Adding Pointer Support to Miri's FFI

Bachelor Thesis Project Description

Lucas Werner
Supervised by Prof. Dr. Ralf Jung
Programming Language Foundations Lab, ETH Zürich

April 3, 2024

1 Introduction

Rust[1] is an emerging programming language with strong emphasis on safety, speed and concurrency. To achieve speeds comparable to other common low-level systems languages, Rust code is compiled through several layers of intermediate representation where optimizations may extensively transform a user's code. This works well with the strong program invariants enforced through Rust's type system and ownership memory model, which are exploited on various levels of Rust's program analysis.

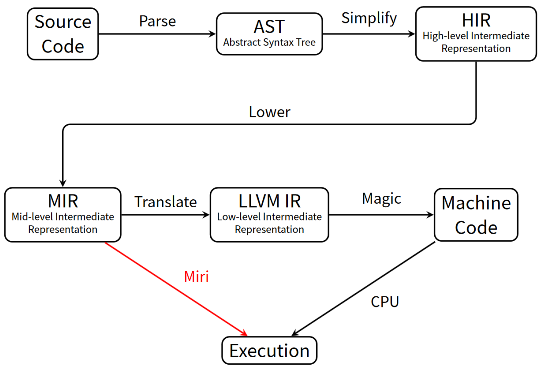
Source Code
Parse
AST
Abstract Syntax Tree
Simplify
HIR
High-level Intermediate Representation
Lower
MIR
Mid-level Intermediate Representation
Translate
LLVM IR
Low-level Intermediate Representation
Magic
Machine Code
Miri
CPU
Execution

Figure 1: The various compilation stages of the Rust compiler and where Miri fits into this.[2]

A special point of interest is Rust's mid-level intermediate representation (MIR), where a number of optimization passes are be performed before LLVM takes over at a lower level of compilation. MIR code at this stage can be run through an interpreter for various useful purposes: Miri[3] is able to make use of MIR interpretation to detect undefined behaviour (UB) in Rust programs, notably inside `unsafe` blocks. For this, Miri keeps track of an extended program state, including memory initialization and pointer provenance[4] to detect when uninitialized or out-of-bounds memory is accessed.

Special care needs to be taken when a given program tries to call an external function. Miri currently recognizes a basic set of common such functions whose behaviour it knows how to handle. Miri is also able to execute arbitrary native functions as long as they only take in and return integer values. In all other cases Miri currently does not know how to handle its bookkeeping of the program state when interacting with native code, and aborts.

The goal of this project is to extend Miri in such a way as to allow for calling external native code that makes use of pointer arguments. The benefit of this would be to enable Miri to run and test a much larger class of real-world Rust programs that rely on native code not explicitly reproduced inside Miri, such as functions accessible through the C FFI.

2 Approach

Extending how Miri is able to execute these foreign functions requires specific, well-adapted changes in parts of its existing codebase.

2.1 Memory Layout and Alignment

Miri is already capable of executing native code that handles integers. However, when we need to pass a pointer this is a problem for Miri since the data actually lives in Miri's "interpreter memory" via `Allocations`. To allow native code to access this memory we need to ensure that the data bytes inside the `Allocation` and the pointer we pass to the code actually match. For this the implementation needs to ensure proper byte alignment of allocated datastructures.

2.2 Initialization and Pointer Provenance

Miri `Allocations` not only store the allocated data they represent, but also which bytes are currently initialized (i.e. valid to access) and for pointers specifically something called pointer provenance. Pointer provenance is necessary in the context of pointer optimizations to ensure a pointer does not 'access memory that it is not supposed to.'[5] Pointer provenance in Rust exists but is still a matter of research, and therefore still a subtle source of UB, and which Miri can keep track of.
Miri needs to decide what state its interpreter memory is in after a call to native code returns. Since the focus is not on dealing with the correctness of external code itself, we make some assumptions:

- We assume the foreign function call is well-behaved (does not cause UB when called from correct code).
- We assume the foreign call might have change the initialization of any bytes it had access to, and therefore mark them as initialized.
- We assume the foreign call might have changed the provenance of pointer bytes it had access to. Since we cannot guess the precise provenance, we set their provenance to Miri's built-in `Wildcard`.

It is important to note that if the native code follows nested pointers then we have to deal with the changed states for those allocations too.

3 Core Goals

The central goals to be achieved in the project are as follows:

I. Implementing proper access to Miri's memory for a native code call that takes in a pointer, as described in 2.1.

II. Implementing proper handling of Miri's extended allocation state (initialization, provenance) for a native code call that takes in a pointer, as described in 2.2.

The core goals only cover native code handling existing allocations. A more challenging goal would be to handle new allocations originating from native code; see extension goals.

4 Extension Goals

Additional relevant issues are defined to be extension goals, consisting of the following:

I. Try loading an actual C library. Achieving this would suggest that the core goals were achieved well enough to already be amenable for practical usage of certain, actual C libraries when running Miri.

II. Implement a mechanism to let Miri handle memory that was newly allocated and returned from inside a C call.

5 Schedule

To accommodate for delays a schedule tighter than six months is laid out:

Core Goals. 10 weeks.

Extension Goals. 6 weeks.

Writing. 4 weeks.

References

[1] *Rust Programming Language*, `https://www.rust-lang.org/`.

[2] Scott Olson, *Miri, An interpreter for Rust's mid-level intermediate representation*, `https://solson.me/miri-slides.pdf`.

[3] *Miri*, `https://github.com/rust-lang/miri`.

[4] *Rust has Provenance (RFC 3559)*, `https://rust-lang.github.io/rfcs/3559-rust-has-provenance.html`.

[5] *Rust has Provenance (Pointers Are Complicated II, or: We need better language specs*, https://www.ralfj.de/blog/2020/12/14/provenance.html.