ETH zürich

Bachelor's Thesis

Institute for Programming Languages and Systems, Department of Computer Science, ETH Zurich

Introducing Concurrency to MiniRust

by Yannik Wyss

Spring 2023

ETH student ID:	20-918-702
E-mail address:	yawyss@ethz.ch
Supervisor:	Prof. Dr. Ralf Jung
Date of submission:	August 20, 2023

Abstract

In recent years, the programming language Rust has emerged as a popular and loved alternative to existing system-level languages. Nevertheless, it is still missing a formal specification, which is where MiniRust comes in. MiniRust is an interpreter designed to formally specify the operational semantics of the programming language Rust. It has a focus on clearly specifying both defined and undefined behaviour. This work extends the existing framework by adding the specification for concurrent programs and the various mechanisms needed to create concurrent code. Due to the fundamental changes introduced by concurrency, the structure of the interpreter has been re-written to replace the single stack by a set of threads, where each thread has access to its individual stack. The logic to define the concurrency is similar to interleaving semantics, where the global state gets changed by a single thread making progress. Furthermore, locking mechanisms have been added to enable safe accesses of multiple threads to shared data. Another fundamental aspect of this work, is the implementation of atomic memory operations and a data race detection system. A data race occurs when multiple threads operate on the same memory location at the same time without specifying that this is intended. It is undefined behavior and must therefore get detected by the interpreter.

Acknowledgements

I want to express my gratitude to my supervisor, Prof. Dr. Ralf Jung for his very direct support during this thesis by always reviewing the code I wrote and discussing every design decision I had to make to the finest detail. These discussions are the reason I now understand many intricacies about operational semantics I never thought I would grasp.

Contents

1 Introduction

In the realm of modern programming languages, Rust has emerged as a prominent contender due to its emphasis on memory safety, concurrency, and performance. Its unique features, such as ownership, borrowing, and lifetimes, make it a powerful tool for developing systems-level software that is both safe and efficient. All of these features make Rust one of the most loved languages. In Stack Overflow's 2023 developer survey Rust was the most admired language for the eighth consecutive year. [8]

While Rust has been very popular in recent years there is a lack of formal specification for it. Such a specification is needed in many use cases. It allows the compiler to be verified and be trusted by the programmer and it allows derived tools like symbolic execution or abstract interpretation to be based on a ground truth. MiniRust is a project that defines the operational semantics of Rust. It has the main focus of defining the behavior of unsafe Rust and clearly specifying both defined and undefined behavior. Undefined behavior means that the program is technically allowed to do anything since the programmer broke the contract between the compiler and the programmer on what programs are legal.

MiniRust is currently lacking many features. One of which is concurrency. In today's world where sequential execution does not speed up exponentially and even smart phones have multiple cores, concurrency is very relevant and can't be missing in the specification of Rust.

We extend MiniRust to define the semantics for basic concurrency and data race detection. In a first step we have to define the systems for basic concurrency. The interpreter loses its stack and gets a set of threads instead. Each of those threads then has its own stack with information about its execution state. To define concurrency, we consider to use a system similar to interleaving semantics, where the change of the global state is defined by the step of a single thread.

In a next phase we will add atomic memory operations and a data race detection system. Data races are the unintended, or unspecified, access to a shared memory location of multiple threads at the same time. It is undefined behavior because it gets in the way of some optimizations the compiler should make. To allow the programmer to have accesses to a shared memory location atomic memory operations are needed, those directly specify to the compiler that these might interact with other threads.

1.1 Organization

The thesis is split into 7 chapters. After this introduction we will give you an overview of the background relevant to this thesis which includes an overview to Rust, a deep dive into MiniRust, and a discussion of Concurrency.

In chapter 3-5 we will present all important features added to MiniRust during this thesis. Chapter 3 discusses function pointer, chapter 4 features the basic concurrency and locks, and chapter 5 introduces data races and atomic memory operations.

The last two chapters will then describe how we evaluated the code and conclude the thesis.

2 Background

2.1 Rust

Rust is a systems programming language that is designed to be safe, concurrent, and performant. It brings many good design choices together learned from problems with other system programming languages like C or C++. It has a very strong type system which hides any null values behind options or results and it has a novel approach to memory management with ownership. Many expressiv features like pattern matching are also supported by Rust.

2.1.1 Memory management system

One of the biggest issues with a language like C++ is memory safety. Programmer themselves are often responsible to make sure that they don't have any bad memory operations. These include dangling pointers, memory leaks, and double frees among other things. All these bugs occur in real systems and according to the Google and Microsoft about 70% of security vulnerabilities stem from memory bugs. [5]

Dangling pointers

A dangling pointer is a pointer that is pointing to an object that no longer exists. This is a bug that occurs when a programmer forgets some reference somewhere and frees the memory the reference points to. For example:

```c
int *x = (int *) malloc(sizeof int);
int *y = x;
free(x);
// From now on y is a dangling pointer
// and very dangerous to dereference.
...
// For example in a double free:
free(y);
```

Listing 2.1: Dangling pointer

While the bug is quite obvious in this toy example, detecting it in real programs is extremely difficult to do since it might not behave deterministically.

Double frees are a bug that could happen with a dangling pointer. When we free a pointer that was already freed we might free an object that happened to be allocated at the same place or simply destroy the data structures the memory manager had in its background.

Memory leaks

A memory leak happens if we allocate some memory on the heap and then forget that it ever existed. Once we lose the last reference to the memory it becomes leaked. While this bug might not seem that problematic for some programs it often becomes a major issue. When programs are long living these continuous leaks trash the memory and might over time even crash the machine because it has no more memory to give to the programs. For example:

```c
int *x = (int *) malloc(sizeof int); //A
x = null;
// From now on allocation site A is not usable but still allocated.
// It has been leaked
...
```

Listing 2.2: Leaked memory

Garbage collection

One solution to these bugs is presented by languages like Java. They have a garbage collection system. This means that they make it easy for the programmer to allocate new memory and then the system detects dynamically when some piece of memory is not usable anymore and frees it then itself. Rust decides not to use such a system, because while it is useful to the programmer it creates a big overhead and unexpected long waits when the system is collecting garbage.

Ownership

The memory management system chosen by Rust is Ownership. This means that any allocated memory belongs to some variable or some other object. Once the variable goes out of scope or the owning object itself gets dropped, the owned object also gets dropped.

```rust
let mut x = 0;
{
    let y = vec![1];
    x = y[0]
    // Since y goes out of scope at this point
    // the vector will get dropped and its memory freed.
}
...
```

Listing 2.3: Ownership

This ensures that memory can not be leaked because every object gets dropped once either the variable that owns it goes out of scope or the object it is owned by gets dropped. While this sounds a lot like garbage collection there is a key difference between the two. Garbage collection happens dynamically, this means that there is additional cost associated with it while the program is running. Ownership can be analysed statically and finds the correct point to drop each object at compile time. An important detail that will become important later is that this does not strictly hold only if we use pure safe Rust.

Borrowing

Ownership solves the issues of memory leaks and double frees because the programmer never has to think about allocations. But how do we ensure that we do not have any references to freed memory? Rusts solution are lifetimes combined with a borrow checker. Each reference has in its type not only the type it references but also a lower bound to how long it can exists, the lifetime of the object. These lifetimes then allow the borrow checker to statically detect that we access an object at some point where we can not be sure that the object still is alive and was not dropped or moved.

```rust
let x = vec![0];
// Borrow x.
let r = &x;
// A function that takes ownership of whatever x owned.
take_ownership(x);

// The borrow checker complains about us still
// using r because we borrowed
// from x, which no longer owns the object we reference.
let y = r[0];
```

Listing 2.4: Lifetime

2.1.2 Unsafe Rust

A system like ownership is very useful because the compiler uses it to keep the programmer safe. But what happens if we want to do some safe but more complex operation? For example would we still like to have a way to have multiple owners. This problem is solved by the standard library's *Rc*. It is a smart pointer that owns an object and can be owned by multiple owners. The issue is that with such a design it is hard if not impossible to convince the compiler that no memory errors are made. We need to do some operations that can not be verified by the compiler, these operations are mostly raw pointer arithmetic's. This is why Rust also contains unsafe Rust, a language in which unsafe operations, operations the compiler can not verify, are allowed. In the case of *Rc* this looks something like this:

```rust
pub fn new(value: T) -> Rc<T> {
    // Here we change to unsafe Rust
    unsafe {
        Self::from_inner(
            Box::leak(Box::new(RcBox {
                strong: Cell::new(1),
                weak: Cell::new(1),
                value
            })).into(),
        )
    }
}
```

Listing 2.5: Rc::new implementation, example for unsafe.[1]

Interesting is that while some unsafe Rust happens inside the *new* function its interface is still in safe Rust. This means that any unsafe Rust code must still keep any invariant we assume for safe Rust. But a reference counting object creates a major issue for us, while pure safe Rust could even keep away memory leaks, *Rc* combined with *RefCell* can once again introduce memory leaks. This is a downside, but since these memory leaks remain restricted to a small number of places this is still better than what C or C++ provides.

[1]Rc in Rust – `https://doc.rust-lang.org/src/alloc/rc.rs.html#372`

2.2 MiniRust

2.2.1 Semantics

Motivation

```rust
fn sort(vec: &mut Vec<i32>) {
    for i in 0..vec.len() {
        for j in 1..vec.len()-i {
            if vec[j-1] > vec[j] {
                (vec[j-1],vec[j]) = (vec[j],vec[j-1]);
            }
        }
    }
}
```

Listing 2.6: Bubble sort

You have just written this algorithm to sort a vector of integers. Now you would like to convince your colleagues about the correctness of your program. How could you go about this? Well you could start by just assuming what the different lines in your code mean and then try to argue in English sentences that your assumptions are most likely correct and that given those the program will in fact sort the vector. This is very imprecise and we would like to have some framework to argue, reason, and discuss about our program in a more rigorous fashion. This framework is what the semantics of the programming language provides.

Definition

The semantics assigns computational meaning to a valid string of syntactically correct code. It defines what it means to declare a variable or to dereference a pointer among other things.

It also indirectly defines the behavior of the programmer. In our case the behavior is the traces of prints the program does and whether it terminates. A program might be allowed to have many different traces it produces. This can stem from non-determinism of the program or from different decisions the compiler can make for the program.

2.2.2 As the machine does

Even if such a rigorous semantics does not yet exist for Rust we can still compile our Rust code and use it, often the program even does what we expected. We might just try to prove the correctness over what the machine does.

We could take the assembly code produced by the compiler:

```
 1  ...
 2  .LBB0_5:
 3      cmpq        %rdi, %rsi
 4      je  .LBB0_11
 5      cmpq        %rdi, %rcx
 6      je  .LBB0_12
 7      movl    (%rax,%rdi,4), %r9d
 8      movl        4(%rax,%rdi,4), %r10d
 9      cmpl        %r10d, %r9d
10      jle .LBB0_8
11      movl        %r10d, (%rax,%rdi,4)
12      movl        %r9d, 4(%rax,%rdi,4)
13      jmp .LBB0_8
14  ...
```

Listing 2.7: The inner body, it compares the integers and swaps them if needed.

The code in combination with an instruction set architecture, the rules by which chips are designed, would give us a rigorous way to argue about our program. An argument could go something like this: As you can see on line 7 and 8 we load both `i32`. We then compare them and if `%r9d > %r10d` it will swaps them. Now with those swaps happening each time we go through the array, it pushes with n loops the biggest element to the right most place and every other also into its position.

But with this choice to define the semantics of our language come some major issues.

Issue optimization

The very attentative reader might have already realized that the assembly code example is optimized. The unoptimized version of the code is so bloated that it would not give us a good example of how we might be able to argue with it. What happens if the behavior of our program gets changed with some optimization? Is the optimization wrong or is this just another behavior that is allowed by the semantics? It is not good to have those questions unanswered because this makes it very hard to argue about the correctness of any program.

Issue instability

Another issue we have to consider is that the compiler changes. The Rust compiler goes through updates all the time and these updates can change how our piece of code is compiled. This would mean that the semantics of our language changes every time the compiler gets updated. What if a compiler change is buggy and does not do what the compiler designer intended to do? It would not be possible to distinguish between such a situation and just a redefinition of Rust's semantic.

2.2.3 C standard

Now that we have analyzed why we need some more high level semantics we should look at other languages that already have a formal semantics and try to understand the benefits and issues of those. One very interesting language for this is C since it is also a systems language and is widely used. In the beginning C did not have any formal semantics and had some of the same issues we just discussed. But at some point the people decided they wanted some specification and they chose to define the semantics of C with axioms. For example:

> The **sizeof** operator yields the size (in bytes) of its operand, which may be an expression or the parenthesized name of a type. The size is determined from the type of the operand. The result is an integer. If the type of the operand is a variable length array type, the operand is evaluated; otherwise, the operand is not evaluated and the result is an integer constant. [2]

Natural language

As you can see the standard is written in natural language. It was first written in an attempt to stop the different C dialects from diverging. Today this creates many conflicts because some axioms are ambiguous and there are many discussions about what they mean.

One work that has extensively analyzed these issues is "The C standard formalized in Coq" [4]. It has many examples of potential flaws and consequences if we only define a semantics through natural language.

In type-based alias analysis, type information is used to determine whether pointers can alias or not. Consider the following example:

```c
struct t1 { int m; };
struct t2 { int m; };
int g(struct t1 *p, struct t2 *q) {
    int z = q->m; p->m = 10; return z;
}
```

Listing 2.8: Non aliasing pointer in C

Since both pointers have different type, the compiler can assume that they do not alias and optimize the code to just do `p->m = 10; return q->m`. But what if the code is called through this code:

[2] 6.5.3.4.2 in the C standard [1]

```c
union t1_or_t2 { t1 x; t2 y; } u = { .y = { .m = 3 } };
// g is called with aliased pointers &u.x and &u.y
return g(&u.x, &u.y);
```

Listing 2.9: Possible UB in C

This code would have changed behavior because of the optimization. To decide whether this is an error by the programmer or the compiler we search in the C standard [1] and in paragraph 6.5.2.3.6 we find:

> One special guarantee is made in order to simplify the use of unions: if a union contains several structures that share a common initial sequence (see below), and if the union object currently contains one of these structures, it is permitted to inspect the common initial part of any of them anywhere that a declaration of the completed type of the union is **visible**. Two structures share a common initial sequence if corresponding members have compatible types (and, for bit-fields, the same widths) for a sequence of one or more initial members.

If we read this specification it becomes clear that the call to `g` was undefined behavior if the union was not visible to `g`. The problem now is, that the meaning of visible is not clearly defined anywhere in the standard and it is only shown with examples.

2.2.4 Undefined Behavior

We already mentioned the term behavior earlier. We declared the behavior of a program as the traces of all prints it can make combined with whether it terminates or not. Undefined behavior occurs when there is no specification for what should happen once the execution reaches it. If the execution can reach the undefined behavior anything is allowed to happen in the execution. It is allowed to have any behavior. Technically it is allowed to shut down the machine or delete system 32.

While this might seem like a very unnecessary burden on the programmer, to always think about what might be undefined behavior it is necessary. Undefined behavior is needed because without it almost no optimization good compilers make can be justified.

One key issue with the C standard is that it mostly defines undefined behavior through not specifying it through axioms. This definition through non definition is very hard to handle. We can not really know if there is anything that is undefined behavior even if we would like it to be defined.

In a modern specification we strive for a better solution. This is why we want MiniRust to explicitly call out undefined behavior. This would be very hard with an informal axiomatic semantics like the C standard.

2.2.5 The interpreter idea

Operational semantics

Operational semantics define the step-by-step rules for the language. They describe how some abstract machine state moves into another based on local rules. There are big step operational semantics and small step operational semantics.

In big step semantics we try to prove one big step from the initial state to the final state, with any sub proof proving some intermediate step.

In small-step semantics on the other hand the proofs only prove some small "atomic" step and we then get a trace of the program execution. This is a more powerful way to create semantics because it gives us a better way to understand parallelism, non-determinism, and non-terminating programs. With this knowledge we analyze these semantics in a deeper fashion by explaining it based on a toy example.

The toy language is as follows. Its building blocks are an assignment and a sequential concatenation of two blocks. A program might look something like this:

$$\begin{aligned} x &:= 3 + 4; \\ y &:= x + 1 \end{aligned}$$

The set of all possible programs is called *Progs* and can be seen as part of the syntax definition of the language.

The first ingredient our small-steps semantics needs is a machine state σ. It should capture any information we need to know about the machine, in our case those are all values assigned to some variable. A possible assignment would be $\sigma = \{x \mapsto 2, y \mapsto 1\}$. The set of all machine states is called *State*.

There also needs to be some form of transition rules that define the transition relation $\rightarrow_1$. It defines what states combined with some program can get to some other state-program pair or to some final state.

$$\rightarrow_1 \subseteq (\mathit{Progs} \times \mathit{State}) \times (\mathit{Progs} \times \mathit{State} \cup \mathit{State})$$

Let us first consider the rule for an assignment. We would like the rule to say, that the state must get updated such that the variable assigned to gets the new value.

$$\frac{}{\langle v := e_1 + e_2, \sigma\rangle \rightarrow_1 \sigma[v \mapsto A(e_1 + e_2)\sigma]}\ \text{Assign}$$

Where A is a function that given some machine state σ returns the arithmetic value of an expression.

The other rules we need are the rules for sequentiallity. Here we must differentiate between the case that the first part would terminate and the first part would not terminate. Giving us the following rules.

$$\frac{\langle p_1, \sigma \rangle \rightarrow_1 \sigma'}{\langle p_1; p_2, \sigma \rangle \rightarrow_1 \langle p_2, \sigma' \rangle}\ \text{SequentialTerm} \qquad \frac{\langle p_1, \sigma \rangle \rightarrow_1 \langle p_1', \sigma' \rangle}{\langle p_1; p_2, \sigma \rangle \rightarrow_1 \langle p_1'; p_2, \sigma' \rangle}\ \text{SequentialCont}$$

The only other rule we need is what to do when the program still has some statements left.

With these three rules and our states we have a very little language with its semantics defined. We can now use this semantics to prove things about programs in our language.

Let us prove that the following program calculates $Y = 4 \cdot X$.

$$x := x + x;$$
$$y := x + x$$

We start with some initial state $\sigma = \{x \mapsto X_0\}$. The lowest rule that can be applied is the sequential one. Which then gets justified by the assign rule.

$$\frac{\dfrac{}{\langle x := x + x, \sigma \rangle \rightarrow_1 \sigma[x \mapsto 2 \cdot X_0]}\ \text{Assign}}{\langle x := x + x; y := x + x, \sigma \rangle \rightarrow_1 \langle y := x + x, \sigma[x \mapsto 2 \cdot X_0] \rangle}\ \text{SequentialTerm}$$

The second time we only have an assignment. This keeps the proof tree small by just justifying the assignment.

$$\frac{}{\langle y := x + x, \sigma[x \mapsto 2 \cdot X_0] \rangle \rightarrow_1 \sigma[x \mapsto 2 \cdot X_0, y \mapsto 4 \cdot X_0]}\ \text{Assign}$$

Overall we then get the final state where $y = 4 \cdot X_0$, which is the thing we wanted to show about this program. One of the major benefits of a small-steps semantics is that it makes derived tools like static analysers or symbolic execution easier.

Interpreter over mathematical notation

Let us consider some of the implications of the mathematical notation. While for a small language these rules are still very manageable for the goals of MiniRust with operations working with bytes such a rule set would be very hard to get correct. With an interpreter written in code this becomes significantly less problematic because we can create tests for it and can actually run it.

Another benefit of the interpreter in code would be that it is more readable for programmers. They are very used to reading code and many are not that familiar with the mathematical notation typically used for small-step semantics.

2.2.6 MiniRust in detail

MiniRust is an idealized MIR-like language with the purpose of serving as a "core" for Rust. To define Rusts semantics through MiniRust we use some form of translation. The semantics of Rust are then defined by all behaviors that MiniRust can have for a given program. Since MiniRust does not really have features like traits, pattern matching, and generics. This means that this translation does a lot of heavy lifting for the Rust specification. This is not a great issue because MiniRust as a project is more concerned with what constitutes as Undefined Behavior and what not. [2]

specr lang

While we mostly care for the interpreter that defines the semantics of MiniRust we must first understand what language is used to define both the semantics and syntax of MiniRust. specr lang is this language and it has some very interesting features we would like to explore.

```
impl<M: Memory> Machine<M> {
    fn eval_statement(
        &mut self,
        Statement::Assign { destination, source }: Statement
    ) -> NdResult {
        let (place, ptype) = self.eval_place(destination)?;
        let (val, _) = self.eval_value(source)?;
        self.mem.typed_store(place, val, ptype)?;

        ret(())
    }
}
```

Listing 2.10: Example of some specr lang code [3]

The first thing to mention is the fact that the syntax looks a lot like Rust, it removes some of the pains with Rust like pointer indirections for recursive types. Its semantics should always be obvious even if its not well defined.

What you might have noticed, is that the function has a pattern matching. This is a special feature of specr lang. Another special feature of specr lang is non-determinism. This would look as follows in code:

```
let x = pick(distr, |x| {
    x >= 0 && x < 10
})?;

```

[3]Assign Statement evaluation – https://github.com/RalfJung/minirust/blob/e81f8100600a84afad7b501d299784bd483c5f41/spec/lang/step.md

```
let y = predict(distr, |y| {
    y >= 10
})?;

if y != 532 {
    do_something_bad();
}
```

Listing 2.11: The non-deterministic functions in specr lang

The limitations on the values returned by the real non-determinism only depend on the closure we hand it, it can only return values for which the closure evaluates to true. The non-determinism is also given a distribution, but this distribution is only needed if specr lang is run in a random setting, which in our case happens if we transpile it to Rust.

- The `pick` non-determinism is arbitrary choice. It returns any value that fulfills the limitations.
- The `predict` non-determinism on the other hand is a best choice non-determinism. It predicts the best value possible for the programmer. In our case this might be 532 since something bad would happen otherwise.

MiniRust language

We will now give you a quick overview over the MiniRust language before we try to understand how the semantic specification works.

A MiniRust program has as its core building blocks functions and globals. Each function defines its locals and its basic blocks. Basic blocks are a execution unit with a list of statements and a terminator. The statements might be things like the assignment and the terminators are things like a call or an if statement.

The assignment statement is described using a `PlaceExpr` and a `ValueExpr`:

```
Statement::Assign {
    destination: PlaceExpr,
    source: ValueExpr,
}
```

Listing 2.12: Syntax of the assignment [4]

The `PlaceExpr` describes how the interpreter can construct a `Place` and a `ValueExpr` can be evaluated to a `Value`. A value is an abstract object, for example a mathematical integer or a pointer. A place on the other hand can be understood as a pointer, that is dereferenced. It represents the bytes in memory

[4]Syntax–`https://github.com/RalfJung/minirust/blob/1585b95ba8c156f1b62eb2ee358c5791ba7429ec/spec/lang/syntax.md`

that are pointed to by the pointer. When the assign statement wants to store the value to the bytes referenced by the `Place` it needs a `Type` like a `u32`. A type implements an encode and a decode method, which allow us to turn a value into bytes, that then can be stored and vice-versa. [7]

Inner-workings of the interpreter

The interpreter is written as a machine. Let us analyze the structure of the machine before it was modified by this work. Here is a list of the most important fields of the machine:

prog is the program this machine is running. It contains all functions and tells us where the start function is.

mem is the memory. It is an interface with all functionality we need from memory, like allocating, storing to, and loading from memory locations.

intptrcast is an IntPtrCast. The integer pointer cast unit tracks which pointer have been cast to integers and are therefore leaked. It then `predicts` where a pointer comes from when it is cast from an integer.

stack is the Stack. It is a list of `StackFrames`. A stack frame contains all information about the function initiation it stems from. It contains the addresses of all locals, the current execution state of the function, and the caller information.

Every semantic definition of MiniRust is made through the machine. If we think back to the small-steps semantics, there we also needed σ as the state we have to consider about the machine. The rules from our toy language like assignment are defined through the `Machine::step(&mut self)` function.

Example with small program

Let us consider this small program and then see how the specification can be used to interpret it. Our example program calculates `x + 5` and prints it:

```rust
fn main() {
    let x = 10;
    print(x + 5);
    exit();
}
```

Listing 2.13: Small program

We must first translate this program to MiniRust syntax. Once that is done we can create a new machine for it by executing `machine = Machine::new(prog)`. This gives us a new machine that is running the program and has not done any progress on it so far. `machine` is comparable to σ_0, the initial state in the mathematical notation. We can then call `machine.step()?` to make the first step. In mathematical notation this would be $\rightarrow_1$. The variable `machine` will then store a next state of the machine σ_1 such that $\sigma_0 \rightarrow_1 \sigma_1$. Let us analyze this first step:

1. `step` gets executed finds the top of the stack and finds the main function. It then evaluates the first statement of the main function.

2. `eval_statement` is the function to evaluate the first statement. This would be the `StorageLive` statement. It is hidden away in the `let` keyword and is used to allocate memory for x.

3. `mem.allocate` chooses a location for this allocation non-deterministcally and returns it.

4. `eval_statement` then remembers the allocation in the stack frame.

We can see that every step is justified by different parts, building a tree, similar to how the mathematical notation works.

We will go over the next steps a bit less detailed, they are:

1. `x = 10;` Here the machine gets the location of the allocation from the stack and then stores the value 10 to it, by turning it into bytes. This means that values are only transitory. There can only ever be values during a step. After the step is executed the value has been written into bytes again.

2. `print(x + 5);` The machine loads the value of x by once again converting the bytes to a transitory value it then adds 5 to it and hands the value to the print statement. Since print is an intrinsic it is a terminator, this means that we change the basic block we are in.

3. `exit();` Is also a terminator and it will throw `MachineStop`. This intrinsic is used to terminate the process.

This leaves us with the final state of the machine, where it produced one output, 15 and terminated. The most important part of all of this is the realization that mathematical values are just transitory and only make sense during a step. In each state of the machine any value is imprinted as bytes into the machines memory.

2.2.7 The toolchain

Since we will mention many of the tools related to MiniRust throughout this thesis, we quickly give you a short overview of each of them.

specr lang is a programming language defined for this project. Its syntax is similiar to the syntax of Rust and its semantics is a mixture of those of Rust and OCaml. It is garbage collected and its library data types are final.

MiniRust is an operational semantics defined by an interpreter written in specr lang and is used to define Rust.

specr-transpile is the tool used to translate specr lang code into Rust code.

minirust-rs is the into Rust transpiled version of MiniRust and can be used to test the MiniRust code. While it is special that the definition can again be written out in Rust there is nothing wrong with this step. It does not imply that we define Rust using Rust, it is just useful to find bugs in the definition that would be very hard to track down by hand.

libspecr is a library written in Rust that supports the execution of specr lang code as Rust code. It implements all the data type used like Maps, Sets, and Lists.

minimize is a tool to convert Rust code to MiniRust code. It is still in a early version since MiniRust does not support all needed language features, like enums, and development focused on other parts of the project.

2.3 Concurrency

In recent years the speed up of CPU through higher clock frequencies and smaller transistors has started to flatten out. Modern computer chips still become faster, but this speed increase does not necessarily make the same program faster. If a program is written as sequential code, the speed up it gets from newer chips is marginal, because today's speed up is achieved through computer chips getting more and more cores. Programmers can write programs that have multiple threads to get more speedups. A thread of execution is just like a little sequential process with its own stack from its function calls. The threads are all part of the same process and can therefore share memory, but they can get executed by different cores at the same time. This creates some issues that have to be addressed. For example is it not quite clear what happens if multiple memory accesses happen at the same time on different threads.

2.3.1 Sequential Consistency

In this thesis we will assume a limited scope of concurrency: sequentially consistent memory ordering. This means that for all threads of a single process any memory operation is visible to all of them at the same time. It is called sequentially consistent because the memory operations in a sequentially consistent memory ordering are ordered in such a way that they could have been produced by a single sequential process executing them.

This is not the case in modern computers, they have caches, store buffers, and out-of-order execution which means that we can not assume any operation to be visible to all threads at the same time. There are of course instructions that force modern computers to run in a sequentially consistent manner, but they slow them down. We will discuss memory models that would allow for these more involved behaviors in the conclusion 7.1.

3 Function Pointer

In the beginning of this work we first wanted to change the call system. So far the `Terminator::Call` had a function name it called, this worked fine so far but in the future we want to add the spawn intrinsic and to fit in with all other intrinsics it should take a `Value` as an argument. This is why we added a new constant, a function pointer and a new type, the function pointer type. With this system we could now give a function pointer to the spawn intrinsic and it would fit in well with the other intrinsics.

3.1 Implementation

We already mentioned some changes that had to be done but here is a full list of the changes done for function pointer.

Terminator::Call takes a `Value::Ptr` as the function instead of a function name.

Machine::new() allocates memory for each function and stores the function-address map in the new machine.

Constant::FnPointer is a new constant that, when evaluated, looks up its function name in the map and returns a `Value::Ptr` to the address.

PtrType::FnPtr is added.

check_wf passes the whole program through the tree because the `Constant::FnPointer` needs to check that it points to a valid function.

The changes are small but still interesting. The first change is the one in the evaluation of `Terminator::Call`.

```rust
// The callee is now a ‘Value‘.
let Value::Ptr(ptr) = self.eval_value(callee)? else {
    panic!("call on a non-pointer")
};

let mut funcs = self.fn_addrs.iter()
    .filter(|(_, fn_addr)| *fn_addr == ptr.addr);
let Some((func_name, _)) = funcs.next() else {
    throw_ub!("calling an address where there is no function");
};
let func = self.prog.functions[func_name];
```

Listing 3.1: Call, logic to find the function. [1]

Interesting is the way we find the function name. In the first iteration of this code, the machine stored the `prog`, the `fn_addrs`, and the `fn_map`. While `fn_addrs` mapped from function names to addresses and was only used for the constants. `fn_map` mapped addresses to functions directly. This machine design had some redundancy between these 3 fields. Initially, we contemplated removing `prog` since it was no longer utilized by the call, which was the only place it was used, but since the globals, which got added in parallel also needed some parts from the program we decided to keep both the program and the `fn_addrs` but remove `fn_map` since it was redundant. Now that the easy way of finding the function via a map is gone, we have to reverse the map which we did with this filter on the iterator.

Zero sized allocations

Another interesting discussion point would be the allocation of the functions memory. We wanted to make the allocations zero sized because we do not care about the size and it would be a nice constant. The only issue with this decision is the unwanted consequence that now multiple functions could be allocated to the same address. Making it impossible to revert the different functions. We solved the issue by not allowing any allocations to start at the same position.

[1] Call Evaluation – `https://github.com/RalfJung/minirust/blob/e81f8100600a84afad7b501d299784bd483c5f41/spec/lang/step.md`

Function pointers everywhere

Let's consider the following program:

```
fn f() {
    print(1);
}

fn main() {
    f();
}
```

Listing 3.2: Function pointer free program.

It can be observed that the program does not have any function pointers. Why then would we even consider to call `f` as a function pointer? The main idea is to keep the language minimal. While it is possible to emulate a function being called directly through function pointers, the other way is not possible. We went with such a design because it is similar to what happens in the machine. We can see the minimal change that happened for functions that get called by name:

```
callee_before: FnName,
callee_after: ValueExpr::Constant(
    Constant::FnPointer(FnName), Type::Ptr(PtrTy::FnPtr)
)
```

Listing 3.3: Called by name, changed syntax.

3.2 Well-formed

In `spec/lang/well-formed.md` the logic for checking well-formedness is found. Well-formedness is a requirement we pose to any program that can be executed by the machine. The goal is that the machine never panics when it executes `step`. These requirements differ wildly in what they do. For example is it required that a binary integer operation happens on two values with integer type, that every local is live when it is first used, or that the condition in an if statement has type `bool`.

3.2.1 Interaction with function pointer

The function pointers also introduce a new well-formedness requirement. They require that every `Constant::FnPointer` refers to an existing function.

```
(Constant::FnPointer(fn_name), Type::Ptr(_)) => {
    ensure(prog.functions.contains_key(fn_name))?;
}
```

Listing 3.4: well-formedness check for function pointer constant [2]

The `Constant::FnPointer` gives us a pointer when it is evaluated. We need to ensure that this does never lead to any panic of the machine, which is why we must check that the expected type to our constant is a pointer type and that the function referenced by the constant exists.

[2]Well-formedness for function pointer – `https://github.com/RalfJung/minirust/blob/e81f8100600a84afad7b501d299784bd483c5f41/spec/lang/well-formed.md`

4 Basic Concurrency

4.1 Interleaving Semantics

Interleaving semantics is a common choice to define the operational semantics for non-sequential programs. A step of a single thread is described as a step of the process, changing the global state of the process. [6] Any trace produced by the process is an interleaving of the threads taking atomic steps in an arbitrary order.

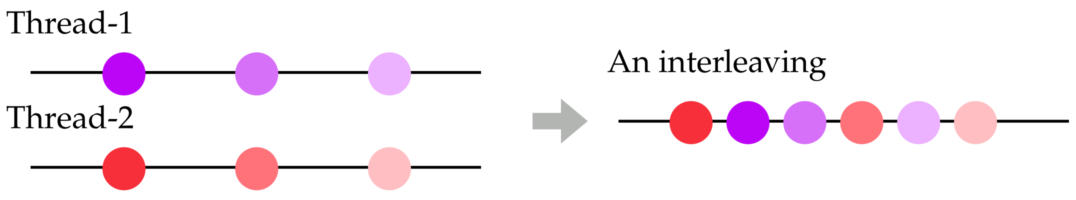

Figure 4.1: A possible interleaving of a system with two threads.

As you can see in Figure 4.1 an interleaving executes instructions from both threads in the desired order but switches between them arbitrarily.

It is important to realize that this semantics is limited by the meaning of the atomic step. Its granularity decides how detailed our interleavings are and therefore limit our ability to track interesting features. To get some intuition we will now consider what interleaving semantics could look like by defining them for our toy language.

4.1.1 Example

As we already have discussed in 2.2.5 small-step semantics are good at capturing concurrency. To quickly give you some insight into how such a system would work we extend our language. We would like to have parallel keyword like `par`. It is a statement that in a step either makes progress with the first or the second of its programs. It needs the following 4 rules.

$$\frac{\langle p_1, \sigma\rangle \rightarrow_1 \sigma'}{\langle par(p_1, p_2), \sigma\rangle \rightarrow_1 \langle p_2, \sigma'\rangle}\ \text{ParLTerm} \qquad \frac{\langle p_1, \sigma\rangle \rightarrow_1 \langle p_1', \sigma'\rangle}{\langle par(p_1, p_2), \sigma\rangle \rightarrow_1 \langle par(p_1', p_2), \sigma'\rangle}\ \text{ParLCont}$$

And of course parallelism can also make progress on the right part:

$$\frac{\langle p_2, \sigma \rangle \rightarrow_1 \sigma'}{\langle par(p_1, p_2), \sigma \rangle \rightarrow_1 \langle p_1, \sigma' \rangle}\ \text{ParRTerm} \qquad \frac{\langle p_2, \sigma \rangle \rightarrow_1 \langle p_2', \sigma' \rangle}{\langle par(p_1, p_2), \sigma \rangle \rightarrow_1 \langle par(p_1, p_2'), \sigma' \rangle}\ \text{ParRCont}$$

The rules are very similar to the sequential rules. But if we consider the implications of those rules more precisely we realize that while so far there always was just a single rule that was applicable at each point in time, now there can be multiple rules that can be applied at the same point.

If we consider the program:

$$par(x := 1 + 1, x := 2 + 2)$$

There are multiple traces possible. If we only consider the root of each step then "ParLTerm" followed by "Assign" would leave us with a state where $x = 4$. Where as "ParRTerm" followed by "Assign" gives us a state where $x = 2$. In our semantics those are both legal behaviors of the program.

An interesting site note to this change in the toy language is that it once again is to simple for MiniRust. An issue with this method of parallelization is that the *par* section ends at some point, implicitly joining the two threads. This should not be the case for MiniRust where we want to explicitly join threads again.

4.2 Threads

To make threads possible we have to change some things about MiniRust. Here are the main changes we made.

Thread is a new struct that captures all information we need about a single thread of execution. This means that it keeps its state and its stack. The `thread_id` is not stored in the thread, it is the index into the list.

ThreadState is an enum that represents the state the thread is currently in. Some of the states are `Enabled`, this means that the thread can make progress and `BlockedOnJoin(thread_id)`, in this state the thread can not make progress until the thread with id `thread_id` has terminated.

thread_list is a field of the machine and stores all threads of the machine.

active_thread is the thread id of the current thread. It is only of meaning during a step.

step function is extended to first select a thread to make progress on. Just like with interleaving semantics.

Spawn intrinsic is an intrinsic to create a new thread based on a function pointer and a data pointer.

Join intrinisic is an intrinsic to join an existing thread, it is, with the spawn intrinsic, our first method to synchronize threads.

Deadlock detection for the case that no thread can make progress. Since deadlocks are defined behavior in Rust, we need to throw a deadlock detection in the scheduler. If we would not do that the non-deterministic part of the scheduler would have no behavior which is not intended.

Blocking statements or intrinisics are possible. The thing that kept the thread blocked will remove the blocked state and clean everything up such that the thread can keep going as if it was never blocked.

4.2.1 Scheduler

```rust
// If no thread can make progress the non-deterministic
// part would have no behavior. To prevent that we throw
// on a deadlock.
if !self.threads
    .any( |thread| thread.state == ThreadState::Enabled ) {
    throw_deadlock!();
}

// Helper to guide the random sampling.
let distr = libspecr::IntDistribution {
    start: Int::ZERO,
    end: Int::from(self.threads.len()),
    divisor: Int::ONE,
};

// Non-deterministic picking of next thread.
let thread_id: ThreadId = pick(distr, |id: ThreadId| {
    let Some(thread) = self.threads.get(id) else {
        return false;
    };

    thread.state == ThreadState::Enabled
})?;
```

Listing 4.1: Scheduler with safety and helper. [1]

[1]Scheduler – `https://github.com/RalfJung/minirust/blob/e81f8100600a84afad7b501d299784bd483c5f41/spec/lang/step.md`

On line 4-7 the deadlock detection is found. Deadlock detection might be a strong word for this case. It is more of a lockdown detection. It gets triggered only if there is no more thread to make progress and will not be triggered by some subgroup of threads being in deadlock.

On line 17-23 we have the logic to non-deterministically choose a `thread_id` and thread to execute a step with. We must choose a thread that is enabled and is therefore able to make progress.

This small change of the `step` function is very similar to the introduction of the `par` statement to our toy language. With this non-determinism we define all possible "Par**" rules at the same time. Just like with interleaving semantics any trace produced by these different choices can be a legal behavior of the machine.

4.2.2 Thread list

In our first iteration of the thread list it was in fact a thread map. The choice was made that to capture any possible thread system it should be allowed for threads to have any integer id:

```
let distr = libspecr::IntDistribution {
    start: Int::ZERO,
    end: Int::from(2).pow(Int::from(32)),
    divisor: Int::ONE,
};

let thread_id: ThreadId = pick(distr, |id: ThreadId| {
    // Thread id positive
    if id < 0 { return false; }

    // No other thread has the same id.
    if self.thread_map.contains_key(id) { return false; }

    true
})?;

self.thread_map.insert(thread_id, Thread::new(func));
```

Listing 4.2: Pick a thread id with the thread map.

There are multiple issues with this design, some easily fixable, others not.

The first issue would be our decision to only limit the dimension of the integer by the distribution and not the non-determinism. This is an issue when you consider any program that would like to spawn a new thread. It gives us some integer type to encode the value with. The non-deterministic operation could now choose a value that does not fit and could not be encoded for that given type. This would render our specification unusable.

The next issue comes in combination with the scheduler and any random execution of the interpreter, there the distribution also was on all integers from 0 to $2^{32} - 1$. With just 50 guesses the `minirust-rs` would have to hit one of the few thread ids that actually exist. Fixing this would work by creating a list of thread ids and then finding a good one over those. That this was less readable and the fact that our design goals with MiniRust do not require any possible thread system to be captured led to the choice of a thread list, were ids are distributed increasingly.

4.2.3 Blocking

In the first iteration of the blocking system we had some interesting ideas in mind. But the overall design had major flaws. The idea was that a blocking statement would set the state of a thread to blocked and then once the thing that blocked the thread, the thread would try to evaluate the statement again. One major issue is, that this would disallow any side effects the statement might have. Currently this is not an issue because every statement that could block, does not have side effects.

But since the system should be future prove, this is not an option. This is why we went with an approach that only executes the evaluation once. After the evaluation, if the thread is blocked, it will no more be chosen for any further steps. Once the issue that blocked the thread is resolved, it gets enabled again by the resolver. In case of a join, the thread, that was still running when the other thread tried to join, will enable the other thread again once it terminates.

4.3 Locks

Threads as they were could only complete a limited number of tasks. We really only could know where the other thread is before we created it (It did not make any progress.) and after we joined it. But many concurrent tasks need some form of coordination between threads while both are running. This is where locks come in. A lock can only be held by one thread at a time, this allows us to have some coordination between the threads.

4.3.1 Reentrant lock

There are multiple lock designs that we could choose. The first we thought of is the reentrant lock. It allows a thread to acquire a lock it already owns.

A prominent place where we can find reentrant locks are in Java:

```java
synchronized void withdraw(int amount) {
    balance -= amount;
}
```

Listing 4.3: synchronized in Java

The **synchronized** keyword is an implicit lock on the object on which withdraw is called. It is very useful to the programmer because it makes synchronization easy. The only issue would be if a synchronized function calls another synchronized function on the same object. This is why Java's locks are reentrant.

4.3.2 Non-reentrant lock

The other option are non-reentrant locks. This means that a thread can not acquire a lock it already owns. Such behavior can be seen with Rust:

```rust
let handle = mutex.lock();

// Will never return. Deadlock!
let handle2 = mutex.lock();

//use handle
```

Listing 4.4: Deadlock in Rust

Because the handle can be used to mutate the data protected by the mutex dynamic ownership checks prevent the same thread to get another mutable reference in some other variable. This is why this code would deadlock.

The fact that the standard library locks are non-reentrant and that their design is simpler let to the decission that the locks in MiniRust should also be non-reentrant.

4.3.3 Implementation

LockState is an enum that keeps track of the state of a lock.

lock_list was added as a field of the machine. It keeps a list of lock states.

Intrinsics to create, acquire, and release locks.

Machine gets methods to manage the locks.

All changes are made in the `spec/lang/lock.md` file because it is an option that locks are only a temporary feature for testing right now.

We will now analyse some of the interesting things this change added. One would be the lock handover. As we already mentioned in the section about blocking 4.2.3 must a thread that enables another thread check that the issue that blocked that thread is resolved. In case of locking a lock this means that the other thread now owns the lock. We call it lock handover.

This lock handover had a serious bug in it, which caused perfectly correct programs to proclaim undefined behavior. Let us look at the handover with the bug in it, can you spot it?

```
 1 let acquirer_id: ThreadId = pick(distr, |id: ThreadId| {
 2     let Some(thread) = self.threads.get(id) else {
 3         return false;
 4     };
 5
 6     thread.state == ThreadState::BlockedOnLock(lock_id)
 7 })?;
 8
 9 // We unblock the selected thread.
10 self.threads.mutate_at(acquirer_id, |thread| {
11     thread.state = ThreadState::Enabled;
12 });
13
14 // Rather than unlock and lock again we just change the lock owner.
15 self.locks.mutate_at(lock_id, |lock| {
16     *lock = LockState::LockedBy(thread_id);
17 });
```

Listing 4.5: The lock bug [2]

The bug occurs on line 16. Instead of setting the state of the lock to be locked by the acquirer we set it back to the thread releasing it. When looking at this it is unbelievable that this bug even occurred, but when this code was first written the `acquirer_id` was `thread_id` as well. Since this meant that we shadowed a variable and this made the code less readable, it was later changed. Just not everywhere. This is how we got to the point of this bug occurring.

The problem was that we did not test the specification well enough. After implementing the locks we only had a test for each undefined behavior, but since the lock handover itself did not contain any undefined behavior it was not tested. Only once we did some integration testing the bug was discovered. Now there is a test that forces a handover to occur with high probability, helping us detect if such an issue ever occurs again. It is explained in more detail in the chapter Evaluation 6.

[2]Locking system with the bug – `https://github.com/RalfJung/minirust/commit/47c853a5903277201bac2d42312bff0321b083c8`

5 Data races

5.1 Data race

Let us consider the following Rust program:

```rust
static mut y: u32 = 0;
static mut x: u32 = 0;

fn data_race(a: u32) {
    thread::spawn(|| {
        x = 2*a+1;
        y = 2*a;
    });

    print(y);
    print(x);
}
```

Listing 5.1: Data race in Rust

To analyse what exactly the program does let us consider a run of this function with `a = 1`. The main thread spawns the second thread. It will first set x to be $2 \cdot 1 + 1 = 3$ and then set y to $2 \cdot 1 = 2$. Meanwhile the main thread prints out first y and then x. If we now consider the interleaving semantics of this program possible behaviors are:

- `2 3`, when the scheduler first executes the second thread and then the main thread.
- `0 3`, when the scheduler changes in between two statements of one of the threads.
- `0 0`, when the scheduler first executes the main thread.

One behavior that is not observable with this code would be `2 0` because x gets changed before y and y gets read before x.

But if we look at the code in the spawned thread alone the compiler might find some optimizations.

```
x = 2*a+1;
y = 2*a;
```

Listing 5.2: Before optimization

There is an optimization called common subexpression elimination. The compiler realises that we calculate $2 \cdot a$ twice and would like to spare this, which is why it changes the order of the assignments and makes them dependent.

```
y = 2*a;
x = y+1;
```

Listing 5.3: After optimization

While on this local scope this optimization seems unproblematic on the global scope it has unforeseen consequences. Now the behavior `2 0` is very much possible. This is wrong in our current understanding of the semantics.

But compiler designer would still like to make those optimizations. To stop the programmer from using such patterns we call them data races.

5.1.1 Definition

Two operations are in a data race if both happen at the same memory location unless

- Both are read operations.
- They happen on the same thread or signal handler.
- Both are atomic.
- One happened-before the other.

There is a lot to be unpacked. The first two bullet points seem quite obvious but what about the others?

Atomic memory operations

While compiler designer want to keep programmers from doing any pattern like in Listing 5.1 there are some reasons programmer would still want direct memory accesses even if they create non-determinism. The main reason being that locks are really slow and many programs would greatly suffer from the overhead induced by the locking mechanisms.

This is why language designer added so called atomic memory operations. Those are memory operations that are specifically made to be communication between threads. In Rust they are made possible by `Atomic*` where `*` could be `U32` for example. The compiler then knows that these operations are meant as communication between threads and therefore has to stop doing some optimizations.

Happened-before

The other thing to be unpacked is the happened-before relation. It can be designed over 6 basic rules if we only consider a sequentially consistent ordering.

Sequential behavior If A is before B in program order, A happened-before B.

Thread creation If A happened-before the thread creation of the thread that executes B, A happened-before B.

Thread join If B happened-after the thread was joined that executed A, A happened-before B.

Synchronous message If A happened-before a synchronous message is communicated and B happened-after the synchronous message is communicated, A happened-before B.

Asynchronous message If A happened-before an asynchronous message is sent and B happened-after the synchronous message is received, A happened-before B.

Transitivity If A happened-before B and B happened-before C, A happened-before C.

Intuitively it just means that if we can find a way in the timeline that forces A to be executed before B then A happened-before B. This might happen because both operations happen in a critical section protected by the same lock.

5.1.2 Example

Now that we have defined a data race we can go back to our example Listing 5.1 and show how it is a data race.

If we look at the execution graph:

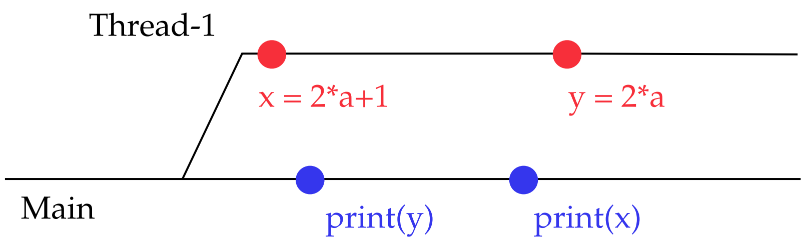

Figure 5.1: Data race example as a graph

We have the center line representing the execution of the main thread. It first spawns thread-1, then it prints out the value of y and then the value of x. The second thread writes to both x and y. If we now consider the operation of writing $2 \cdot a + 1$ to x and reading x to print it we can see that:

- One is a write operation, so both being read operations does not hold.
- One operation happens on thread-1 while the other happens on the main thread. So they do not happen on the same thread.
- Neither is an atomic operation.
- As we can see in Figure 5.1 there is nothing that forces either operation to happen before the other. Which is why we informally conclude that they are not in a happened-before relation.

With all exceptions not occurring we must conclude that these operations are in a data race and therefore the program has undefined behavior. If the programmer would still want to use this non-deterministic way to assign a value they would need to use `AtomicU32` or something similar.

5.2 Atomic Memory

5.2.1 Atomic memory in the machine

While in systems atomic loads and stores are often executed just like non-atomic operations, there are some special operations the machine provides. One of those is compare and exchange, it takes an atomic memory location, a value we expect to be there, and a value we would like to put there. After the compare and exchange is executed, we get the value that was written there before and based on that we will know if our operation was successful. This means that the value we expected was truly written there and we therefore wrote our value we wished to write.

```rust
fn compare_exchange(ptr: *mut u32, current: u32, next: u32) -> u32 {
    atomic {
        let before = *ptr;
        if before == current { *ptr = next; }
        return before;
    }
}
```

Listing 5.4: Compare and exchange if it were Rust code.

The compare and exchange operation is one of the most useful operations there are. It is used in many programs that strive to be lock free, but still want to allow for many threads to communcate effectively. It can also be used to build in process locks and other primitives. This is why it is the first additional operation we will also add to the MiniRust machine.

5.2.2 Implementation

For the implementation of our atomic memory we still cared for the atomicity of store and load operations since we plan to detect data races and should therefore know if the programmer intended accesses to happen at the same time.

AtomicMemory is a wrapper around `Memory` that exposes the interface as is, except that it demands an `Atomicity` for each store and load operation.

Atomicity is an enum that represents the different atomicities. It is either `Atomic` or `None`.

Store and Load intrinsic to have basic atomic memory operations.

Compare Exchange intrinsic as a more complex atomic memory operation.

This change is rather light weight but there are still some interesting things to look at. The first I would like to consider is the code for the compare exchange intrinsic.

```
// The value at the location right now.
let before = self.mem.typed_load(ptr, pty, Atomicity::Atomic)?;

// This is the central part of the operation.
// If the expected before value at ptr is the current value,
// then we exchange it for the next value.
if current == before {
    self.mem.typed_store(ptr, next, pty, Atomicity::Atomic)?;
}

ret(before)
```

Listing 5.5: Compare Exchange in MiniRust [1]

As you can see it is very similar to the pseudo code we wrote earlier 5.4. This is one of the strengths of MiniRust and writing the specification as code. It is very readable for programmers and gives them great inside into the semantics of the language.

The compare exchange intrinsic is only defined for integer types. For them the value comparison with == makes sense. This would not be the case for all types, for example for pointers the equals operator also compares their provenance, which is not what happens in a real system.

[1]Compare Exchange intrinsic – https://github.com/RalfJung/minirust/blob/e81f8100600a84afad7b501d299784bd483c5f41/spec/lang/intrinsic.md

Typed memory

One issue was the `TypedMemory` trait. It was a trait that was implemented for every memory and allowed it to take typed accesses. At first the plan was to implement the typed memory trait also for atomic memory, but this is not an option because their interfaces are different. One has to take an atomicity as an argument while the other does not. This left us with the only logical choice of removing the typed memory and instead implement the typed accesses for atomic memory. If we kept the typed memory and added typed accesses to the atomic memory we would have a redundancy because both the typed memory and the atomic memory would need to figure out the size of a type just to keep track of the access.

5.3 Data race detection

We already motivated the main reason we care about data races in Section 5.1. Since we consider them undefined behavior and MiniRust explicitly defines all undefined behavior, we need some way to detect that an execution has undefined behavior.

5.3.1 Happened-before

The main issue is to track if two operations can happen at the same time. One option would be to track the complete happened-before relation with total clocks. One such clock would be vector clocks, they allow us, with their clever design, to track the happened-before relation of an execution.

5.3.2 Trace based detection

Another approach we could take is to use our current design of interleaving semantics to track what can happen at the same time. The claim is, that two operations on different threads can only happen directly after each other in the trace if they are not in a happened-before relation. This is trivially true for all operations that are not used to communicate between threads. For atomic memory operations we need to argue for this and for the join, spawn, and lock intrinsics we add an additional detection system. Because they can be right after each other in the trace but still be in a happened-before relation.

Atomic Operations

We will first show why there can not be any communication of information about any non-atomic memory accesses the intrinsic do through the atomic operation, which would put them into a happened-before relation. If we can show this it is easy to follow that any data race detected is correct since they are not in a happened-before relation.

Let us look at the different abstracted accesses the different intrinsics take.

Atomic Write

1. `load(ptr)`
2. `load(val)`
3. `atomic_store(*ptr)`

Atomic Read

1. `load(ret_place)` We already load the place here
2. `load(ptr)`
3. `atomic_load(*ptr)`
4. `store(*ret_place)`

Compare Exchange

1. `load(ret_place)` We already load the place here
2. `load(ptr)`
3. `load(current)`
4. `load(next)`
5. `atomic_load(*ptr)`
6. (sometimes) `atomic_store(*ptr)`
7. `store(*ret_place)`

We can see that all executions of those intrinsics follow a similar pattern. They do all the loads to get the values and places they need, they then do the atomic loads then the atomic stores and then do any store operation. Because any pointer we dereference is first loaded non atomically we can conclude that it is not possible that the non-atomic operations are ordered through the atomic communication.

Synchronizing Operations

There are three operations that need special treatment because they allow for two operations from different threads to happen directly adjacent to each other but should not raise a data race because they are in a happened-before relation.

Spawn The spawn intrinsic creates a new thread. As we have already discussed in section Happened-Before 5.3.1 this synchronizes the threads and any operation that happens in the spawned thread happen strictly after the spawn.

Join The join intrinsic blocks the current thread until the thread it tries to join terminates. The terminating thread wakes up all threads that waited for it. This action once again synchronizes the threads, which is why we don't want any data races after this.

Lock handover When a lock is directly handed over to another thread this is also a synchronizing action. In the different happened-before rules from 5.3.1 we could see this as a synchronous message being sent from the releasing thread to the acquiring thread.

In our implementation we must track these synchronized threads to keep data races from being detected where there are none. The solution we propose is to keep a set of threads that have been synchronized by the last step. The data race detection only checks is the thread was not synchronized in the last step.

5.3.3 Implementation

Access struct is added, it contains all information we need about a memory access, the `AccessType`, the `Atomicity`, the `Address`, and the `Size`.

accesses field is added to the atomic memory to keep track of all accesses that happen within the current step.

check_data_races is a function used to compare a list of accesses with the current list of accesses to figure out if there are any data races between them.

reset_accesses is a function used to reset the current accesses. It returns the previous accesses.

step has to be changed to detect data races.

synchronized_threads is a set of threads that are synchronized in the current step. All actions they will do must happen after this step. This is needed to know when not to detect a data race.

The most important addition is the race detection. It works by matching every pair of accesses between the previous and the current list of accesses. The code with which the accesses compare looks as follow:

```rust
/// We assume they happen on different threads.
fn races(self, other: Self) -> bool {
    // At least one access modifies the data.
    if self.ty == AccessType::Load
        && other.ty == AccessType::Load
        { return false; }

    // At least one access is non atomic
    if self.atomicity == Atomicity::Atomic
        && other.atomicity == Atomicity::Atomic
        { return false; }

    // The accesses overlap.
    let end_addr = self.addr + self.len.bytes();
    let other_end_addr = other.addr + other.len.bytes();
    end_addr > other.addr && other_end_addr > self.addr
}
```

Listing 5.6: Data race detection logic [2]

Here we can clearly see the points with which we defined a data race in the first place. We check that at least one of them is a store operation, that at least one of them is non-atomic, and that they actually touch the same parts of memory. The only thing that is missing is that none of them happened-before the other. But as we have argued in Trace based detection 5.3.2 if they happen right after each other in the trace, they can not be in a happened-before order.

[2]Data race detection – `https://github.com/RalfJung/minirust/blob/e81f8100600a84afad7b501d299784bd483c5f41/spec/mem/atomic.md`

6 Evaluation

As we have already mentioned during the thesis there are some tests we wrote along side the implementation of concurrency. While specr lang is not executable we can use specr-transpile to turn it into minirust-rs. A randomized version of the interpreter in Rust. Which is why we can use Rust to directly run the interpreter.

6.1 Unit testing

For every intrinsic there are a number of things that can cause undefined behavior such as calling it with the wrong number of arguments or with the wrong type of arguments. For each of those we write an individual unit test to check that they are in order and check what we expect them to check. Such a test could look like this:

```
#[test]
fn spawn_arg_count() {
    let b0 = block!(
        Terminator::CallIntrinsic {
            intrinsic: Intrinsic::Spawn,
            arguments: list![],
            ret: None,
            next_block: Some(BbName(Name::from_internal(1))),
        }
    );
    let b1 = block!(exit());
    let f = function(Ret::No, 0, &[], &[b0, b1]);

    let p = program(&[f]);
    assert_ub(p,
        "invalid number of arguments for ‘Intrinsic::Spawn‘"
    )
}
```

Listing 6.1: Test spawn argument count.

This function creates a program with a single function, that tries to spawn a thread but does not supply any function pointer or data pointer. `assert_ub` then runs the program and expects the interpreter to exit with UB because there is an invalid number of arguments to the intrinsic.

6.2 Integration testing

We also did some integration testing to test that the concurrency works, the lock handover works, and the data race detection detects data races. All of those tests are interesting because they will not always succeed. The data race detection needs two racy operations to happen right after each other to detect them. We therefore have to hope that the scheduler puts them next to each other. Since the scheduler is uniformly random over the threads, we can calculate the probability of two instruction getting scheduled right after each other.

The test between a non-atomic write and a non-atomic read would look something like this:

```
static mut racy = 0;
static mut helper = 2;
fn second() {
    helper = racy;
}

fn main() {
    let id = spawn(second);
    racy = 1;
    join(id)
}
```

Listing 6.2: Data race test

It can be observed that the second thread will read from racy while the main thread will write to it, creating a data race. Something weird about our design is that we use a global helper to read from racy. Shouldn't this also be possible without a global?

Yes, this would also work fine with a local variable, but it has some implications we have to consider. For this let us calculate the probability of a single run detecting the data race. Let us assume that the racy instruction is after the n-th step of the first thread and after the m-th in the other. Initially we have done zero steps in both threads. We will now make the next move in a thread with probability $\frac{1}{2}$. We can use a binomial distribution to calculate the probability to reach the state in which both threads will execute the racy operation in the next step. $P[(i,j) = (n,m) \text{ in the trace}] = \binom{n+m}{n} \cdot \frac{1}{2}^{n+m}$

This is a necessary state to reach to then get a race. After this the scheduler must simply switch between the threads to catch the data race. In total this gives us a probability:

$$P[\text{Data race detected}] = \binom{n+m}{n} \cdot \frac{1}{2}^{n+m+1} \tag{6.1}$$

With just one instruction in front of either of the instructions or both decreases the probability from 0.5 to 0.25. When we now consider the amount of executions we need to get the probability of us missing the data race below $q, n = log(q)/log(1-p)$. Comparing 0.5 and 0.25 we get a factor of 2.41 between them. This is why we try to keep the number of instructions before the racy operations minimal.

Similar considerations were also made for the concurrency test where we want both the first and the second thread to be the one that writes later at some point during our iterations. Important for this test is, that both paths to the acquire about the same length. If one of them has a path length 3 while the other has one of length 0, the probability of the first one getting it can also be calculated with the binomial distribution and in that case would only be $\frac{1}{16} = 0.0625$. This design issue is the main reason why some of test seem weird. After considering this we got a failure probability of $\frac{1}{2}^{19} \approx 0.0002\%$.

The last important integration test we made was in response to the bug in the locking system 4.3. Here we want the lock to be handed over from one thread to the other. The threads will each take four steps to be in the acquired state. Once they are in the critical section they take more than $3 \cdot 256 = 768$. We want to approximate the probabilty that the lock handover does not occur. For this we analyze the chance that in $192 = 768/4$ steps, the other thread never gets to make a step: $p = 2^{-192}$. The probability that in one of those four 192 sized windows the other thread does not make progress is $1 - (1 - 2^{-192})^4 \approx 2^{-188}$. This gives us a very high probability that the handover must occur.

7 Conclusion

In this thesis we extended the MiniRust interpreter to define sequentially consistent concurrent programs. This is an important step in the formalization of Rust because any formal semantics that is not able to express concurrency is insufficient. During the design of the different components we had some interesting questions that needed to be answered.

The first is the decision of the concurrency model. Here we went with interleaving semantics. They are the best fit for MiniRust since the idea is that the semantics can be read just like an interpreter. Interleaving semantics allow us to do just that, they are powerful enough to be used for data race detection but still easy to understand with minimal impact to the rest of the interpreter.

Another design decision was the method for data race detection. There are many systems that can be used to detect a data race. One of them would have been vector clocks, a powerful method that is used to detect possible race conditions in a testing environment because it allows to detect them even if this execution happened to not get into one directly. The main issue with it would have been the added complexity to the system. This is why we decided to go with a trace based approach where we would detect data races if they happened within two adjacent steps. Sometimes, for example, if a thread gets spawned, there might be no data race to be detected. In such a case we added the synchronized threads, which are threads that have been synchronized during the step and should not be considered for a race in the next.

7.1 Future work

7.1.1 Weak Memory

We already mentioned that we only considered sequentially consistent atomic memory operations. This does not capture the whole concurrency spectrum Rust currently provides. Sometimes the programmer wants to use an atomic memory operation to communicate between threads, but the enforcement the sequential consistency asks for is to restrictive. In such a case we might want some more relaxed atomic operations.

Restriction Sequential consistency imposes the following restrictions on the program: Every memory operation that is before the operation in program order also happens before the atomic operation and analogously any operation after it happens after it.

These restrictions limit the optimizations the compiler can make and also limit the out-of-order execution of the processor and have with those two issues often great impact on the performance of concurrent code.

Main issue While these orderings are great to allow the programmer to write more performant code, they are very hard to specify and get right. The C++ specification for example allows for out-of-thin-air values. One option is to have a thread promise to execute a write in the future, thus enabling other threads to read from that write out of order like in [3].

Bibliography

[1] Programming languages — C. Standard, International Organization for Standardization, Geneva, CH, 2018.

[2] R. Jung. MiniRust. `https://github.com/RalfJung/minirust`. Accessed: 2023-08-16.

[3] J. Kang, C.-K. Hur, O. Lahav, V. Vafeiadis, and D. Dreyer. A promising semantics for relaxed-memory concurrency. *ACM SIGPLAN Notices*, 52(1):175–189, 2017.

[4] R. J. Krebbers. *The C standard formalized in Coq*. PhD thesis, Radboud University Nijmegen, 2015.

[5] NSA. CSI Software Memory Safety, 2022.

[6] W. Reisig. Partial order semantics versus interleaving semantics for csp—like languages and its impact on fairness. In *International Colloquium on Automata, Languages, and Programming*, pages 403–413. Springer, 1984.

[7] R. Schneider. Towards a formal and executable specification of Rust semantics, 2023.

[8] Stack Overflow. Stack Overflow developer survey 2023, 2023.