WebAssembly Support for clang-repl

- Google Summer of Code 2022 with LLVM Compiler Infrastructure (Project)
- Student: Anubhab Ghosh (Github)
- Mentors: Vassil Vassilev, Alexander Penev

Introduction

The Clang project includes an interpreter ClangREPL that allows interactive execution of C++ code by just-in-time (JIT) compiling it to native code. Xeus framework can be used to access it through Jupyter in a web browser. However, it shifts the actual computational load to the server instead which is often undesirable.

WebAssembly is a solution to the problem. It allows sandboxed execution of native (e.g. C/C++/Rust) programs compiled to an intermediate bytecode at closer to native speeds. The idea is to run clang-repl within WebAssembly and generate JIT-compiled WebAssembly code and execute it on the client side.

However, generating JIT-compiled code in WebAssembly comes with many challenges. The root problem is code inside a WebAssembly module is immutable which is unacceptable for JIT. WebAssembly uses a Harvard-like architecture where code does not even reside in the same memory address space as data. Instead at runtime, all functions are identified by integer indices whose actual internal representation is opaque to the program. This gives the engine more flexibility over how they want to implement it but is problematic for JIT compilers.

Goals

1. Enable clang-repl to generate WebAssembly output.
2. Port clang-repl and all associated dependencies and other infrastructure to run in WebAssembly inside a browser.
3. Execute the generated WebAssembly code in the browser to truly become an interpreter.
4. Add associated libraries, headers, etc. to make it functional for a user.
5. Integrate it with JupyterLite, which is a version of Jupyter that runs entirely on the client side.

Design and Implementation

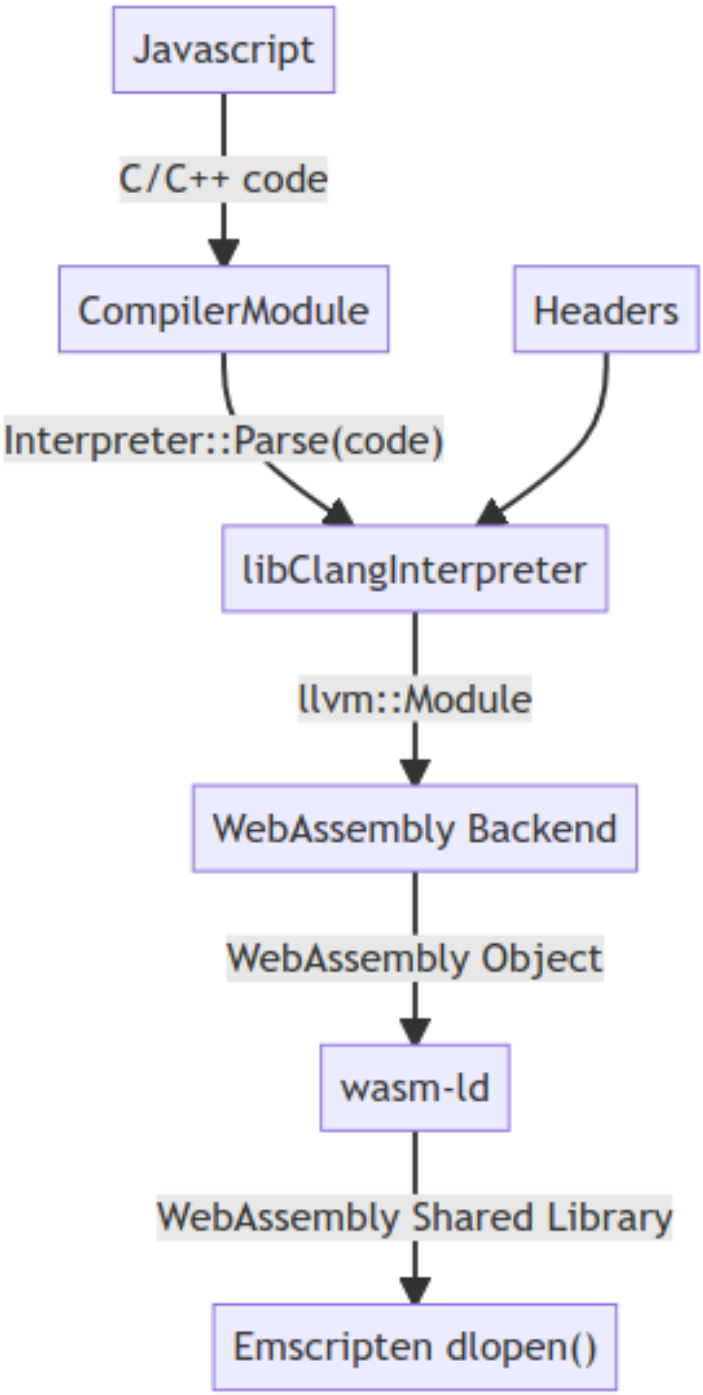
The solution to avoid the code immutability problem was to create a new WebAssembly module at each iteration of the REPL loop. Intially, a big precompiled WebAssembly module containing the Standard C/C++ libraries, LLVM, Clang, wasm-ld is sent to the browser. This module is responsible for running the interpreter and compiling the user code.

Each time the user provides the input code, it is run through the standard Clang interpreter pipeline beginning with `Interpreter::Parse()` where the incremental mode of Clang is used to parse the code and generate a `llvm::Module`.

But after this, we differ from the standard ClangREPL pipeline. We cannot call `Interpreter::Execute()` to execute the module because it relies on `JITLink` which directly places the compiled code into memory and marks the regions as executable. This is impossible in the immutable code world of WebAssembly.

Instead, we use the LLVM WebAssembly backend manually to produce an object file. This file is then passed to the WebAssembly version of LLD (`wasm-ld`) to turn it into a shared library which is written to the virtual file system of Emscripten.

Now the dynamic linking facilities of Emscripten can be used to load this library. Under the hood, this creates a new WebAssembly module from the virtual shared library file and instantiates it. It shares its data memory with the main module. WebAssembly exports and imports from this new module are linked with the original and earlier ones.


Javascript
C/C++ code
CompilerModule
Headers
Interpreter::Parse(code)
libClangInterpreter
llvm::Module
WebAssembly Backend
WebAssembly Object
wasm-ld
WebAssembly Shared Library
Emscripten dlopen()

Current State

A proof of concept demo is available that implements all of this out-of-tree. It includes standard libraries and SDL. But the goal is to integrate as much of it into upstream Clang as possible which is currently in progress (D158140) and expected to take some time. The integration with XeusLite/JupyterLite is currently not possible until the core functionality is accepted into Clang.

Future Work

XeusLite/JupyterLite integration is pending. That should allow a more convenient interface for using the interpreter in the browser.