# LLVM Social #6

Berlin, 2017.02.23

# Mull meets Rust

Implementing mutation testing system for

Rust programming language

## Who is speaking?

- iOS developer at Uberchord Engineering GmbH
- 1 year of programming for LLVM in a spare time
- Interests: code quality, testing and generally everything that can help a human being to write a better code
- http://stanislaw.github.io/
- https://systemundertest.org/
- https://github.com/mull-project/mull

## What is Mutation Testing?

- Software testing technique
- Based on controlled fault-injection: source code is modified (mutated)
- Simulates mistakes that programmers often make, misuse of a software, hardware errors etc.
- "Unit testing on steroids": programmers often design their tests against most-known green and red cases so a lot of code is still not tested. Mutation testing gives additional feedback about how good the tests are.

```
void test() {
  assert(sum(5, 10) > 0);
}
```

```
int sum(int a, int b) {
  return a + b;
}

int sum_1(int a, int b) {
  return a * b;
}

int sum_2(int a, int b) {
  return a - b;
}
```

## Mutation Testing: raw algorithm

```
run_test(program, test);
mutant = mutate(program);
result = run_test(mutant, test)
if (result == Failed) {
  report_killed_mutant(mutant, test);
} else {
  report_survived_mutant(mutant, test);
}
```

**Killed mutant** means our test is good: it is able to detect a change in a code.

**Survived mutant** means our test is not able to detect a change in a code: we either need to improve existing test or write more tests to kill survived mutants.

`assert(sum(5, 10) > 0);` is the example of a shallow test: to kill the mutants we just need to make it `assert(sum(5, 10) == 15);`

See: "FOSDEM 2017: Mutation Testing, Leaving the Stone Age by Alex Denisov" for a proper introduction (links are on the last slide).

## What is Mull?

- Mutation testing system based on top of LLVM
- Potentially can work with any programming language that compiles to LLVM IR
- Already implemented:
    - C++ / Google Test
    - Rust (work in progress)
- Pending:
    - C++ / Catch
    - Objective-C
    - Swift

## How to make Mull support a programming language?

- Language must be LLVM-friendly
    - Should support compilation to LLVM IR: `*.bc` and `*.ll`
- Find tests
    - Each programming language has its own implementation details.
    - Algorithm depends on a test framework used: Google Test, XCTest, Rust's native test framework etc.
- Find testees (code that is tested), find mutation points in them
- Make tests run with LLVM JIT

## Good news #1: Rust is LLVM friendly

- Stable Rust is working against LLVM 3.9, people are working on LLVM 4.0 support. For contrast: Haskell is LLVM 3.7.
- Similar to C/C++: has `main()` function. Made `lli` to run "hello world" in 15 minutes.
- No sophisticated runtime magic: method names are method names, method calls do stuff out of the box. For contrast: Objective-C Runtime is currently not working properly in LLVM JIT.
- Started proof-of-concept with LLVM 4.0 API but failed, had to rollback to 3.9.

## Good news #2: Rust is LLVM bitcode friendly

- No CMake, no Ninja and compilation databases, no Bash magic
- `cargo test --verbose` to learn about compilation commands
- Just works:

```
compile: clean
    mkdir -p /Users/stanislaw/Projects/nom/target/debug/deps
    touch /Users/stanislaw/Projects/nom/target/debug/deps/nom-2045d976cde84170.d
    rustc -Z print-link-args \
          --crate-name nom src/lib.rs \
          --emit=dep-info,link,llvm-ir,llvm-bc \
          -C debuginfo=2 \
          --test \
          --cfg 'feature="default"' \
          --cfg 'feature="stream"' \
          -C metadata=2045d976cde84170 \
          -C extra-filename=-2045d976cde84170 \
          --out-dir /Users/stanislaw/Projects/nom/target/debug/deps \
          -L dependency=/Users/stanislaw/Projects/nom/target/debug/deps
```

## Finding tests: brute force?

"How do I find the function pointers for tests from the LLVM IR code of a Rust program?"

http://stackoverflow.com/questions/42177712/how-do-i-find-the-function-pointers-for-tests-from-the-llvm-ir-code-of-a-rust-pr

**TL;DR:** Should we use a brute-force search or is there anything better?

---

@matthieu-m:

> "I am not sure how you'll manage to do it, but I certainly wish you luck. Having mutation testing in Rust would be awesome."

## Finding tests: brute force!

[https://users.rust-lang.org/t/how-do-i-find-the-function-pointers-for-tests-from-the-llvm-ir-code-of-a-rust-program/9407](https://users.rust-lang.org/t/how-do-i-find-the-function-pointers-for-tests-from-the-llvm-ir-code-of-a-rust-program/9407)

@nagisa:

> "Why not just collect anything that looks like a function pointer to LLVM by exhaustively searching the whole structure passed into the function called test_main_static?
>
> In general I can not think of any approach that could work if the test suite is compiled with optimisations LLVM can and will inline even indirect calls just fine if the body of test_main_static ever gets its IR put into the same module as main."

## Finding tests: LLVM IR for Rust module with tests

```
void test1() {}
void test2() {}

// `tests` is a static array of structs
tests = [
  {
    name: "Test1",
    pointer: &test1
  },
  {
    name: "Test2",
    pointer: &test2
  },
  ...
]

void test_main() {
  // https://doc.rust-lang.org/1.1.0/test/fn.test_main_static.html
  test_main_static(&tests);
}

void main(int argc, int argv) {
  std::rt::lang_start(&test_main, argc, &argv);
}
```

## Finding tests: filtering a noise

Things we have to filter:

- Instructions without a debug information
- Instructions with a debug information that points to Rust sources

```
/Users/rustbuild/src/rust-buildbot/slave/stable-dist-rustc-mac/build/obj/../src/li
```

Solution: build Rust from sources.

- Call instructions that are calls to panic methods:

```
_ZN4core9panicking5panic17hd383cb12a44b01ffE
```

## Finding tests: filtering a noise #2

```
class RustTestMutationOperatorFilter : public MutationOperatorFilter {
public:
  bool shouldSkipInstruction(llvm::Instruction *instruction) {
    if (CallInst *callInst = dyn_cast<CallInst>(instruction)) {
      if (Function *calledFunction = callInst->getCalledFunction()) {
        if (calledFunction->getName().find("panic") != std::string::npos) {
          return true;
        }
      }
    }

    if (instruction->hasMetadata() == false) {
      return true;
    }

    int debugInfoKindID = 0;
    MDNode *debug = instruction->getMetadata(debugInfoKindID);

    DILocation *location = dyn_cast<DILocation>(debug);
    if (location) {
      if (location->getFilename().str().find("buildbot") != std::string::npos) {
        return true;
      }
    }

    return false;
  };
};
```

## Filtering a noise in C++ / Google Test

```cpp
class GoogleTestMutationOperatorFilter : public MutationOperatorFilter {
public:
  bool shouldSkipInstruction(llvm::Instruction *instruction) {
    if (instruction->hasMetadata()) {
      int debugInfoKindID = 0;
      MDNode *debug = instruction->getMetadata(debugInfoKindID);

      DILocation *location = dyn_cast<DILocation>(debug);
      if (location) {
        if (location->getFilename().str().find("include/c++/v1") != std::string::n
          return true;
        }
      }
    }

    return false;
  };
};
```

## Running tests: normal flow of Rust

```
tests = [{...}];

void test_main() {
  // https://doc.rust-lang.org/1.1.0/test/fn.test_main_static.html
  test_main_static(&tests);
}

void main(int argc, int argv) {
  std::rt::lang_start(&test_main, argc, &argv);
}
```

- If tests pass program exists with `0`
- Program exits with `101` if at least one test fails
- Internally in each test: every failing assertion `panics`
- `lang_start` creates a new thread, wraps tests in an exception handler, waits for their execution and exits with `0` on success or `101` on failure.

## Running tests with Mull

```
tests = [{...}];

void test_main() {
  // https://doc.rust-lang.org/1.1.0/test/fn.test_main_static.html
  test_main_static(&tests);
}

void main(int argc, int argv) {
  std::rt::lang_start(&test_main, argc, &argv);
}
```

- `main` is not run
- `test_main` is also not run, but is used to find the pointers to test functions
- For each test Mull creates a separate process using **fork** and runs this test's function directly:
  - If a test returns normally, it is considered "passed".
  - If a test panics with `assertion failed` or with any other reason, the test is considered "failed".
- Mull's parent process considers an exit code and signal handlers of the child process and saves a test execution result as "passed" or "failed".

## Add Mutation Operator

Addition is represented by @llvm.add.with.overflow calls.

```
bool AddMutationOperator::isAddWithOverflow(llvm::Value &V) {
  if (CallInst *callInst = dyn_cast<CallInst>(&V)) {
    Function *calledFunction = callInst->getCalledFunction();

    if (calledFunction == nullptr) {
      return false;
    }

    if (calledFunction->getName().startswith("llvm.sadd") ||
        calledFunction->getName().startswith("llvm.uadd")) {
      return true;
    }
  }

  return false;
}
```

LLVM Language Reference Manual - Arithmetic with Overflow Intrinsics:

http://llvm.org/docs/LangRef.html#arithmetic-with-overflow-intrinsics

## Add Mutation Operator: Example

Replace:

```
c = a + b;
```

With:

```
c = a - b;
```

Expect:

```
thread 'main' panicked at 'assertion failed: sum(a, b) == 4', example.rs:27
```

Instead:

```
thread '<unnamed>' panicked at 'attempt to add with overflow', src/lib.rs:117
note: Run with `RUST_BACKTRACE=1` for a backtrace. fatal runtime error:
failed to initiate panic, error 5
```

## Add Mutation Operator: Example (continued)

Under the hood:

```
%136 = call { i64, i1 } @llvm.uadd.with.overflow.i64(i64 %131, i64 %135), !dbg !27
```

In context:

```
bb50: ; preds = %bb48
%134 = load i32, i32* %c, !dbg !274, !range !272
%135 = zext i32 %134 to i64, !dbg !274
%136 = call { i64, i1 } @llvm.uadd.with.overflow.i64(i64 %131, i64 %135), !dbg !27
%137 = extractvalue { i64, i1 } %136, 0, !dbg !274
%138 = extractvalue { i64, i1 } %136, 1, !dbg !274
%139 = call i1 @llvm.expect.i1(i1 %138, i1 false), !dbg !274
br i1 %139, label %panic4, label %bb51, !dbg !274
```

## Demo: rustc-demangle

https://github.com/alexcrichton/rustc-demangle (Rust symbol demangling)

## TODO #1

### Compilation pipeline

Goal: for any given Rust project get its LLVM IR easily, both code and tests.

- So far we are only testing unit tests that are embedded in src/* files:
  - rustc produces only one .bc file - re-compilation is slow
  - having a program and its tests in one module is convenient, but it would be great to decompose it into the smaller chunks.
- Learn how to generate bitcode for the tests from `tests/*`.

### Optimizations

Learn more about what optimizations are done. Learn how to disable them.

Example: disable add with overflow checks.

## TODO #2

### Rust symbols demangling

Needed to make reports more readable so that

```
_ZN8demangle5tests8demangle17h4d833f55bb2fea15E
_ZN8demangle5tests16demangle_dollars17h654ff4516cb2a88eE
_ZN8demangle5tests21demangle_many_dollars17h929844db6571e81cE
```

becomes:

```
demangle::tests::demangle
demangle::tests::demangle_dollars
demangle::tests::demangle_many_dollars
```

Direct port of [https://github.com/alexcrichton/rustc-demangle](https://github.com/alexcrichton/rustc-demangle) to C++ will do.

## TODO #3: Analyze

### More mutation operators

Currently 3 operators are supported:

- Add
- Remove Void Function Call
- Negate Condition

Many more operators can be added.

### In the wild

- Analyze a reasonable number of real world projects
- Filter out more noise
- Learn more about mutations

## Open questions

These are the questions we want to answer with Mull eventually.

- Can mutation testing actually detect serious errors in the real-world programs?
- What are the most effective mutation operators?
- Compare mutation testing with other methods like static analysis, fuzz testing, etc.
- Hot topic: can mutation testing help to find errors that a normal testing with a code coverage cannot find?

## Links

- Mull project: https://github.com/mull-project/mull
- "FOSDEM 2017: Mutation Testing, Leaving the Stone Age" by Alex Denisov https://www.youtube.com/watch?v=YEgiyiICkpQ
- "Are Mutants a Valid Substitute for Real Faults in Software Testing?" https://homes.cs.washington.edu/%7Emernst/pubs/mutation-effectiveness-fse2014.pdf

## Questions?

## Thanks to co.up Coworking for hosting us

[http://co-up.de/](http://co-up.de/)