LLVMN to Midden
Proof of Princple Project

John Skaller
mailto:skaller@internode.on.net

8 Dec 2022

1 Purpose

The purpose of this document is to outline a proposal to develop technology which demonstrates translation of LLVM assembler to Midden Assembler is possible, and discover restrictions on the inputs to allow the translation to proceed, and enhancements required in order to cover a significant part of the Midden VM capabilities.

2 Deliverables

2.1 The executable

Source code and build instructions to build an executable which can translate a single LLVM assembler source file into an Midden assembler source file.

2.2 Usage Description Document

Should specify:

1. How to use the executable to perform the translation.
2. How to assmeble the translation and execute the result on a Midden VM.
3. How to obtain the result and verify it is as expected.
4. How to obtain LLVM assembler from inputs in C and Rust.

3 Input Requirements

1. Input test cases written by hand in LLVM assembler.
2. Input test cases written in C.

3. Input test cases written in Rust.

The input tests need two forms. The translator developer will write some unit level test cases in LLVMN assembler and a few more complex functional cases; these will not be executable on Midden by demonstrate the translation has proceeded as expected. Verification will by examination.

Experts in Midden and the kinds of small operations it is expected to perform will need to write and supply additional test cases intended to actually execute on Midden, along with the expected behaviour. If these are written in C or Rust, the commands to invoke the front end translator of LLVM and produce assembler output must be provided.

4 Operating environment

For test purposes, both MacOS and Linux must be supported.

5 Translator Structure

The translator will need to operate in three phases.

1. The first phase, the *translator front end* will parse LLVM assembler into an internal encoding used as inputs by the second stage.
2. The second phase translates this internal input format into an internal output format by various processes including traditional term reductions (similar in concept to beta reductions in the lambda calculus).
3. The final phase, the *translator back end* will print out the Midden Assembler as a text file.

6 Translator programming language

The translator needs to be written in some programming language. Options are roughly:

1. C++
2. Felix
3. Ocaml
4. Rust
5. Haskell

Since LLVM is written in C++, it has the advantage the future integration with LLVM, that is, making MiddenVM a standard LLVM back end, may be easier. However C++ is a very poor choice for writing a compiler.

Rust is a better choice however this author is not an experience Rust programmer. Down the track this may be more viable since many developers involved in the Midden project use Rust. However Rust memory management is specifically designed to avoid the need for garbage collection which is very big advantage in developing translators.

Haskell would be an excellent choice especially for the middle phase of the translation, it has all the facilities required. However the author is not an experienced Haskell programmer and the purely functional model can sometimes be an impediment.

Ocaml is a good choice, since the author is familiar with it, and the Felix compiler is written in Ocaml, so the author is familiar with the techniques for writing compilers in Ocaml. However not many developers know Ocaml.

Felix is an interesting possibility. It is capable of both procedural and functional programming, has a strong static type system, although not as strong as Ocaml or Haskell, but much better than C++, and it is garbage collected. Unfortunately only the author, being the prime developer of Felix, is familiar with it.

Felix has another advantage: it generates C++ and is specifically designed to embed C++ too. So future integration with LLVM may be easier and large numbers of useful libraries are available. For example Google RE2 is actually already part of the run time.

It can also be run like Python, with no need for any build systems. It has much better string handling than Ocaml, and is not so heavily oriented to purely functional programming.

7 Intellectual Design Requirements

The principal intellectual problem to be solved and demonstrated by the initial work is to find an architectural mapping between the two machine models.

1. Memory access model
2. Instruction mapping
3. Function call mapping

Although both LLVM and Midden are stack machines, LLVM code assumes instructions can access arbitrary local variables in a function stack frame, and uses three address code to represent operations.

Midden on the other hand uses zero address code to perform operations and cannot access most of the stack using offsets (except the first few words).

The translator therefore needs a model to keep track of where local variables are on the stack and a way to retrieve them, a process which itself modifies the location of the variables with respect to the current stack pointer. So whilst the mapping of three address instructions such as integer addition from LLVM to pair of loads followed by a zero address add is actually fairly straight forward, modelling LLVM's random access linear memory model correctly is a significant problem and doing so efficiently is even harder.

In addtion, Midden has some specialised functions for hashing and other operations which has no direct equivalent in LLVM. So a way to invoke these instructions must be specified, probably by using an LLVM function call. The translator will then see these particular calls and invoke wrapper code, hand written in Midden assmbler, which actually performs the operation on the required arguments.

Finally, LLVM does not specify directly how a function is called. Instead it provides a simple universal call syntax with annotations suggesting which callling protocol should be used. A calling protocol in Midden is likely to be

1. Push each argument onto the stack
2. call the function
3. the return value is left on the stack.

where the significant challenge is getting the arguments specified by a LLVM stack offset onto the top of the Midden stack where instructions can process them.

Another point that needs investigation is whether use of operations on the Midden base prime field are simply using the field as a local approximation to the integers, or actually requires respecting the field operations (especially division).