GSOC 2023

Autocompletion in Clang-REPL

Yuquan (Fred) Fu (yuqfu@iu.edu)

Mentor: Vassil Vassilev

Introduction

Inspired by Cling, an LLVM/Clang-based interpreter developed for the scientific data analysis framework ROOT, Clang-REPL is a C++ interpreter built upon Clang and LLVM incremental compilation pipelines. It features a REPL, i.e., read-eval-print-loop, so that developers can program in C++ interactively.

Currently, the input support of Clang-REPL is primitive, and advanced editing features are lacking. One of them is Auto-completion. Users need to type every symbol of an expression or statement. The process is tedious and error-prone.

```
[clang-repl] class HelloMyFirstClassThatHasAReallyLongName{}
[clang-repl] new HelloMyFirstClassThatHasAReallyLongName()
```

For the purpose of demonstration, in the code above, we first define a class whose name has thirty-nine letters. We need all thirty-nine keystrokes to invoke the constructor to create an instance. We could resort to copy-and-paste, but it would break the coding workflow.

Our goal is to develop robust auto-completion in Clang-REPL with upstream components from Cling.

Motivation

Auto-completion is a nice-to-have, or even must-to-have, feature in modern day-to-day software development. Support of robust auto-completion in Clang-REPL will be a step forward in exploratory programming in C++.

First, it will assist users in avoiding laborious typing that is likely to lead to accidental typos. For example,

```
[clang-repl] class HelloMyFirstClassThatHasAReallyLongName{}
[clang-repl] new H_
// _ denotes the cursor
```

Instead of typing all the rest of thirty-eight letters, the user can press <tab>, and then a list of candidates including `HelloMyFirstClassThatHasAReallyLongName` will show up.

Secondly, the auto-completion should only provide well-typed candidates. Let us take a look at an example.

```
[clang-repl] struct Pear{int m;};
[clang-repl] struct Apple{ void foo(Apple& other){}};
[clang-repl] Pear c, d;
[clang-repl] Apple a, b;
[clang-repl] a.foo(_);
// _ denotes the cursor
```

When the user press <tab>, a naive implementation would provide all bound identifiers in the current namespace. However, since `a.foo` is a method whose parameter type is Apple& , the candidates should be narrowed down to a and b.

Furthermore, candidate filtering should respect subtyping relations. In the following example, we define a structure named Car, its superclass Vehicle, and its subclass RedApple.

```
[clang-repl] struct Vehicle{};
[clang-repl] struct Car : Vehicle{void crash(Car& other){}};
[clang-repl] struct Sedan : Car{};
[clang-repl] Vehicle v;
[clang-repl] Car c1, c2;
[clang-repl] Sedan s;
[clang-repl] c.move(_);
// _ denotes the cursor
```

When <tab> is pressed, the list of candidates that pops up should include only c1, c2, and s. Lastly, the auto-completion should take scopes into consideration. Let us continue our example with Pear and Apple.

```
[clang-repl] void bar(Apple& a1, Apple& a2){ some.foo(_)}
[clang-repl] b.foo(_)
```

At the first completion site, the candidates should include a1 and a2 along with a and b. On the contrary, only a and b will be provided at the next completion site.

Implementation Plan

- Port necessary modules, classes, and functions from cling/lib/UserInterface/textinput/ to Clang-REPL to handle the <tab> key event.
- Port lib/Interpreter/ClingCodeCompleteConsumer from Cling to Clang-REPL
- Design and Implement a prototype of type-directed Completion in Clang-REPL
    - Implement a class that serves a type environment to keep track of the bindings and their types.
        - This includes all classes, their methods, functions, and global variables imported via #include
        - All the classes, their methods, functions, and variables created in an ongoing REPL session.
    - Add a candidate filter based on type information from a type environment.

Timeline

Week 1 (Coding Begins):

Start porting Cling's auto-completion infrastructure to Clang-REPL.
Deliverables:

1. Port Keybinding.[h/cpp], StreamReader.[h/cpp], StreamReaderUnix.[h/cpp], Callbacks.h, History.[h/cpp], Reader.h, Editor.[h/cpp],Range.[h/cpp], SignalHandler.[h/cpp], TerminalDisplay.[h/cpp], TerminalDisplayUnix.[h/cpp], TerminalConfigUnix.[h/cpp]
2. Write tests for classes and functions defined in the files above.

Week 2:

Start porting Cling's auto-completion infrastructure to Clang-REPL.
Deliverables:

1. Possibly finish the remainder of the last week's week.
2. Port TextInput.[h/cpp], TextInputContext.[h/cpp]
3. Port the class UITabCompletion from UserInterface.[h/cpp] to Clang-REPL.
4. Port ClingCodeCompleteConsumer.[h/cpp] to Clang/lib/interpreter.cpp.

5. Write tests for classes and functions defined in the files above.

Week 3:

Buffer week for the previous work

Week 4:

Start designing and implementing type-directed auto-completion. Discuss the design plan with the mentor.

Deliverable:

1. Wrote a design document demonstrating how the subsystem works internally and with the auto-completion system.

Week 5

Implement type contexts for the type-directed auto-completion system.

Deliverables:

1. Implement the class TypeContext to keep track of bindings and their types. Type representations use classes from llvm:Type
2. Write a function, extractBindingAndType, to obtain the valid binding and its type from an AST. The result is put into the TypeContext.
3. Write tests from the class TypeContext, the function extractBindingAndType

Week 6

Buffer week for the previous work

Week 7

Integrate the TypeContext class and the function extractBindingAndType with the completion system.

Deliverables:

1. Put extractBindingAndType in a proper place in UserInterface to gather bindings and their type information from user input.
2. Add a function, `typeOfAt`, to get a type related to the cursor's current position.
3. Add a filter function to filter valid candidates based on the result type of `typeOfAt` and TypeContext.
4. A preliminary type-directed auto-completion in Clang-REPL.

Week 8

Buffer week for the previous work.

Week 9

Improve type-directed auto-completion with subtyping.

Deliverables:

1. Implement a function `subtype` that checks if the first argument is a subtype of the second argument.
2. Integrate the function into the completion system.

Week 10

Buffer week for the previous work.

Week 11

Polish the patch and submit it to the LLVM project for review.

Deliverables:

1. A ready-to-be-reviewed patch
2. A submission to the LLVM project

Week 12

Change the code per reviewers' suggestions. This process may go on for more than one week.

Week 13

Wrap up the project

Deliverables:

1. Improve the documentation
2. Write a blog post about the work
3. Prepare a presentation.

About Me

Yuquan (Fred) Fu is a Ph.D. student specializing in programming languages. He primarily works on type systems for Typed Racket and other gradually typed languages under Dr. Sam Tobin-Hochstadt. He is an open-source enthusiast who has been long wanting to learn and contribute to LLVM and Clang. Coming from Racket, where the REPL plays a central role, He believes Clang-REPL is an excellent starting point for his LLVM/Clang contribution journey.

Availability

I can start as soon as the project is announced.
Usually, I can work for 20 hours per week.
I am on eastern time (UTC -4), and I am responsive and reachable by email and other tools the team and mentor use.