Handle Execution results in clang-repl

Jun Zhang
jun@junz.org

Mentors:
Vassil Vassilev & David Lange

About me

I’m Jun Zhang, 3rd year undergraduate student major in Software Engineering.

Working in the compiler research team since May 2022.

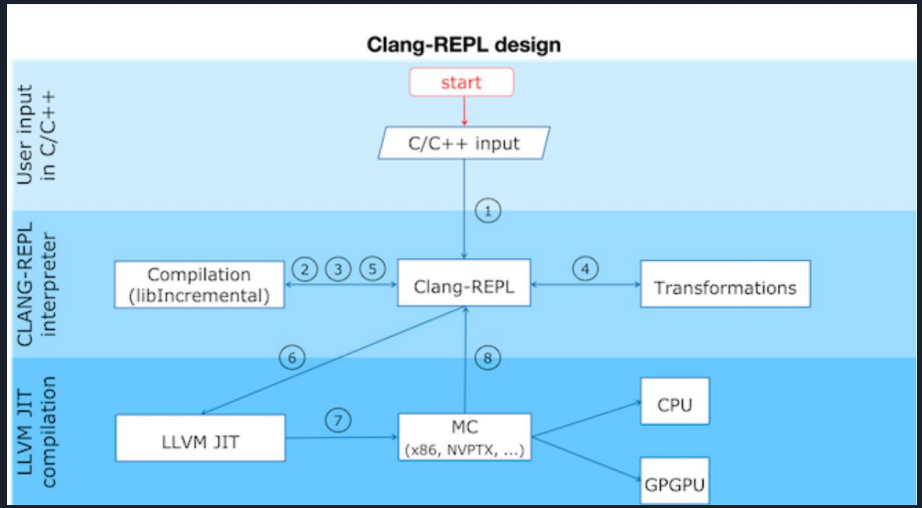
Last year Google Summer of Code student, Clang/LLVM contributor. (Land ~70 patches)

Project Background

An introduction to clang-repl

clang-repl is an interactive C++ interpreter that allows for incremental compilation, based on Clang and LLVM Orc JIT.

```
clang-repl> #include <iostream>
clang-repl> std::cout << “Hello, world!\n”;
Hello,world!
```



Project Goals

Value pretty printing

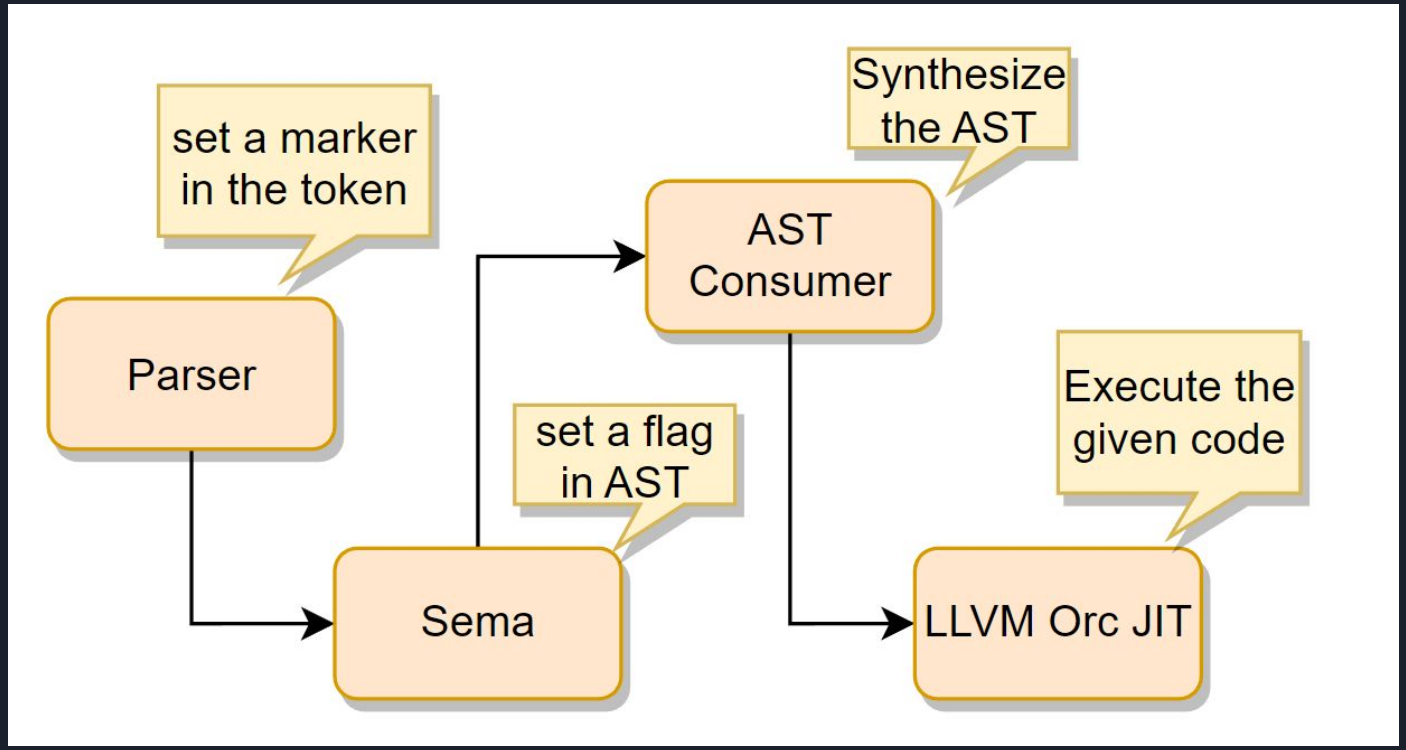
```
clang-repl> int x = 42;

clang-repl> x

(int) 42

clang-repl> std::vector<int> v{1,2,3};

clang-repl> v

(std::vector<int> &) { 1, 2, 3 }
```

Compiled/Interpreted code interop

```
int Global = 42;

void setGlobal(int val) { Global = val; }

int getGlobal() { return Global; }

Interp.ParseAndExecute(“void setGlobal(int val);”);

Interp.ParseAndExecute(“int getGlobal();”);

Value V;

Interp.ParseAndExecute(“getGlobal()”, &V);

std::cout << V.getAs<int>() << “\n”; // Prints 42
```

The implementation (Highly inspired by Cling)

Overview


set a marker in the token
Synthesize the AST
AST Consumer
Parser
set a flag in AST
Execute the given code
Sema
LLVM Orc JIT

Top level expressions extension

```
clang-repl> int x = 42;
clang-repl> x // Missing semicolon.
```

Invalid in standard C++, but fine in the incremental C++.

Omit the semi tells clang-repl we want to capture the value of the DeclRefExpr (x)

Annotation token: annot_repl_input_end

```
945  + // Annotation for end of input in clang-repl.
946  + ANNOTATION(repl_input_end)
947  +
```

When parsing an ExprStmt and the last semi is missing, we'll pretend that there's one and set a marker for the late use.

```
547       Token *CurTok = nullptr;
548       // If the semicolon is missing at the end of REPL input, consider if
549       // we want to do value printing. Note this is only enabled in C++ mode
550       // since part of the implementation requires C++ language features.
551       // Note we shouldn't eat the token since the callback needs it.
552       if (Tok.is(tok::annot_repl_input_end) && Actions.getLangOpts().CPlusPlus)
553         CurTok = &Tok;
554       else
555         // Otherwise, eat the semicolon.
556         ExpectAndConsumeSemi(diag::err_expected_semi_after_expr);
557
558       StmtResult R = handleExprStmt(Expr, StmtCtx);
559       if (CurTok && !R.isInvalid())
560         CurTok->setAnnotationValue(R.get());
561
562       return R;
563     }
```

AST transformation

```
5460 +
5461 +   if (Tok.is(tok::annot_repl_input_end) &&
5462 +       Tok.getAnnotationValue() != nullptr) {
5463 +     ConsumeAnnotationToken();
5464 +     cast<TopLevelStmtDecl>(DeclsInGroup.back())->setSemiMissing();
5465 +   }
5466 +
```

Sema sets a flag after seeing the special token, so we know we should transform the AST before the real CodeGen process.

```
41 +   bool HandleTopLevelDecl(DeclGroupRef DGR) override final {
42 +     if (DGR.isNull())
43 +       return true;
44 +     if (!Consumer)
45 +       return true;
46 +
47 +     for (Decl *D : DGR)
48 +       if (auto *TSD = llvm::dyn_cast<TopLevelStmtDecl>(D);
49 +           TSD && TSD->isSemiMissing())
50 +         TSD->setStmt(Interp.SynthesizeExpr(cast<Expr>(TSD->getStmt())));
51 +
52 +     return Consumer->HandleTopLevelDecl(DGR);
53 +   }
```

Code synthesis

clang-repl> x → __clang_Interpreter_SetValue(x); // pseduo code.

DeclRefExpr → CallExpr

```
+ // This synthesizes a call expression to a speciall
+ // function that is responsible for generating the Value.
+ // In general, we transform:
+ //   clang-repl> x
+ // To:
+ //   // 1. If x is a built-in type like int, float.
+ //   __clang_Interpreter_SetValueNoAlloc(ThisInterp, OpaqueValue, xQualType, x);
+ //   // 2. If x is a struct, and a lvalue.
+ //   __clang_Interpreter_SetValueNoAlloc(ThisInterp, OpaqueValue, xQualType,
+ //   &x);
+ //   // 3. If x is a struct, but a rvalue.
+ //   new (__clang_Interpreter_SetValueWithAlloc(ThisInterp, OpaqueValue,
+ //   xQualType)) (x);
+
+ Expr *Interpreter::SynthesizeExpr(Expr *E) {
```

Value runtime

A value is a container that can carry the arbitrary result of an expression in an endian-independent way with small buffer optimization.

```
class Value {
public:
clang::QualType* getType(); // Obtain the type information of the expression.
template<typename T>
T castAs(); // Cast the value to corresponding type.

void printType(llvm::raw_ostream& OS);

void printData(llvm::raw_ostream& OS);

void print(llvm::raw_ostream& OS);
void dump() const; // Dump the value, called print(llvm::outs()) internally.
};
```

The value that holds the information of the expression can be passed around after construction.

Pretty print implementation

If the user asks for the Value, we pass it as output parameter.

Or we perform pretty printing: Invoke Value::dump()

```
306 + llvm::Error Interpreter::ParseAndExecute(llvm::StringRef Code, Value *V) {
307 +
308 +   auto PTU = Parse(Code);
309 +   if (!PTU)
310 +     return PTU.takeError();
311 +   if (PTU->TheModule)
312 +     if (llvm::Error Err = Execute(*PTU))
313 +       return Err;
314 +
315 +   if (LastValue.isValid()) {
316 +     if (!V) {
317 +       LastValue.dump();
318 +       LastValue.clear();
319 +     } else
320 +       *V = std::move(LastValue);
321 +   }
322 +   return llvm::Error::success();
323 + }
324 +
```

Implementation of Value::dump

If the type is a builtin type:

Just print it directly.

Else:

Synthesize a call to another runtime function: PrintValueRuntime(const T*)

```
519 +   } else {
520 +     // All fails then generate a runtime call, this is slow.
521 +     SS << SynthesizeRuntimePrint(V);
522 +   }
523 +   return Str;
```

std::string PrintValueRuntime(const T*)

All overloads live in a header, which are included at runtime.

So print a std::vector is equivalent to:
PrintValueRuntime(&v);

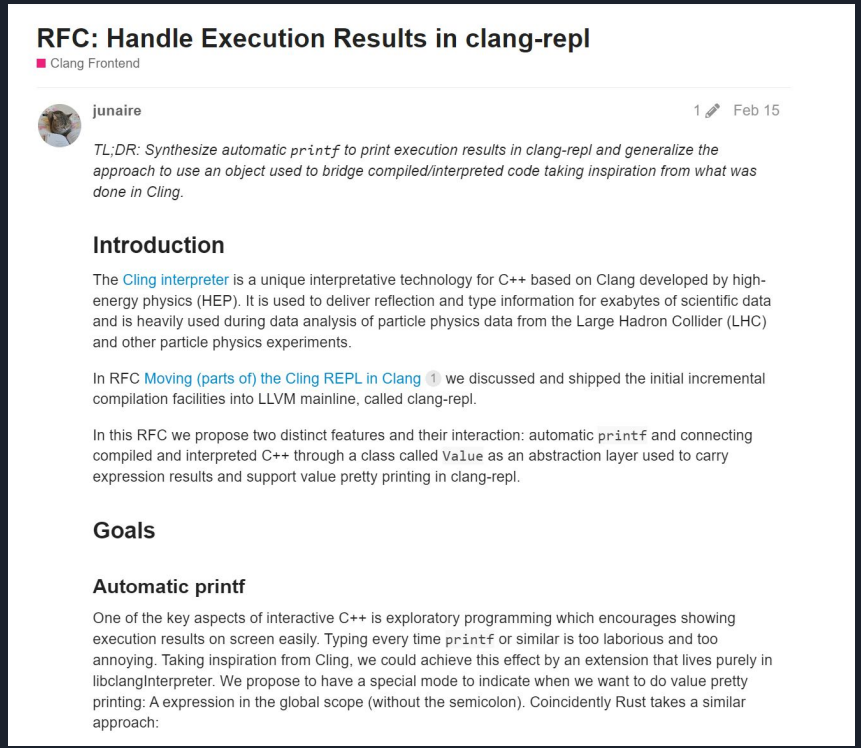
```
271 + static std::string SynthesizeRuntimePrint(const Value &V) {
272 +   Interpreter &Interp = const_cast<Interpreter &>(V.getInterpreter());
273 +   Sema &S = Interp.getCompilerInstance()->getSema();
274 +   ASTContext &Ctx = S.getASTContext();
275 +
276 +   // Only include this header once and on demand. Because it's very heavy.
277 +   static bool Included = false;
278 +   if (!Included) {
279 +     Included = true;
280 +     llvm::cantFail(
281 +         Interp.Parse("#include <__clang_interpreter_runtime_printvalue.h>"));
```

This means users can write their own overload for their types:

clang-repl> struct S{};
clang-repl> std::string PrintValueRuntime(const S* s) { return “My printer!”; }
clang-repl> S{}
(S) “My Printer!”

How we make it

Submit the RFC



RFC: Handle Execution Results in clang-repl

Clang Frontend

junaire 1 Feb 15

TL;DR: Synthesize automatic `printf` to print execution results in clang-repl and generalize the approach to use an object used to bridge compiled/interpreted code taking inspiration from what was done in Cling.

Introduction

The Cling interpreter is a unique interpretative technology for C++ based on Clang developed by high-energy physics (HEP). It is used to deliver reflection and type information for exabytes of scientific data and is heavily used during data analysis of particle physics data from the Large Hadron Collider (LHC) and other particle physics experiments.

In RFC Moving (parts of) the Cling REPL in Clang 1 we discussed and shipped the initial incremental compilation facilities into LLVM mainline, called clang-repl.

In this RFC we propose two distinct features and their interaction: automatic `printf` and connecting compiled and interpreted C++ through a class called `Value` as an abstraction layer used to carry expression results and support value pretty printing in clang-repl.

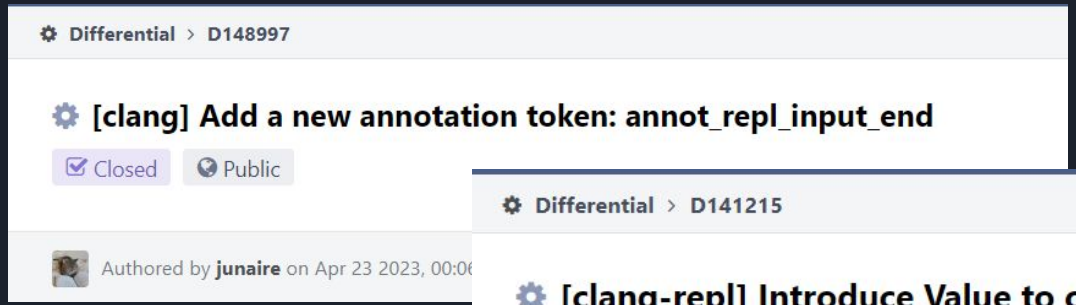
Goals

Automatic printf

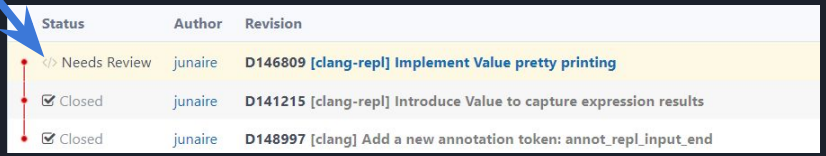
One of the key aspects of interactive C++ is exploratory programming which encourages showing execution results on screen easily. Typing every time `printf` or similar is too laborious and too annoying. Taking inspiration from Cling, we could achieve this effect by an extension that lives purely in libclangInterpreter. We propose to have a special mode to indicate when we want to do value pretty printing: A expression in the global scope (without the semicolon). Coincidently Rust takes a similar approach:

Spend almost one month writing the RFC and discussing the implementation with the community

Submit patches to Phabricator



Made 2 patches in but still one left!



What I learned

Large patches are hard to get it in!

Crazy long revision history!

Diff 1	508073	1c18b17	
Diff 2	509306	f8d8cb5	Update + Rebase
Diff 3	510168	0c983b8	Update
Diff 4	510288	08aad49	CHANGELOG:
Diff 5	511871	457bf61	Rebase
Diff 6	513024	d40a321	Update
Diff 7	513927	1494a32	Sync
Diff 8	513971	a08b090	Rebase
Diff 9	514160	a08b090	Update
Diff 10	517836	3ad0351	Update
Diff 11	518266	2575ad3	.
Diff 12	518287	fcdedcf	.
Diff 13	518289	fcdedcf	.
Diff 14	518458	84312e2	Fix AutoType sugar
Diff 15	520912	11f405a	Fix
Diff 16	520913	11f405a	Remove unused code.
Diff 17	520914	11f405a	Add some comments
Diff 18	520998	11f405a	Export symbols in Windows.
Diff 19	521961	11f405a	Don't use C++17 because Clang on Windows is not default to that :(
Diff 20	521962	11f405a	Add macro guard.
Diff 21	521981	11f405a	* include <tuple> to avoid incorrect lookup on Windows
Diff 22	521985	bf4ea60	Make and use our own std::void_t
Diff 23	524088	d20afbd	.
Diff 24	524091	d20afbd	.
Diff 25	524093	d20afbd	add `caas` namespace
Diff 26	524096	d20afbd	.
Diff 27	525420	cf1ef41	Address comments.
Diff 28	525425	cf1ef41	Remove `Interpereter::getParser` + More clean up

Diff 31	510286	3b24769	Fix known memory issues.
Diff 32	510885	4110934	Update
Diff 33	511084	4110934	Add IncrementalASTConsumer
Diff 34	511090	4110934	Remove old code
Diff 35	511857	4f8f01b	Update...
Diff 36	511862	4f8f01b	Clean up the patch
Diff 37	511867	4f8f01b	Address comments
Diff 38	511870	4f8f01b	Remove some unnecessary code.
Diff 39	511984	4f8f01b	Refactor how we compile a destructor
Diff 40	511999	4f8f01b	Address some comments
Diff 41	512072	4f8f01b	Address some comments
Diff 42	512073	4f8f01b	fix some nits
Diff 43	512075	4f8f01b	Use IncrementalParser::GetMangledName instead
Diff 44	512801	915a45c	use unsigned long instead of std::size_t when including the runtin
Diff 45	512837	915a45c	dont include <new> if we don't have it
Diff 46	512869	915a45c	fix ci
Diff 47	513918	ca4fb46	Address some comments
Diff 48	513926	ca4fb46	Address more comments
Diff 49	513954	fe1f344	Rebase
Diff 50	516080	f99b38b	Break the IsSemiMissing/ repl_input_end parts out from the Value
Diff 51	516081	f99b38b	Rebuild, please.
Diff 52	517829	b12c23b	fix windows buildbots
Diff 53	518166	b12c23b	Partially address some comments
Diff 54	518265	b12c23b	Add unittests for void & member pointers types
Diff 55	518285	957d723	Only enable __clang_Interpreter_SetValueCopyArr support in C++
Diff 56	518290	957d723	.
Diff 57	518292	957d723	.
Diff 58	518450	1f6370b	.
Diff 59	518952	cffd417	Address comments from @aaron.ballman, thanks!
Diff 60	518960	cffd417	Add more comments.
Diff 61	518961	cffd417	Fix typos
Diff 62	520324	743ff9c	Rebase + Update
Diff 63	520326	7cd10e4	.
Diff 64	522527	5558d54	Add comments
Diff 65	522530	5558d54	Address comment from Vassil, thx
Diff 66	522545	2d1cacd	remove whitespace

Writing portable code is hard!

aosp-O3-polly-before-vectorizer-unprofitable	10752
openmp-clang-x86_64-linux-debian	37790
clang-hexagon-elf	11870
llvm-clang-win-x-armv7l	12097
sanitizer-x86_64-linux-autoconf	37728
ppc64le-flang-rhel-clang	70187
sanitizer-aarch64-linux-bootstrap-msan	2731
llvm-clang-win-x-aarch64	13329

To revert, or not to revert, it's a question…

Thank you!

Special thanks to:

Aaron Ballman
Axel Naumann
Lang Hames
Richard Smith
Stefan Gränitz