dg/analyzer* tests to be run with both C and C++ [PR96395] and record both false positives and negatives by building the analyzer with -fanalyzer enabled. At this point, a solution might have to be found to split the dump graphs, as on my box xdot is impossibly slow when loading the exploded-graph dump of '*sm-fd.cc*', even when using the default depth parameters [**M**].
2. Support the options relative to the checker sm-malloc
   - -Wanalyser-double-free should behave properly for C++ allocations pairs new, new[], delete and delete[] both throwing and non-throwing versions. At the moment, only their non-throwing counterparts are somewhat handled, yet incorrectly as the expected -Wanalyzer-double-free is replaced by -Wanalyzer-use-after-free [**m**] and an incorrect -Wanalyzer-possible-null-dereference is emitted [**m**, **almost-fixed**]. I filed it as bug PR109365 [2].
3. Add support for tracking unique_ptr null-dereference [**M**]. While smart_ptr is correctly handled, the code snippet below demonstrates that this warning is not emitted for unique_ptr [4].

   Figure 1 - First test case for unique_ptr *support*

```
struct A {int x; int y;};
int main () {
  std::unique_ptr<A> a;
  a->x = 12; /* -Wanalyzer-null-dereference missing */
  return 0;
}
```

4. Improve the diagnostic paths for the standard library, with shared_ptr as a comparison point, so that they do not wander through the standard library code [**M**]. See Figure 2 and the diagnostic on https://godbolt.org/z/o16jnWocs.

Figure 2 - Reproducer to demonstrate unnecessarily long diagnostic paths when using the standard library.

```
struct A {int x; int y;};
int main () {
  std::shared_ptr<A> a;
  a->x = 4; /* Diagnostic path should stop here rather than going to shared_ptr_base.h */
  return 0;
}
```

May-be solved alongside

- Support of 'placement new' sizes, emitting warning on incorrect pairing of placement new/delete, as part of goal {2}.

Timeline (format DD:MM [% of initial total project time])

During the entirety of the project, I will spend 35 hours per week on it, i.e. the legal maximum according to my university's internship regulation, and be available from mid-June to late September. I would also like to take a 4-days break around the 20th of July, but will be otherwise fully available from mid-June to 23rd of September. Thereafter, I could possibly commit 4 hours a day at most until November to polish the project. The schedule is based on an expected 350-hours long project.

Pre coding period: from now on to June

Gain more familiarization with the GIMPLE and SSA formats (e.g. when {CLOBBER} occurs in GIMPLE, and how does it morph into SSA {v} {CLOBBER}) as well as insight in the analyzer dumps. I believe a good side goal would be to complete David Malcolm's newbies guide https://gcc-newbies-guide.readthedocs.io/en/latest/

PART 1 - 21:06 - 11:07 [30%]

1. Run the analyzer against itself and write down the issues, for instance record any source file the analyzer proves too slow to run on. At the moment of writing, building '*sm-malloc.cc*' with "-fanalyzer -Wanalyzer-too-complex" yields 90 "-Wanalyzer-too-complex" and 13 false positives "-Wanalyzer-possible-null-dereference" as a result of the above bug marked [**almost-fixed**]. All but the simplest functions yield not "-Wanalyzer-too-complex" with the default depth parameters. Therefore, a moment will be taken to tune those parameters to draw a balance between analysis's completeness and build time.

2. Elect the first test cases to move from *gcc.dg* to c-c++-common, writing their C++ counterparts as necessary, and update the Tcl script *analyzer.exp*. I already have some prior knowledge of Tcl, yet some time is allocated here to fully comprehend the script.

3. Add any new test cases to the above project goals. At this point, they are still expected to change and be incomplete, but should provide an approximation fine enough for the following phases.

This time is also allocated to formally describe the goals and the possible pitfalls.

PART 2 - 12:07 - 21:07, [4-days break], 25:07 - 20:08 [55%]

Actual coding phase. The relative order of the project goals mirrors the order I will tackle them, as goal {2} is the simplest of all, and I have already begun solving it - see Figure 3.

Then the unique_ptr support is to be added. A grep through the *gcc/gcc/* subdirectory yielded 2903 results, so this is a core feature of this project.

At the end of each problem, the proposed work should be confronted to PART 1 test cases, and a patch should be submitted. This implies a quality check of the provided patches, and a request for feedback from the community and not only the mentor.

*Figure 3 - The above [**almost-fixed**] bug no longer emits false positives when using the common* operator new*, whereas non-throwing versions of* new*, for which the warning was actually justified, are distinguished using the predicate* TYPE_NOEXCEPT_P*. Need proper testing on* placement new.

```
// gcc/analyzer/sm-malloc.cc:malloc_state_machine:on_stmt
 if (is_named_call_p (callee_fndecl, "operator new", call, 1)) // always noexcept
   on_allocator_call (sm_ctxt, call, &m_scalar_delete, true);
 else if (is_named_call_p (callee_fndecl, "operator new", call, 2))
  {
   bool returns_nonnull = !TYPE_NOEXCEPT_P (TREE_TYPE (callee_fndecl));
   on_allocator_call (sm_ctxt, call, &m_scalar_delete, returns_nonnull);
  }
 else if (is_named_call_p (callee_fndecl, "operator new []", call, 1))
   on_allocator_call (sm_ctxt, call, &m_vector_delete, true);
 else if (is_named_call_p (callee_fndecl, "operator new []", call, 2))
  {
   bool returns_nonnull = !TYPE_NOEXCEPT_P (TREE_TYPE (callee_fndecl));
   on_allocator_call (sm_ctxt, call, &m_vector_delete, returns_nonnull);
  }
```

PART 3 - 21:08 - 31:08 [15%]

While testing already occurs all along the coding period, these 2 weeks should be about consolidating the integration of the newly added C++ support with the existing C -see *Figure 4* and *Figure 5* below-, and trying to find what false positives and negatives persist. This will also be the occasion to once again run the analyzer on itself, and actually fix the true positives found out.

Figure 4 - Inconsistent behavior when mixing malloc *and* new *falsely emits* *-Wanalyzer-use-after-free* *instead of* *-Wanalyzer-mismatching* *allocation and* *-Wanalyzer-double-free*

```
class A {};

int main () {
   A* a = (A*) __builtin_malloc(sizeof(A));
   __builtin_free (a);
   delete a;
   return 0;
}
```

Figure 5 - Falsely emits no diagnostics

```
class A {};

int main () {
   A* a = new A ();
   delete a;
   __builtin_free (a);
   return 0;
}
```

Motivation and Skill set

I am currently in the first year of a Master degree in Computer Science and Applied Mathematics, at the engineering school ENSIMAG - Grenoble INP UGA.
During the course of my studies or on my personal time, I had several relevant experiences, including :

- Contribution to the development of Easytracker, a debugging library written in Python as a research intern, where I implemented the support of multi-threaded inferiors.
- Introductory and advanced language theory courses.
- A full-time compiler project during the entirety of last January, where we had to build from scratch a Deca compiler as a team of 5 students.
- A Master level course on formal proof and validation.
- Writing a patch for PR12341. Note that at the time of the writing, it has yet to be marked as fixed, as I asked for feedback on gcc-patches maillist. The patch correctly adds the warning and passes the regression tests, but my current implementation could use more of the already existing functions. Still, I have already gained hands-on experience from this, and familiarity with the GENERIC format and the testsuite.

I am also fluent in C++17, and I'm still upgrading to C++20. Moreover I am familiar with references such as *Efficient C++*, *the Dragon Book* or *Crafting Interpreters* (R. Nystrom).

My motivation comes from my interest in compiler and language theory, and finally contributing to meaningful open-source projects. I often wanted to undertake open-source development, yet I have always backpedaled when confronting the enormity of it. To me, GSoC represents the opportunity to confront this. My affinity towards compiler development led me to consider GCC. Even though I was utterly intimidated by the project at first, the community proved itself up to the task of supporting newbies. To work on the static analyzer would be a valuable learning experience for me, as with a friend we are currently trying to develop our own language. Its purpose would be to ease the learning of programming concepts (C is often used to introduce memory management, yet the ambiguity and complexity of C syntax are more of a hindrance than anything else). Hands-on experience in static analysis, with a compiler at the scale of GCC, would significantly improve that undertaking.

My Github and Gitlab accounts @vultkayn are mostly empty since I am mainly using a Gitlab private to my university.

Mentor

David Malcolm.

Related work

The memory model adopted in the analyzer is based on the work of [5]. The paper [6] is quoted in the analyzer internal documentation by David Malcolm, as a reference for the exploded graph model. However the actual implementation of the analyzer differs from the IFDS framework exposed in this paper, and is rather based on symbolic execution [7], as the problems are hardly distributive. Yet, the PHASAR framework [8] adopted by LLVM's analyzer could be a source of inspiration to improve the construction of call summaries, as actually they cannot be used to avoid duplicating exploded nodes, as the IDE framework would. Clang's static analyzer [9] currently better supports C++ than GCC's, and hence could be used as a comparison point.

Future work considered

- Support iteration over C++ containers.
- Support exception handling. Not tackled here as the analyzer is built without EH.
- Support use-after-move. This is well within the scope of enabling the self-analysis, however the project would be too large. Instead, I consider it as a future potential enhancement.
- Add support for dynamic_cast. I propose the following snippet as a first test case to introduce -Wanalyzer-rtti-dynamic-cast. This would also require acquiring RTTI knowledge.

Figure 6 - A first test case towards supporting dynamic_cast

```
class B {
public:
 virtual void foo () {}
};
class D : public B {};
class E {virtual void goo () {}};
void goo ()
{
  B* b = new D (); E* e = new E ();
  D* d = dynamic_cast<D*> (b); /* dg_bogus -Wanalyzer-rtti-dynamic-cast  */
  D* d2 = dynamic_cast<D*> (e); /* dg-warning -Wanalyzer-rtti-dynamic-cast */
  d2->goo (); /* dg-warning -Wanalyzer-null-dereference */
  delete b; delete e;
}
```

References

[1] analyzer/PR97110 Tracker bug for supporting C++ in -fanalyzer. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=97110.

[2] analyzer/PR109365 Double delete yields -Wanalyzer-use-after-free instead of -Wanalyzer-double-free. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=109365.

[3] GSoC 2021 : Extending gcc static analyzer to support virtual functions calls. https://gist.github.com/Arsenic-ATG/8f4ac194f460dd9b2c78cf51af39afef

[4] analyzer/PR109366 No -Wanalyzer-null-dereference for unique_ptr. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=109366

[5] A Memory Model for Static Analysis of C Programs. Zhongxing Xu, Ted Kremenek, and Jian Zhang, dx.doi.org/10.1007/978-3-642-16558-0_44

[6] Precise Interprocedural Dataflow Analysis via Graph Reachability. Thomas Reps, Susan Horwitz, and Mooly Sagiv.

[7] Symbolic Path Simulation in Path-Sensitive Dataflow Analysis. Hari Hampapuram, Yue Yang, and Manuvir Das, https://doi.org/10.1145/1108768.1108808

[8] PhASAR: An Inter-procedural Static Analysis Framework for C/C++. Philipp D. Schubert, Ben Hermann, and Eric Bodden, http://doi.org/10.1007/978-3-030-17465-1_22

[9] Clang Static Analyzer source code. https://github.com/llvm/llvm-project/tree/main/clang/include/clang/StaticAnalyzer