Introducing Concurrency to MiniRust

Bachelor Thesis Project Description

Yannik Wyss

Supervised by Prof. Ralf Jung

ETH Zürich

December 2022

1 Problem Description

MiniRust[1] is both a language and an operational semantics. The MiniRust project aims to define a precise specification of Rust's semantics. Since the goal is not just to define the semantics, but also to communicate it to the programmer, the specification is written as an interpreter. In the greater picture, MiniRust is a proposal for the soon-to-be-formed operational semantics team of Rust. The MiniRust language is condensed to a small core, far from the feature richness of a language like Rust. This is by design. It is much easier to translate Rust into MiniRust and define MiniRust precisely, compared to defining Rust directly.

This work adds concurrency to MiniRust. As a first goal, it adds some necessities like extending the machine to allow for multiple threads, adding intrinsics for creating and joining threads, and adding basic synchronization methods like locks or atomic memory accesses. As an extension, the detection of data races could be added.

2 Approach

In this section, the key contributions needed for concurrency in MiniRust are outlined.

2.1 Extension of the Machine

To allow for concurrency, a threading system must be added to the machine. In a PR[2] to miri, this is done by replacing the global stack and program counter with a ThreadSet. Each of these threads will then have its own private stack and program counter. This system also needs a scheduler to decide the order in which threads will make progress.

2.2 Intrinsics

Threads need a way to be created and joined. Therefore, this work must add intrinsics to MiniRust, which allow for these actions. In Rust, the intrinsic to create threads takes a function pointer as an argument. This is currently not possible in MiniRust, since it does not have function pointers. There are two possible approaches to this problem. The intrinsic could use the current system, taking a FnName as an argument. Another option is to change the current system to work with function pointers.

2.3 Synchronization

For concurrency to be useful, we need to coordinate between threads. An example would be locks. Locks can be used to create critical sections, in which we can be sure that the thread is the only one in that critical section. Locks can be provided by the system. It can then manage queues for blocked, waiting, and runnable threads. The scheduler can be extended to take this information into account when it chooses the next thread to make progress.

Other synchronization methods include atomic memory operations. When writing sequential code, the default memory accesses are thought to happen in a fixed order. In reality, this is not the case. The compiler is allowed to move those instructions around to increase efficiency, as long as the behavior doesn't change in a sequential execution. This becomes an issue once we want to use memory operations to communicate between threads. For this reason special atomic memory operations are added. They are understood as communication between threads and are not to be reordered by the

compiler. We additionally need a mechanism to keep the programmer from using default memory operations for communication. Therefore such interactions, called data races are undefined behavior.

2.4 Data Race

Intuitively, a data race is a situation in the execution of a program where multiple threads try to change the same location at the same time. As mentioned before, this is undefined behavior, which means that the abstract machine is allowed to interpret those instructions in any way possible. By definition, a program contains a data race if there is a reachable program state where there are two distinct threads for each of which the next operation is a memory access, both accesses are to the same location, at least one write, at least one non-atomic. [3]

When designing a system to detect those data races, we can exploit that MiniRust is primarily intended for verification. This means that every possible program trace will be checked. It is therefore sufficient for our system to detect the data race in one of those traces. The idea is to keep a set of all memory operations executed by a statement. If the next statement is then executed by another thread, we can check the set and find any conflicting operations.

3 Core Goals

The core goal of this thesis is to implement basic concurrency for MiniRust.

3.1 Machine and System extension

Extend the machine to allow for multiple threads to run concurrently. Add intrinsics like create and join for concurrency.

- Extend the existing interpreter without removing its core idea of being humanly understandable.
- Understand and decide what intrinsics are needed for concurrency.

3.2 Synchronization

Create basic synchronization methods like locks.

- Implement locks
- Adding intrinsics for atomic memory accesses.

4 Extension Goal

Some more extension goals might be added during the work.

4.1 Data Race Detection

Extend the memory or thread system to detect data races. The goal here is to do it in a Sequential Consistent setting, where changes to the memory are seen by all threads at some defined time.

- Analyze the solution space and decide on a well-suited solution considering the design goals of MiniRust.
- Implement the solution.

4.2 Synchronization Extended

There is a wide range of synchronization methods that could be useful when the machine implements them. An example would be semaphores.

- Implementing additional synchronization methods

References

[1] R. Jung, “minirust: A precise specification for ”rust lite / mir plus”.” [Online]. Available: https://github.com/RalfJung/minirust

[2] V. Astrauskas, “Implement basic support for concurrency (linux/macos only).” [Online]. Available: https://github.com/rust-lang/miri/pull/1284/

[3] “Rustbelt.” [Online]. Available: https://plv.mpi-sws.org/rustbelt/popl18/paper.pdf