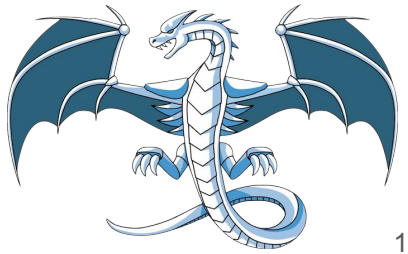
Recovering from Errors in Clang-Repl and Code Undo

Authors: Purva Chaudhari, Jun Zhang
Mentor: Dr Vassil Vassilev, Dr David Lange

https://compiler-research.org/



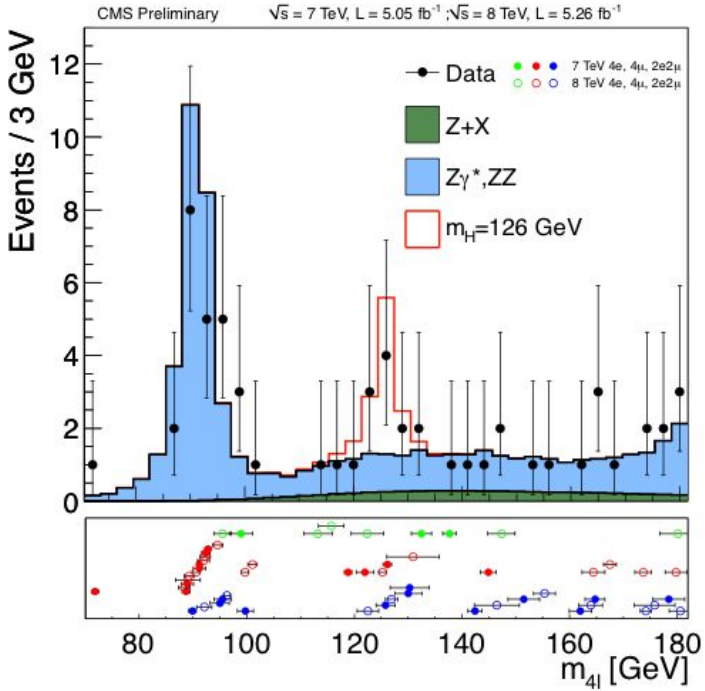
ROOT

Data Analysis Framework

ROOT is a set of OO frameworks developed by high-energy physics (HEP) which is used to handle and analyze large amounts of data in a very efficient way.

```
❱ bin/cling

****************** CLING ******************
* Type C++ code and press enter to run it *
*             Type .q to exit             *
*******************************************
[cling]$ #include <iostream>
[cling]$ auto foo = []() {
[cling]$ ?   std::cout << "Hello, world!\n";
[cling]$ ?   };
[cling]$ foo
((lambda) &) @0x7f09b3754000
[cling]$ foo();
Hello, world!
[cling]$
```



The core part of ROOT is the Cling Interpreter, which built on top of Clang and LLVM compiler technology. It realizes the read-eval-print loop (REPL) concept, in order to leverage rapid application development.

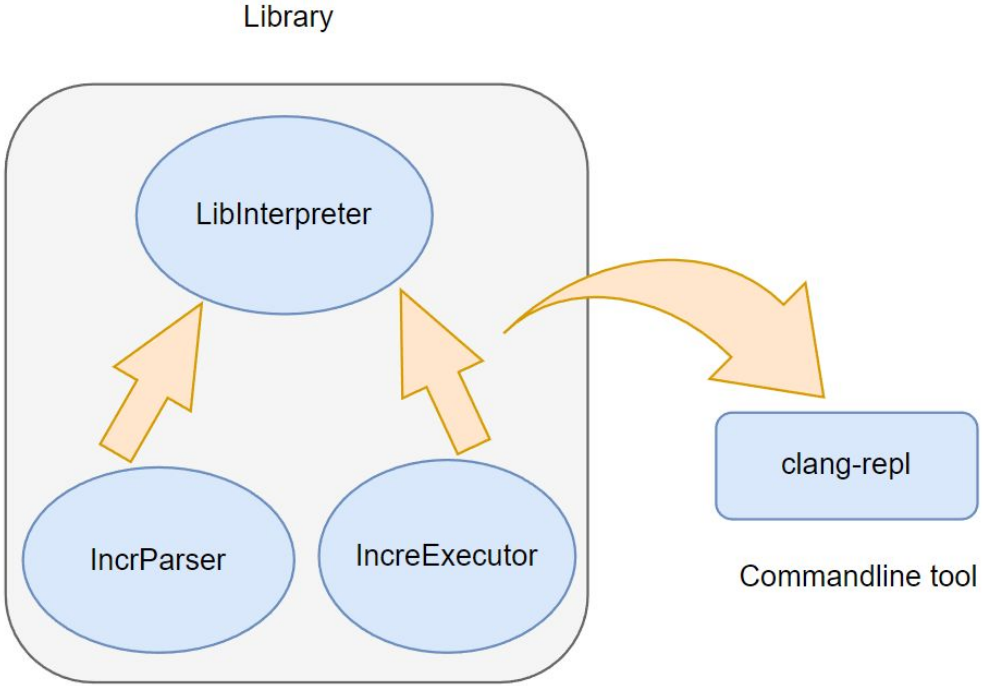
Clang-Repl Overview

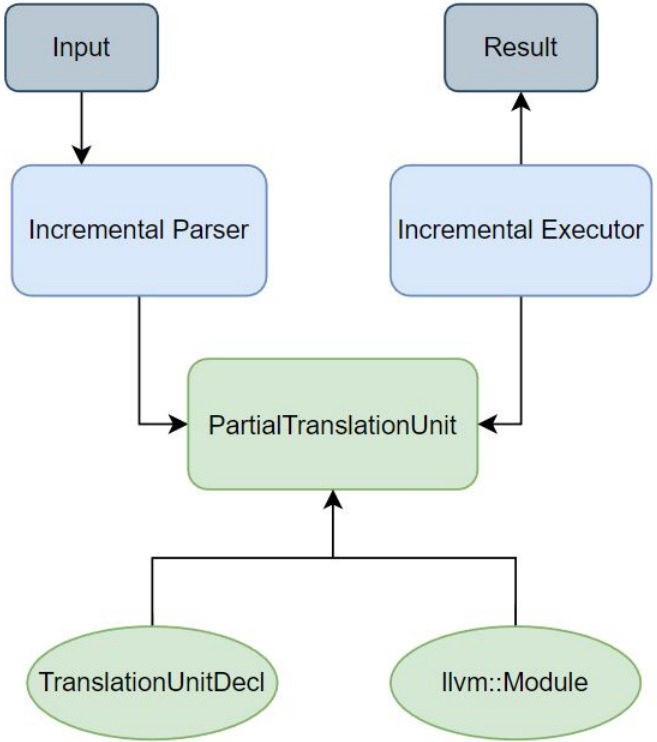
Clang-Repl is a new tool which incorporates Cling in the Clang mainline.

```
❱ bin/clang-repl
clang-repl> int x = 42;
clang-repl> extern "C" int printf(const char*,...);
clang-repl> auto r = printf("%d\n",x);
42
clang-repl>
```

- [llvm-dev] [RFC] Moving (parts of) the Cling REPL in Clang *Vassil Vassilev via llvm-dev*
  - [llvm-dev] [cfe-dev] [RFC] Moving (parts of) the Cling REPL in Clang *Chris Lattner via llvm-dev*
  - [llvm-dev] [cfe-dev] [RFC] Moving (parts of) the Cling REPL in Clang *Hal Finkel via llvm-dev*
    - [llvm-dev] [cfe-dev] [RFC] Moving (parts of) the Cling REPL in Clang *JF Bastien via llvm-dev*
      - [llvm-dev] [cfe-dev] [RFC] Moving (parts of) the Cling REPL in Clang *David Rector via llvm-dev*
      - [llvm-dev] [cfe-dev] [RFC] Moving (parts of) the Cling REPL in Clang *Vassil Vassilev via llvm-dev*
      - [llvm-dev] [cfe-dev] [RFC] Moving (parts of) the Cling REPL in Clang *Hal Finkel via llvm-dev*
      - [llvm-dev] [cfe-dev] [RFC] Moving (parts of) the Cling REPL in Clang *JF Bastien via llvm-dev*
      - [llvm-dev] [cfe-dev] [RFC] Moving (parts of) the Cling REPL in Clang *Vassil Vassilev via llvm-dev*
      - [llvm-dev] [cfe-dev] [RFC] Moving (parts of) the Cling REPL in Clang *Hal Finkel via llvm-dev*
      - [llvm-dev] [cfe-dev] [RFC] Moving (parts of) the Cling REPL in Clang *Vassil Vassilev via llvm-dev*
  - [llvm-dev] [cfe-dev] [RFC] Moving (parts of) the Cling REPL in Clang *Richard Smith via llvm-dev*
    - [llvm-dev] [cfe-dev] [RFC] Moving (parts of) the Cling REPL in Clang *Vassil Vassilev via llvm-dev*
      - [llvm-dev] [cfe-dev] [RFC] Moving (parts of) the Cling REPL in Clang *Richard Smith via llvm-dev*
      - [llvm-dev] [cfe-dev] [RFC] Moving (parts of) the Cling REPL in Clang *Chris Lattner via llvm-dev*
      - [llvm-dev] [cfe-dev] [RFC] Moving (parts of) the Cling REPL in Clang *Vassil Vassilev via llvm-dev*
      - [llvm-dev] [cfe-dev] [RFC] Moving (parts of) the Cling REPL in Clang *Vassil Vassilev via llvm-dev*
      - [llvm-dev] [cfe-dev] [RFC] Moving (parts of) the Cling REPL in Clang *Raphael “Teemperor” Isemann via llvm-dev*

Code Infrastructure & Pipeline





Incremental Parser

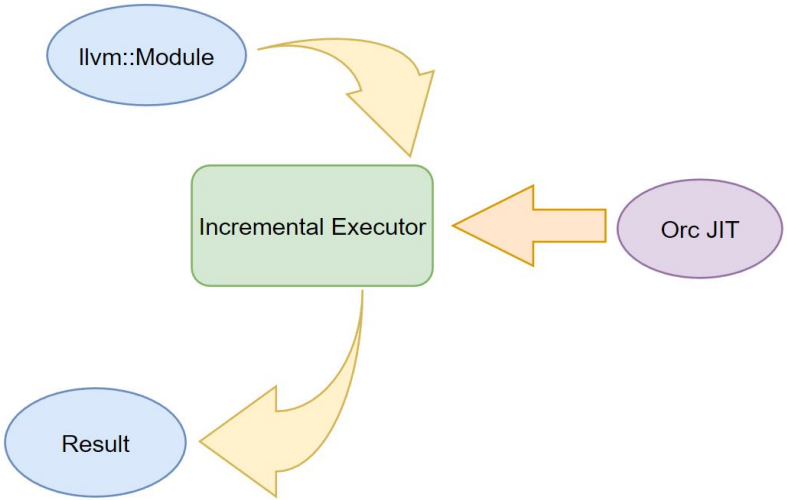
In the interactive C++, the parsing phase is a bit different from traditional C++ execution.

Because the input is incremental and there's real no source file exists, we manually create a memory buffer that take the input and feed it to the SourceManager. 😈

```
227
228     // Create an uninitialized memory buffer, copy code in and append "\n"
229     size_t InputSize = input.size(); // don't include trailing 0
230     // MemBuffer size should *not* include terminating zero
231     std::unique_ptr<llvm::MemoryBuffer> MB(
232         llvm::WritableMemoryBuffer::getNewUninitMemBuffer(InputSize + 1,
233                                                           SourceName.str()));
234     char *MBStart = const_cast<char *>(MB->getBufferStart());
235     memcpy(MBStart, input.data(), InputSize);
236     MBStart[InputSize] = '\n';
237
```

Incremental Executor

Thanks to the great work of LLVM folks, the incremental executor is completely powered by the LLVM Orc JIT and get all performance for free 😻



Error Recovery In Clang-Repl

- Translation unit in Clang can be split into a sequence of partial translation units (PTUs)
- Owning PTU is not always the most recent PTU and processing a PTU might extend an earlier PTU.
- Clang-repl recovers from errors by disconnecting the most recent PTU and update the primary PTU lookup tables

```
clang-repl> int i = 12; error;
In file included from <<< inputs >>>:1:
input_line_0:1:13: error: C++ requires a type specifier for all
declarations
int i = 12; error;
            ^
error: Parsing failed.
```

Ref: Vassil V. Commit - Implement partial translation units and error recovery.

1. Template Recovery

- ➢ Patch added support for template recovery which was previously aborting the interactive mode in case of error encountered
- ➢ Done by declaring a Sema Class for performing the pending instantiations in the destructor

```
clang-repl> template<class T> T f() { return T(); }
clang-repl> auto ptu2 = f<float>(); err;
In file included from <<< inputs >>>:1:
input_line_1:1:25: error: C++ requires a type specifier for all
declarations
auto ptu2 = f<float>(); err;
                        ^
clang-repl: /home/purva/llvm-project/clang/include/clang/Sema/Sema.h:9406:
clang::Sema::GlobalEagerInstantiationScope::~GlobalEagerInstantiationScope(
): Assertion `S.PendingInstantiations.empty() && "PendingInstantiations
should be empty before it is discarded."' failed.
Aborted
 (core dumped)
```

Before

```
clang-repl> template<class T> T f() { return T(); }
clang-repl> auto ptu2 = f<float>(); err;
In file included from <<< inputs >>>:1:
input_line_1:1:25: error: C++ requires a type specifier for all
declarations
auto ptu2 = f<float>(); err;

clang-repl> auto ptu2 = f<float>();
```

After

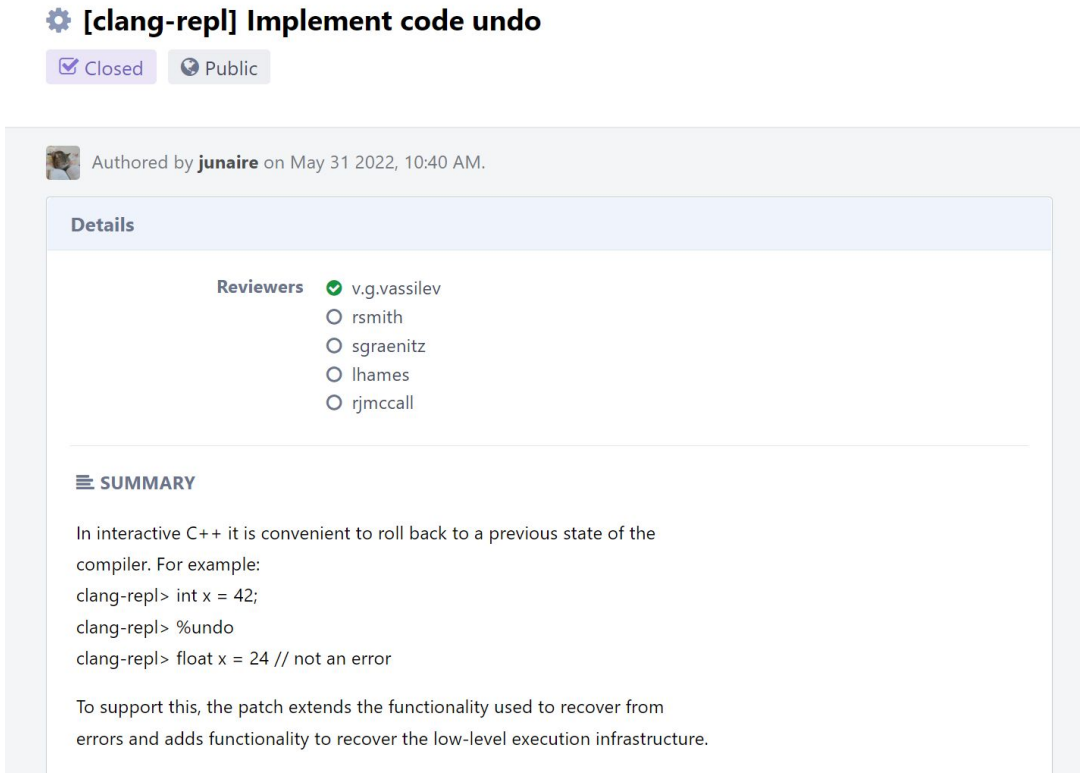
2. Undo Support

```
❯ bin/clang-repl
clang-repl> extern "C" int printf(const char*, ...);
clang-repl> int x = 42;
clang-repl> %undo
clang-repl> const char* x = "Hello, world!"; // It compiles!
clang-repl> auto r = printf("%s\n",x);
Hello, world!
clang-repl> %quit
```

- In interactive C++ it is convenient to roll back to a previous state of the compiler
- The patch extends the functionality used to recover from errors and adds functionality to recover the low-level execution infrastructure.
- The current implementation is based on watermarks.

Internals about code undo

- Erase the most recent element in the PartialTranslationUnit list
- Kill the LLVM module in JIT
- Let the Parser clean the state



Ultimate goal for Clang-Repl 😺

Currently clang-repl is still in the early stage and we’re continue working on it.

In the future, we want to export it as a production ready library so users like Cling can use it directly.

Thank You 

https://compiler-research.org/