TECHNISCHE UNIVERSITÄT BERLIN

Towards a formal and executable specification of Rust semantics

A thesis presented for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

Berlin, May 2023

Rudi Schneider
matriculation number: 381662

First examiner and Supervisor: Prof. Dr. Ralf Jung
Second examiner: Prof. Dr. Uwe Nestmann

Sworn Affidavit

I hereby declare that the thesis submitted in my own, unaided work, completed without any unpermitted external help. Only the sources and resources listed were used.

[illegible]

Rudi Schneider
Berlin, 02.05.2023

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

[illegible]

Rudi Schneider
Berlin, den 02.05.2023

Abstract

Rust is a modern systems programming language designed to be a memory-safe alternative to fast languages like C and C++. It provides certain safety mechanisms that constrain the programmer to write code in a provably safe way. Yet some problems cannot nicely be solved within these limitations. To address these cases, Rust provides "unsafe Rust", which allows the programmer to partially bypass Rust's safety restrictions. When unsafe Rust is used incorrectly it might result in *undefined behavior*, which causes unexpected outputs of the program. The first step towards preventing programmers from running into undefined behavior is to develop an agreed-upon definition for UB. MiniRust is a vastly simplified version of Rust, that intends to define UB for unsafe Rust. In this thesis we make the MiniRust specification executable; we extend it to cover more Rust programs; we derive tools for it, including a Rust to MiniRust compiler; and finally, we prove some of MiniRusts internal consistency requirements in Coq.

Zusammenfassung

Rust ist eine moderne Programmiersprache, welche eine sichere Alternative zu performanten Sprachen wie C und C++ darstellt. Sie stellt bestimmte Sicherheitsmechanismen bereit, die Programmier dazu verpflichten auf eine beweisbar sichere Art zu programmieren. Manche Probleme lassen sich nicht gut innerhalb dieser Limitationen lösen. Um diese Fälle zu behaldeln stellt Rust die Subsprache "unsafe Rust" bereit, welche Programmern es erlaubt Teile der Sicherheitsrestriktionen zu umgehen. Das falsche Verwenden von "unsafe Rust" kann *undefiniertes Verhalten* nachsichziehen, welches unerwartete Ausgaben des Programms hervorruft. Der erste Schritt, um zu verhindern, dass Programmier undefiniertes Verhalten herbeiführen, ist eine klare Definition für undefiniertes Verhalten zu entwickeln. MiniRust ist eine deutlich vereinfache Version von Rust, welche eben dies anstrebt. In dieser Arbeit machen wir die MiniRust-Spezifikation ausführbar; wir erweitern sie, sodass sie mehr Rust Programme abbilden kann; wir entwickeln Werkzeuge für sie, unter Anderem ein Rust nach MiniRust compiler; und wir beweisen einige von MiniRusts internen Konsistenzanforderungen in Coq.

Contents

Chapter 1

Introduction

Rust is a modern systems programming language that promises to be blazingly fast, while also guaranteeing memory safety. The language achieves this by providing powerful abstractions, and performing numerous compile-time checks to guard against different kinds of problems.
In some scenarios however, the programmer might need to bypass these safety measures.This can be worthwhile whenever the fastest solution to a problem can not be expressed within the boundaries of safe Rust. To address these cases, Rust provides "unsafe Rust". Using unsafe Rust allows the programmer to lift certain restrictions of the language.It provides the programmer with direct control over the memory. This however means that programmers have to guarantee the memory safety of their software themselves again. Typically this is done by implementing a safe abstraction that internally makes use of unsafe Rust. This is encouraged, as it allows the programmer to verify the safety of an unsafe operation locally, independent of how the abstraction will be used in the rest of the code base. As you have to explicitly opt-in to unsafe Rust by enclosing potentially dangerous operations in an `unsafe` block, the usage of unsafe Rust is explicit and can be located effectively.
Misusing unsafe Rust can lead to *undefined behavior*, or UB for short. This refers to situations during the execution of a program, where the language does not specify a clear behavior. Typical examples for this are out-of-bounds array accesses, use-after free errors, dereference of a null pointer and so on. In these cases, the compiler is free to choose any behavior for your program. In the best-case scenario, this causes a crash which the programmer can then debug; but in the worst-case the software works fine, until some time in production.
The concept of undefined behavior can be exploited by compiler developers to optimize your code. Concretely, this means that all cases which lead to undefined behavior can immediately be eliminated. To make this more tanglible, consider this simple example:

```
if condition {
    // ... dereference a null pointer ...
    foo();
} else {
    bar();
}
```

If `condition` is `true`, the program will raise UB as it dereferences a null pointer. Hence, the given code snippet can be optimized to directly call `bar()` instead. This is legal, because the compiler is allowed to insert *any behavior it likes* in the case of UB. And in this case, calling `bar()` was the chosen behavior.

A precise definition of whether an unsafe Rust program has undefined behavior or not, is advantageous for everyone. Compiler developers can confidently and safely optimize out every case of undefined behavior that might come up; and Rust programmers can directly verify that their unsafe Rust segments are indeed safe.
Yet, finding a good specification for undefined behavior for Rust is hard. Defining undefined behavior in a too strict way, prohibits the programmer to do certain reasonable operations. Whereas defining undefined behavior in a too lax manner takes freedom from the compiler to optimize your code. Additionally, Rust is a very complex language. So formalizing undefined behavior for it is a very extensive task either way. Rust supports many features that are orthogonal to the question of whether a program has undefined behavior. Concepts like modules, generics, pattern matching are useful features of Rust, but when trying to define UB they can be considered unnecessary complexity.

This brings us to MiniRust. MiniRust is a drastically reduced version of Rust, that gets rid of all features that are unnecessary for defining unsafe Rusts behavior. Ralf Jung conceived MiniRust as part of his pursuit to formally specify Rusts semantics.
MiniRust is sufficiently simple, so that its semantics can be formally expressed using an acceptably concise definition. And by compiling a Rust program to MiniRust, we can lift MiniRusts semantics up to Rust. Effectively, defining Rusts semantics in terms of MiniRusts semantics. This is the key idea, on how MiniRust might help define Rusts semantics.
MiniRusts semantics itself is defined by an operational semantics - a step-by-step manual specifying how to execute MiniRust code. To make this semantics precise and testable, it was written as an interpreter. Next to simply running MiniRust code, this interpreter has the additional task detecting undefined behavior. And for non-deterministic programs, where a simple execution is insufficient to detect undefined behavior effectively, it provides a definition for UB instead.
In order to keep the interpreters code accessible, it was implemented in *specr lang*. This new dialect of Rust is intended to be simpler, and easier to read

than Rust itself. Rust puts a strong focus on performance, whereas specr lang prefers simplicity at the cost of speed.

Thesis Structure and Contributions

In this thesis, we have worked on improving MiniRust from many different angles.
First, we worked on making MiniRust executable, by implementing a compiler for specr lang. This allows us to run MiniRust programs, and lets us check whether the interpreter is implemented correctly. After introducing Rust and MiniRust in the Chapters 2 and 3, we present specr lang in Chapter 4, and we discuss the implementation of our specr lang compiler in Chapter 5.

In Chapter 6 we present our work on the MiniRust interpreter itself. We extended the MiniRust specification, so that it is able to cover a larger set of Rust programs correctly.

Recall that the core idea of MiniRust is to be a compilation target for Rust. This allows us to use MiniRusts semantics to specify Rusts semantics. Hence, in Chapter 7 we talk about `minimize`, our Rust to MiniRust compiler. Additionally, we discuss further tools implemented for MiniRust. These include a pretty printer and test suites.

And finally in Chapter 8, we tackled a few internal consistency properties of MiniRust. These properties govern how MiniRust represents values in memory. We formalized them in Coq, and proved them for all MiniRust types.

Related Work

Let us briefly mention a few projects related to MiniRust.

First and foremost, we want to mention Miri [Mir]. Miri is an interpreter for MIR - Rusts *mid-level intermediate representation.* It is designed to find undefined behavior in Rust programs, somewhat similarly to MiniRust. MiniRust itself is an *idealized* version of Miri. Whereas Miri is intended mostly to detect UB in practice, MiniRusts focus lies on formally and accessibly specifying UB.

Another related project is a-mir-formality [A m]. It too works on MIR, but in comparison to Miri, this project aims to define a full formal model for MIR. MiniRust differs from this, due to its specific focus on unsafe Rust.

Another project that notably influenced MiniRust is Cerberus [Cer]. Cerberus' goals are similar to those of MiniRust, although for the C programming language.

Chapter 2

Preliminaries I: Rust

In this Chapter, we will briefly introduce the reader to the Rust programming language. As we cannot squeeze a full detailed introduction to Rust into a preliminaries Chapter, we will limit ourselves to the Rust features relevant to this work. For a more universal introduction to Rust we refer to the Rust book, see `https://doc.rust-lang.org/book/`. This Chapter is intended for readers with no prior experience with Rust. So if you feel comfortable with Rust you may simply skip this Chapter.

2.1 Introduction

In this section, we will provide a few hands-on Rust programs to get the reader familiar with Rust syntax. So, let's get started.

Hello world

```
fn main() {
    println!("Hello world!");
}
```

This is a typical "Hello world" program. We declare a function named `main`, which only executes a single `println!` that prints the string "Hello world!" to the standard output. It is worth noting that `main` is a special name - Rust will look for a function with this name as an entry point to your program.

A recursive factorial function

```
fn factorial(n: u32) -> u32 {
    if n == 0 {
        1
    } else {
        factorial(n-1) * n
    }
}

fn main() {
    let n: u32 = 5;
    let fact_n = factorial(n);
    println!("The factorial of {n} is {fact_n}."); // prints "The factorial of 5 is 120."
}
```

This snippet defines a function `factorial`, which gets an argument `n` of type `u32` and also returns a `u32`. The type `u32` represents an unsigned integer of 32 bits. [1] The implementation of `factorial` is a typical recursive implementation.

- `factorial(0)` simply returns `1`, whereas
- `factorial(n)` for `n > 0` returns `factorial(n-1) * n`

The main function declares two variables `n` and `fact_n`. Note that `n` gets an explicit type annotation `n: u32`, whereas `fact_n` does not. In many contexts Rust can infer the types of variables automatically, so they can be left out.

2.2 Structs

Structs allow you to bundle your data into compound objects.

```
struct Person {
    name: String,
    age: u32,
}
```

This struct expresses that a `Person` consists of a name and an age.

```
fn main() {
    let x = Person {
        name: String::from("John"),
        age: 21,
    };
    println!("My name is {}", x.name);
    println!("I am {} years old", x.age)
}
```

This is how you construct a `Person`. We constructed a person named ”John” at the age of 21. The expressions `x.name` and `x.age` are used to access the fields of a struct.
A close relative of the struct is the tuple:

[1] Rust provides `u8`, `u16`, `u32`, `u64`, `u128` for unsigned integers with different sizes, and respectively `i8`, `i16`, `i32`, `i64`, `i128` for *signed* integers. Finally there are `usize` and `isize` to represent unsigned and signed integers with the same number of bits as a pointer.

```
fn main() {
    let tuple: (i32, i32) = (2, 3);
    println!("The numbers are {} and {}", tuple.0, tuple.1);
}
```

Tuples store multiple values as well. The inner values can be accessed using `.0` or `.1` and so on.

2.3 Enums

An `enum` allows you to define types, that hold values chosen from a set of distinct variants. These types are generally called *sum types*!
Let us consider an example.

```
enum OptionNumber {
    SomeNum(i32),
    NoNum,
}
```

Here we define a new type named `OptionNumber`. An `OptionNumber` may either be `SomeNum` equipped with a given `i32`, or it might be a `NoNum` without a number.

```
fn unwrap_or_default(x: OptionNumber, default: i32) -> i32 {
    match x {
        OptionNumber::SomeNum(n) => n,
        OptionNumber::NoNum => default,
    }
}
```

We define a function `unwrap_or_default` using an `OptionNumber`. If the given `OptionNumber` stores a number we return it, and otherwise we return the number `default`. This `match` block is the typical way you work with enums, it allows us to safely access the values stored within enums.

```
fn main() {
    let opt = OptionNumber::SomeNum(42);
    println!("{}", unwrap_or_default(opt, 0)); // prints 42

    let opt = OptionNumber::NoNum;
    println!("{}", unwrap_or_default(opt, 0)); // prints 0
}
```

In this code snippet we call `unwrap_or_default` twice, once with a number and once without it.

2.4 Generics

Now, we might intend to use `OptionNumber` and its functions but we don't want it to store an `i32`, but some other types. This is where we need *generics*:

```
enum Option<T> {
    Some(T),
    None,
}
```

Our new type `Option<T>` is the generic version of `OptionNumber`. The type `Option<T>` we have just defined is used very regularly in Rust. It is defined in Rusts standard library, see here.
Not only types can be generic, but functions can be generic too:

```
fn unwrap_or_default<T>(x: Option<T>, default: T) -> T {
    match x {
        Option::Some(n) => n,
        Option::None => default,
    }
}
```

The above code is the generic version of our `unwrap_or_default` function from the last section.

2.5 Traits

Another fundamental concept of Rust are *traits*. They allow you to define common functionality across multiple types.

```
trait ToString {
    fn to_string(&self) -> String;
}
```

This `trait` is intended to be implemented by types that are convertible to a `String`.
Thus in order to implement to `ToString` for a type, we need to specify a `to_string` method.

```
struct Person {
    name: String,
    age: u32,
}

impl ToString for Person {
    fn to_string(&self) -> String {
        format!("{} ({})", self.name, self.age)
    }
}
```

We can now convert a `Person` to a string by using the `to_string` method.

```
fn main() {
    let p = Person {
        name: String::from("Clara"),
        age: 21,
    };
    println!("{}", p.to_string()); // "Clara (21)"
}
```

The real power of traits becomes apparent when used in conjunction with generics. When implementing a generic function, you can use the syntax `<T: ToString>` to restrict your function to be only applicable for the types `T` that implement `ToString`. Adding this constraint allows you to use the `to_string` method on values of the type `T`:

```
fn print<T: ToString>(t: T) {
    println!("{}", t.to_string());
}
```

2.6 Move semantics

In this section, we talk about what happens to the variable `x` when we do any of the following operations:

- `let y = x;`
- `foo(x);`
- `let o = Object { field: x, };`

What all of these code snippets have in common, is that we intend to give `x` to a new location in memory. Either by putting it into a new variable, giving it to a function, or storing it within a struct.

Copy types

For most types we have encountered so far, the answer to this question is clear: The value from `x` will be copied byte-by-byte over to the new location. This, for example happens for integers.
Generally, any type that implements Rusts marker trait `Copy` is allowed to be copied byte-by-byte, as it simply contains *plain old data*.

Heap allocations

Let us briefly present a few types, where this is not so easy.

- `Box<T>`: A pointer type that uniquely owns a heap allocation of type `T`, and

- `Vec<T>`: A contiguous growable array of `T`, allocated on the heap.
- `String`: A UTF-8–encoded, growable string.

What these types have in common is that they internally store a pointer, whereas the actual data lies on the heap. Now we want to argue that *byte-by-byte* copying is not acceptable for those types. For example, let us consider a `Vec`. When we byte-by-byte copy it, we end up having two different Vecs pointing to the same memory. If one `Vec` is mutated, re-allocated, or freed - the other `Vec` could be corrupted:

```rust
let mut v = vec![1, 2, 3];
let v2 = v;
v.push(4); // might reallocate
println!("{}", v2[0]); // -> might be undefined behavior
```

In this scenario, `v` could re-allocate its memory while freeing the original data. This would leave `v2` in an invalid state!

Move semantics

Rust's solution for this problem is that `v` is not allowed to be used after this operation. This is called a *move*. Effectively we have moved the `Vec` from `v` to `v2`. Trying to access `v` after the move will raise a compile-time error.

Clone

Sometimes though, you might not want to move your data at all. You might prefer it to be cloned instead. In scenarios like these, one can make use of the `Clone` trait. An implementor of the `Clone` trait has to provide a `clone` method, which clones the data. All heap-allocated types we have covered before implement `Clone`, so using ...

```rust
let v2 = v.clone();
```

... does the trick.

2.7 References

If you intend to point to some piece of data without owning it, references are typically the way to go. References are effectively pointers, but with certain safety guarantees.

```rust
fn main() {
    let mut x = 2;

    let y = &mut x; // `y` is a mutable reference to `x`
    *y = 3;
    let z = &x; // `y` is an immutable reference to `x`
    println!("{}", *z);
}
```

In this example we use references to set `x` to three, and then print `x`.
Again, References have certain safety mechanisms; so you might be wondering what happens if we try to take a reference of something that is freed too early.

```
fn main() {
    let y: &i32;

    {
        let x = 2;
        y = &x;
    } // `x` is freed here, but `y` is still pointing to `x`!

    println!("{}", *y);
}
```

You will find that the Rust compiler rejects this code with the error message `` `x` does not live long enough``: Crisis averted! But, in some occasions, the rules that make references safe will turn out to be too restrictive to express what you wanted to express. In occasions like these, one has to make use of the `unsafe` alternative *raw pointers*..

2.8 Unsafe

Rust is a language with a special focus on safety, hence it is quite restrictive in what it allows you to do. Yet sometimes those restrictions might not work for your particular use-case. For example, if you wish to get the last bits of performance. In this case, you might require `unsafe` Rust. Unsafe Rust allows you to do certain *unsafe* operations, like dereferencing a raw pointer and reading from a union field. To opt-in to unsafe Rust, one has to put a `unsafe { ... }` block around the unsafe operation that one is willing to do.
Unsafe Rust should generally be avoided if there is a safe alternative, but as MiniRust is trying to specify how unsafe Rust behaves, it is highly relevant to this Master's Thesis.

Raw pointers

Unsafe Rust allows you to dereference a raw pointer.

```
fn main() {
    let y: *const i32;
    {
        let x = 2;
        y = &x as *const i32;
    } // `x` is freed here, but `y` is still pointing to `x`!

    println!("{}", unsafe { *y });
}
```

This example is the same as 2.7, but now we use raw pointers instead of references. If you try to compile this, the Rust compiler will be happy to do so - as we are now using pointers instead of references. But when you run this code it will result in undefined behavior.

Unions

Another thing that is possible within `unsafe` is the use of `unions`:

```
union IntOrBool {
    i: i64,
    b: bool,
}
```

The union `IntOrBool` stores either an integer or a boolean. The important difference between `enums` and `unions` is that a `union` does not know which of both it stores. So you might access the wrong field:

```
fn danger() {
    let x = IntOrBool { i: 42 };
    unsafe {
        println!("{}", x.b);
    }
}
```

This function `danger` defines `x` to be an `IntOrBool` storing an integer. But then it prints the boolean inside of this union. This is undefined behavior.

Chapter 3

Preliminaries II: MiniRust

In this Chapter we will provide a brief introduction to *MiniRust*. We will mostly explain MiniRust as a language, and only occassionally provide some insights into the interpreter. If the reader has further interest in the implementation of the interpreter, we recommand taking a look into MiniRusts code at `https://github.com/RalfJung/minirust`. After all, this implementation is written to be read!

3.1 Functions

The first central concept we intend to explain are *functions*. A MiniRust function is somewhat similar to a Rust function, although quite a bit more low-level. A function consists of two things:

- The locals
- The instructions

A *local* is a piece of memory that the function can use to store and read intermediate results. A local variable (including function arguments) in Rust will typically become a local in MiniRust. But there are some additional locals in MiniRust, namely

- the return local, which stores the return value of the function, and
- temporary values that the compiler needed to store somewhere

Just like MIR and LLVM IR, we will follow the convention of naming locals as an underscore followed by a number: `_0`, `_1` etc.
The instructions of a function are stored within *BasicBlocks*. A BasicBlock is a sequence of *Statements* that are executed in order, followed by a *Terminator* that decides what to do after this BasicBlock.
Let us consider the first example of a MiniRust function:

```
fn fourty_two() -> _0 {
    let _0: i32@align(4);
    start bb0:
        _0 = 42;
        return;
}
```

Figure 3.1: A simple MiniRust function

This function simply returns the number 42. We will go through it line by line to explain what happens:

- `fn fourty_two() -> _0`
  We declare the function `fourty_two` with no arguments, and `_0` as its return local
- `let _0: i32@align(4);`
  We declare the local `_0`. It stores one `i32`, further it has to be allocated with alignment 4. This means that the address of `_0` is required to be divisible by four.
  The integer types of MiniRust are the same as their Rust equivalents. Hence also `u8`, `i128`, `usize`, `isize` etc. are supported in MiniRust.
- `start bb0:`
  We declare a basic block `bb0`. `start` means that this is the first basic block being executed when this function gets called.
- `_0 = 42;`
  This is an `assign` statement, it stores the number 42 within the local `_0`.
- `return;`
  This is the `return` Terminator. It gives back control to the function that called `fourty_two` in the first place.

3.2 Value expressions vs place expressions

Consider this new function `identity`. It will return the number it receives as input.

```
fn identity(_1) -> _0 {
    let _0: i32@align(4);
    let _1: i32@align(4);
    start bb0:
        _0 = load(_1);
        return;
}
```

Figure 3.2: The `identity` function

This example contains two new things. First-of-all, we have a function argument local, namely `_1`. Secondary, the assign statement `_0 = load(_1)` is new. The expression `load(_1)` reads the value stored within the local `_1`. There is a very important distinction between `_1` and `load(_1)`. This is the distinction between *place expressions* and *value expressions*. `load(_1)` or for example the number `42` are *value expressions*. On the other hand `_1` is a *place expressions*, which merely stores a value. One can think of places as *pointers* to a piece of memory. Hence, one fundamental difference is that you cannot write to a value, but you can write to a place. The assign statement stores a value within a place. For example, `_0 = 42;`.

Analog to the distinction between place expressions and value expressions, there is a distinction between place types and value types (or just "types"). You have already seen a place type in action, the line `let _0: i32@align(4);` associates a place type to the local `_0`. A place type is just an ordinary type in conjunction with an alignment requirement. Recall that a place can be seen as a pointer: An alignment requirement forces the address of that pointer to be divisible by a given divisor. Currently, we only considered the type `i32`, but in the following sections, this will change.

3.3 Program

A MiniRust program is a collection of functions with one dedicated `start` function. It is the fundamental unit that can be executed.

```
start fn main() -> _0 {
    let _0: bool@align(1);
    start bb0:
        _0 = calculation(42) -> bb1;
    bb1:
        return;
}

fn calculation(_1) -> _0 {
    let _0: i32@align(4);
    let _1: i32@align(4);
    start bb0:
        _0 = /* some computation */;
        return;
}
```

The novel thing to see here is that there is a `start fn`. [1] Just as a `start` basic block is the first basic block being executed in a function, the `start` function is the first function being executed in a program.
Additionally, the snippet contains this: `_0 = calculation(42) -> bb1` - the function call Terminator. It means that we call the function `calculation` with the argument `42`, store its return value within `_0`, and then jump to the basic block `bb1`.

3.4 Machine

In the last Section we have presented the `Program`, MiniRusts central unit of execution. Generally, in all previous Sections we introduced MiniRust - the language. But now we will take a small peak into MiniRusts interpreter.
In order to define how this execution works, we need to store the state of the program somewhere. This is where the `Machine` comes in. A machine fully expresses the state of a program at a particular point in its execution. Among other things, the `Machine` consists of the program, the Memory and the stack frame with its instruction pointer.
A `Machine` can be constructed using the `new` function, which gets a `Program` as an argument. It will set everything up to make the program start. Also, it will reject programs that have certain statically detectable flaws using the *well-formedness checker*. This checker will find type errors, usage of dead locals and similar problems.
Next to the `new` function, `Machine` provides a `step` method. This method will execute the next instruction of the program. When the program terminates, `step` will return a `TerminationInfo`. This `TerminationInfo` informs us about the way to program terminated: Whether it was a graceful shutdown, or whether undefined behavior has been raised.

3.5 Loops

In this subsection, we will introduce a few more of MiniRusts control flow primitives. Concretely we will learn how to express loops in MiniRust.
We start by an iterative algoritm to calculate whether a number is even. As we want to show some control flow, we will not make use of the modulus operator, and instead subtract 2 until the number is either 0 or 1.

[1]For the sake of completeness, we want to mention that during the development of this thesis, the rules for `start fn` changed. The `start fn main` would be considered ill-formed by the current version of MiniRust. For more details, check 6.1.

```
fn is_even(_1) -> _0 {
    let _0: bool@align(1);
    let _1: u64@align(8);
    start bb0:
        if load(_1) < 2 {
            goto -> bb2;
        } else {
            goto -> bb3;
        }
    bb2:
        _0 = load(_1) == 0;
        return;
    bb3:
        _1 = load(_1) -<u64> 2;
        goto -> bb0;
}
```

Figure 3.3: An iterative `is_even` function

Again, we explain the new concepts:

- `bool@align(1)`
  We have seen another type! Namely `bool`.
- `load(_2) < 2`
  This is an integer comparison. It compares whether the number stored within `_2` is less than 2, and returns `true` or `false` accordingly.
- `if load(_2) < 2`
  `if` is another Terminator: It receives a value expression of type bool as argument, and jumps to either one of the two basic blocks.
- `_0 = load(_1) == 0;`
  The == comparison checks whether two numbers have the same value, and returns a boolean accordingly.
- `load(_1) -<u64> 2`
  This is an integer subtraction. It subtracts two from the number stored in `_1`. The generic parameter `<u64>` expresses that the output type of this subtraction is a `u64`.

3.6 Interlude: Encoding of values

Up to this point, we enjoyed a rather high-level perspective on MiniRusts values. We worked with integers and booleans without the need to specify how they are laid out in memory. However, when the programmer starts to use certain low-level operations, this model will fall apart. To name an example, transmuting one type into another is common in low-level programming, but MiniRust can only support it if it represents values as byte sequences.

MiniRust provides two representations of data

- The high-level *value* representation
  It considers everything like a human would think about it. Any integer is just a mathematical integer. A boolean is either `true` or `false` and so on.

- The low-level byte representation
  Everything is a sequence of bytes in memory.

Each MiniRust type provides an `encode` and a `decode` function that converts between these two representations. The function `encode` will convert a value to a list of bytes, and `decode` will do the converse.

Encoding and decoding of values in MiniRust are facilitated in the following operations:

- The `load(p)` value expression:
  It *decodes* the bytes stored within the place `p`. If this byte pattern does not represent any value, it raises undefined behavior.

- The `assign` statement `p = v`:
  The value `v` will be stored in the place `p`. To do that, the value needs to be *encoded* first.

The way MiniRust understands memory is, in practice, a bit more involved than an ordinary sequence of bytes. To motivate this, consider the following example:

```
fn ub() -> _0 {
    let _0: u8@align(1);
    start bb0:
        _0 = load(_0);
        return;
}
```

Figure 3.4: This function raises UB.

In this function, we load `_0` before initializing it: Thus it is clear that this function has undefined behavior, so MiniRust *should* raise UB here. But the question is, how does MiniRust detect that this is undefined behavior? It does not matter which byte the memory stores at `_0`, every byte can be flawlessly decoded into a `u8`. To solve this, the memory can not be understood as storing sequences of raw bytes - these bytes need some additional information. This brings us to the so-called *AbstractByte*: It either stores a byte, or it is *uninitialized*.

By representing our memory as a sequence of `AbstractByte`, we are now able to detect uninitialized reads! Similarly, the functions `encode` and `decode` will use `AbstractByte` instead of plain bytes. We will attach even more information to our AbstractBytes, but more on that later.

3.7 Composite types

So far we've only seen MiniRusts integer and boolean types. In this subsection we will introduce three new kinds of composite types that MiniRust supports. Note that MiniRust does not yet support enums!

Arrays

Arrays are the simplest kind of composite type. They allow for a new place expression, namely indexing: `array[index]`.
We will demonstrate arrays by yet another example function that simply returns 42. This time, it will create the array `[41, 42, 43]` and index it at `1`. The array will be stored in `_1`:

```
fn fourty_two_v3() -> _0 {
    let _0: u64@align(8);
    let _1: [u64; 3]@align(8);
    start bb0:
        storage_live(_1);
        _1[0] = 41;
        _1[1] = 42;
        _1[2] = 43;
        _0 = load(_1[1]));
        storage_dead(_1);
        return;
}
```

Note that reading or writing to an array outside of its bounds will raise undefined behavior.

Tuples

Both Rust tuples and Rust structs can be understood as a MiniRust tuple. A MiniRust tuple stores a list of types, but additionally it stores them at explicitly given offsets.

```
tuple T0 (8 bytes) {
  at byte 0: i32,
  at byte 4: i32,
}

tuple T1 (16 bytes) {
  at byte 0: i32,
  at byte 8: i32,
}
```

Figure 3.5: Two simple tuple types.

This first definition defines the type `T0`: It has a size of 8 bytes, and two `i32` fields at offsets 0 and 4.
The second tuple type is very similar, even though it has larger amounts of *padding*, i.e. unused space within the tuple.
Similar to arrays, we also have a place expression to access fields of a tuple.

```
fn fourty_two_v4() -> _0 {
    let _0: u64@align(8);
    let _1: T0@align(4); // see the T0 definition from above!
    start bb0:
        storage_live(_1);
        _1.0 = 41;
        _1.1 = 42;
        _0 = load(_1.1);
        return;

}
```

Unions

The last kind of composite type that MiniRust supports are `unions`.

```
union T0 (8 bytes) {
    at byte 0: u64,
    at byte 0: bool,
}

fn ub() -> _0 {
    let _0: bool@align(1);
    let _1: T0@align(8);
    start bb0:
        storage_live(_1);
        _1.0 = 42;
        _0 = load(_1.1);
        storage_dead(_1);
        return;
}
```

First, we declare a union `T0`. This looks pretty similar to a tuple definition, the only difference is that fields are allowed to overlap. In our scenario, the two fields are even at the same offset (namely zero). The function `ub` creates a union `T0` and writes `42` into its `u64` field. It then returns, the `bool` interpretation of this. This raises undefined behavior, as a bool can only be 0 or 1.

3.8 Pointers

We will now introduce pointers, again we will do so by example:

```
fn fourty_two_v2() -> _0 {
    let _0: i32@align(4);
    let _1: (*layout(size=4, align=4))@align(8);
    start bb0:
        _1 = &raw _0;
        deref<i32@align(4)>(load(_1)) = 42;
        return;
}
```

This function takes a pointer to `_0`, and then it writes 42 to it. The new local `_1` will store this pointer.
There are again three new concepts to explain in this example:

- `let _1: (*layout(size=4, align=4))@align(8);`
  Let us first consider the pointer type `*layout(size=4, align=4)`: One can think of this type as `*mut i32`, but MiniRusts pointers are not type-safe. They only store the *layout* information of the inner type. It consists of the size and its alignment constraints. Hence the exact inner type `i32`

will be forgotten, and only `layout(size=4, align=4)` will be remembered.

- `_1 = &raw _0;`
  The `&raw _0` is an `addr_of` value expression. It converts a place to a pointer value pointing to that place. This statement stores the raw pointer to `_0` in `_1`.

- `deref<i32@align(4)>(load(_1)) = 42;`
  The value `load(_1)` is a pointer value pointing to some place in memory. The place expression `deref` can be used to address this place. As MiniRust only remembers the layout of the inner type, `deref` needs to explicitly state the inner place type again.

Note that there are two kinds of pointers at play here: Places can be understood as pointers to values, but additionally some values are pointers as well. `addr_of` and `deref` are used to convert between the place pointer world and the value pointer world.

For the sake of completeness, let us briefly mention two further pointer operations that MiniRust supports.

- *pointer offset*:
  In addition to dereferencing pointers, we want to be able to offset pointers.

- *Integer-Pointer casts*:
  Another important feature is the possibility to cast pointers to integers and back using the `ptr2int` and `int2ptr` expressions.

3.9 Provenance

We have now established the basics of pointers. Yet there is one issue: Our current model allows pointers from one allocation to be offseted into another allocation, and dereferenced there. For example, if we had a pointer `x` and a pointer `y`: If we now offset the pointer `x` by `y - x`, we will end up in allocation `y`. This might not look like a problem, but if we allow programmers to do that, certain compiler optimizations are not possible anymore.
Let us consider an example:

```
x = 2;
*y.offset(i) = 3;
return x;
```

When the compiler sees this code, it cannot optimize this to simply return 2, because a write to `y.offset(i)` *might* actually write to `x`. [2]

[2] See a more complete example at [Jun]

In practice, this causes miscompilations by major compilers, as they wrongly assume that x and y are pointing to distinct allocations. To combat this problem, we forbid the programmer to change the allocation by offsetting.
For MiniRust to detect, whether pointers change their allocations, a pointer has to keep track of the so-called *provenance*. The provenance remembers from which allocation the pointer originally came. And, whenever a pointer is dereferenced, it will check whether it is in-bounds in its original allocation.
As pointers that are stored in memory need to remember their allocation, we need to extend the `AbstractByte` by provenance.
Now, we can present the final definition an `AbstractByte`: [3]

```
enum AbstractByte {
    Uninit,
    Init(u8, Option<Provenance>),
}
```

3.10 Nondeterminism

So far we have had the luxury that MiniRust was able to find each case of undefined behavior simply by executing the program once. Yet, when we enter the realm of non-determinism this luxury fades.
Let us consider an example:

```
fn ub_or_not() -> _0 {
    let _0: bool@align(1);
    let _1: usize@align(8);
    start bb0:
        _1 = ptr2int(&raw _0);
        if load(_1) % 2 == 0 {
            goto -> bb1;
        } else {
            goto -> bb2;
        }
    bb1:
        _0 = load(_0);
        return;
    bb2:
        _0 = true;
        return;

}
```

[3]In MiniRust, `AbstractBytes` are actually generic over the provenance. For the sake of simplicity we leave that out.

This function checks whether the address of `_0` is even or not. If it is even, the function goes to basic block `bb1` where it raises UB due to an uninitialized load of `_0`. If it is odd, the function returns gracefully.
Memory allocation is fundamentally non-deterministic in MiniRust. The address that is chosen for a new memory allocation is selected non-deterministically from the set of feasible addresses. Thus some executions of this program might allocate `_0` at an odd - and some on an even address.

Definition 1 (Memory allocation non-determinism)**.** *A program doing a memory allocation is said to have undefined behavior whenever it raises undefined behavior for at least one feasible address.*

Following this definition, the program given before does indeed have undefined behavior, because there is a non-deterministic choice that makes it raise undefined behavior.
It is worth noting that most previously shown programs were already non-deterministic. In fact, every program that executes a function having any allocated local is non-deterministic, as the local needs to be allocated in memory. But for previous programs this non-deterministic choice did not affect the outcome of the program.

int-to-pointer casts

Another source of non-determinism is int-to-pointer casts. Recall that provenance is used to remember the allocation from which certain pointers come. For int-to-pointer casts, this is not so easy. The original allocation is not known when casting an int to a pointer. MiniRusts solution is to predict the provenance from a feasible set of previously exposed provenances.

Definition 2 (int-to-pointer cast non-determinism)**.** *A program doing an int-to-pointer cast is said to have undefined behavior whenever it has undefined beaviour for all of the feasible provenances.*

There is an important distinction between those two definitions. Memory allocation is interpreted *daemonically.* This means that a program has UB whenever one of the choices raises UB. On the other hand, int-to-pointer casts are interpreted *angelically.* This means that a program has UB whenever all of the choices raise UB.

Chapter 4

specr lang

Recall from the Introduction that MiniRust is written in a new language, a dialect of Rust named *specr lang*. This language is designed to be readable by a Rust-knowing audience. But where Rust prefers performance, specr lang values readability and simplicity.
In this Chapter, we will explain how specr lang differs from Rust, and in the next one, we'll talk about the specr lang to Rust compiler.
Generally, Rust is quite explicit about anything that might affect performance. For example:

- memory layout of data
- where clones happen
- memory management

specr lang on the other hand, tries to abstract over those things, and intends to redirect focus towards what is being expressed using the software.
Note that the exact set of features specr lang were to offer wasn't clear before this thesis. In the following sections we will see what it converged to.

4.1 Implicit Copying

Recall from the section 2.6 that in Rust, types a *moved* instead of being *cloned* per default. So, whenever you don't want a value to move, but instead you intend to clone it, you need to call the `.clone()` method.
This default makes sense for Rust, as often enough you move out of a variable which you don't even need afterwards. So inserting a move is much faster and less resource consuming, than inserting a full clone of your data.
Still, move semantics per default is not a great idea for specr lang, as specr lang intends to be readable and concise. The explicit cloning is more verbose than necessary, and hence should not be part of specr lang. This is why specr lang does not have move semantics per default, but instead it will copy values implicitly.

4.2 Recursive enums

In Rust, enums cannot recurse without any form of indirection. This poses problems for the MiniRust specification, as it requires a lot of recursive enums: `ValueExpr`, `PlaceExpr`, `Type`, `Value` etc. To explain why Rust forbids this, let us look at a simple example. This simple enum - defining the natural numbers `Nat` - is not valid in Rust:

```
enum Nat {
    Zero,
    Succ(Nat),
}
```

Rust forbids this, as a `Nat` needs to store

- at least one bit to differentiate between `Zero` and `Succ`
- and `sizeof(Nat)`-many bytes to store the `Nat` for the `Succ` variant

So we quickly end-up with the equation
`sizeof(Nat) = 1 + sizeof(Nat)` for which there can be no solution.
So what you typically have to do in Rust to make this work, is explicitly inserting an indirection:

```
enum Nat {
    Zero,
    Succ(Box<Nat>),
}
```

This is now valid in Rust, as `Nat` doesn't need to store a whole other `Nat` but only a pointer. Next to the fact that we have just implemented the natural numbers as as linked list, which is a crime in of itself,
there are other problems with this approach:
When using this `Nat` type, we have to always work around this `Box`:

```
fn succ(n: Nat) -> Nat {
    Nat::Succ(Box::new(n))
}

fn pred(n: nat) -> Nat {
    match n {
        Nat::Zero => Nat::Zero,
        Nat::Succ(m) => *m,
    }
}
```

So, when constructing a `Nat::Succ` we need to create a Box using `Box::new`, and when matching upon a `Nat::Succ` we need to dereference the Box to access the `Nat`.
As specr lang highly values conciseness, this is not acceptable for us.
This is why enum indirection can be done as follows in specr lang:

```
enum Nat {
    Zero,
    Succ(#[specr::indirection] Nat),
}

fn succ(n: Nat) -> Nat {
    Nat::Succ(n)
}

fn pred(n: nat) -> Nat {
    match n {
        Nat::Zero => Nat::Zero,
        Nat::Succ(m) => m,
    }
}
```

So you need to still explictly insert an indirection using the attribute `#[specr::indirection]`, but when constructing or matching upon the `Nat::Succ` variant, you can simply ignore the fact that the `Nat` is behind an indirection.

4.3 Literate programming

A central goal of specr lang is to be easily understandable, and well documented. So, choosing plain code as a medium might not be the best solution.
Thus instead, specr lang makes use of a certain form of *literate programming* (see [Knu84]). This entails that specr lang is primarily written in a text document which explains the design of the software. And within those explanations we embed snippets of specr lang code.
We chose `markdown` as a host language for our text document.
Generally, specr lang code is embedded into markdown files. And code blocks are used to express

~~~
```rust
...
```
~~~

This allows MiniRust to write long comments in markdown, and similar things. It's worth noting that specr lang has a module system, so not all snippets are in "the same module". specr lang has a one-to-one correspondence between directories and modules. So if two files are in the same directory, they are in the same module. This is a difference to Rust, as each Rust file opens its own module. This difference was taken deliberately in order to lessen the noise of importing things.

4.4 Breaking down large pieces of code

In order for literate programming to work nicely, the underlying source code should not have too large blocks of code. This is why specr lang provides a few features that allow you to decompose your code into smaller pieces.

Argmatch

MiniRust has quite a few methods that require to match over a large number of cases, where additionally each case does a non-trivial amount of work. This can lead to quite unreadable code, where the reader might quickly lose track of what you are actually implementing. To solve this problem, specr lang allows you to split up match blocks into different functions, hence whenever another match arm begins the full context is presented again, so you are not as easily lost. Syntactically, this looks as follows:

```
impl Server {
    fn handle(&mut self, msg: Message) {
        match msg {
            Message::NewClient(ip_addr) => {
                // ...
                // ...
                // ...
            },
            Message::ClientDisconnected(client) => {
                // ...
                // ...
                // ...
            },
        }
    }
}

// ... can be equivalently written in specr lang as ...
```

```rust
impl Server {
    #[specr::argmatch(msg)]
    fn handle(&mut self, msg: Message) { .. }
}

impl Server {
    fn handle(&mut self, Message::NewClient(ip_addr): Message) {
        // ...
        // ...
        // ...
    }
}

impl Server {
    fn handle(&mut self, Message::ClientDisconnected(client): Message) {
        // ...
        // ...
        // ...
    }
}
```

Figure 4.1: Argmatch Example: handling a message

Separated trait impls

Similar to argmatch splitting up large match blocks, we also want to split up large trait implementations. In Rust, implementing a trait for a type needs to be done in one large impl block. In specr lang, you can separate that block into as many subblocks as you need to make it readable.

```
impl Foo for () {
  fn foo1(&self) { ... }
  fn foo2(&self) { ... }
}

// ... can be equivalently written in specr lang as follows:

impl Foo for () {
  fn foo1(&self) { ... }
}

impl Foo for () {
  fn foo2(&self) { ... }
}
```

Figure 4.2: Separated trait impls Example

4.5 The standard library "libspecr"

specr lang provides a standard library called libspecr, see `https://docs.rs/libspecr`. The items exposed by libspecr can roughly be separated into three categories:

- *General purpose data structures*: For example collections like `List<T>`, `Set<T>`, `Map<T>`, and the mathematical integer `Int` would fall into this category.
- *specification-specific items*: This category covers things that are generally useful when writing a low-level specification. For example `Align`, `Size`, `Endianness` and similar things are provided here.
- *The non-determinism framework*: You might recall from the MiniRust introduction, that MiniRust is fundamentally non-deterministic (see 3.10). libspecr provides two primitives `pick` and `predict` to non-deterministically choose from a feasible set of options, either *deamonically* or *angelically*.

Chapter 5

Compiling specr lang to Rust

In this Chapter, we focus on making MiniRust executable. Recall that MiniRust is written in *specr lang*, a Rust dialect with a strong focus on readability. At the beginning of this project, specr lang had no interepreter or compiler, or even checker. This means that you cannot "run" MiniRust, and it is only usable to be read. This is what we changed in this Chapter.
Let us briefly summarize the general procedure of the compiler: The tool `specr-transpile` will

- look for markdown files,
- filter out their specr lang code snippets,
- compile them to Rust,
- and collect the results into a Rust library crate.[1]

Where to look for input files, and where to generate the output crate can be configured by using a `specr-transpile` config file. The generated Rust library crate has a dependency on `libspecr`, the standard library and runtime of specr lang.
Now the interesting part is how the "compile them to Rust" step works concretely: One solution would be to write a full compiler with parser, then name resolution macro expansion, type deduction etc. which will finally result in Rust. Yet, this doesn't make sense, as our target language is already Rust - and it takes forever to write such a compiler. Hence, we decided to only use syntactical transformations. We use Rusts parser `syn` (see [Syn]) to parse the input. This is possible as specr lang is syntactically similar enough to Rust so that the parser could not tell the difference. After the transformations on the

[1] A Rust *crate* is a compilation unit. Intuitively it can be understood as a Rust project.

AST are done, we use the `prettyplease` crate ([Pre]) to convert the AST back to a String.
Doing all operations on a syntactical level has quite a few disadvantages though. We will not be able to know the type of a variable, we won't be able to resolve names (i.e. if there are multiple items named X, we won't be sure which is which). This makes the transformations pretty hard, as they should work even though the code we work with is quite opaque.
The transformations for `argmatch` and `separated trait impls` work by converting the specr lang code to the respective Rust equivalent provided in the figures 4.1 and 4.2. As these translations are not too interesting, we will not cover them in more detail. One the other hand some concepts like `implicit copy` and `enum indirection` require more involved solutions, and hence will receive more attention in this Chapter.

5.1 Implicit Copying

Recall that specr lang does not inherit Rusts move semantics, but instead of moving specr lang implicitly copies the value.
The options to achieve this we considered were:

add .clone() where necessary

The first idea, we had was that we should tranform the code in a way that adds `.clone()` whenever an expression might be copied. Generally, cloning something that already was a reference, is not really a problem. (As immutable references aren't really used in specr lang) The question is rather, how do we find out when not to clone. The problems were:

- we don't want to clone a mutable reference
- we don't want to clone large datastructures for performance reasons

The second problem can be rather easily solved by using immutable datastructures for collections; or by wrapping large types in an `Rc`, a reference counted smart pointer. The first problem on the other hand can't be solved that easily. There are multiple approaches and problems with this approach. To see an example on why it might be hard, consider the following code:

```
  foo.bar()
```

converting this to, say

```
foo.clone().bar()
```

breaks if `bar` gets `&mut self` as receiver. But, this cloning might be necessary if `bar` gets `self` instead.
Yet, there might be multiple types having a method named `bar` Thus only if we know the type of `a` we can truly figure out whether we are allowed to clone here or not. Recall that this transpiler works on a syntactical level, hence type-dependent decisions are off the charts.

Make every type Copy

We chose to use Copy. Making every user-facing type Copy has a range of restrictive consequences. First, one should note that Copy types cannot be Drop[2], which means our collections cannot simply free their data upon being dropped. A second consequence is that having a manual Clone implementation is futile, as it will be bypassed when *copied*. This already prevents doing reference counting, as we need a Clone implementation that cannot be bypassed - in order to keep track of the reference count.
The solve all of those problems we implemented a smart pointer GcCow<T> that implements Copy even if T does not. This allows us to use non-Copy types - for example collection types - under the hood, while still providing a Copy-only API.
In the next section, we will talk about how we implemented GcCow<T> by using garbage collection.

5.2 Gc

In the last section we have already motivated our need for a Copy smart pointer - namely GcCow<T> - in order to wrap non-Copy types. This smart pointer will allow to implement Copy collection types like List<T>, and it will help us later on when resolving enum indirection. To implement such a smart pointer, we chose to use garbage collection.
In search of a feasible garbage collector, we looked through crates.io. The problem is that the promising garbage collection libraries - like rust-gc - do not implement Copy for their smart pointers. Hence, we resorted to writing our own very minimalistic garbage collector within libspecr.
Our garbage collector has three central components:

- GC_STATE: A singleton object storing all garbage collected objects.
- GcCow<T>: Our "clone-on-write" Copy smart pointer.
- GcCompat: A trait expressing that a type is compatible with our garbage collector.

Note that the programmer using specr lang, does not need to interact with the GC_STATE, a GcCow or the GcCompat trait. These concepts are implemented in a hidden[3] module in libspecr, and are only used in libspecrs internal data structures and the generated Rust code.

[2]Implementing the Drop trait allows you to specify a *destructor*, a function that is called upon destruction of the object.

[3]*hidden* does not mean private, but excluded from the public API visible in the documentation. These items are required to be public so that the generated code can use them.

5.2.1 The Gc state

The `GC_STATE` stores the garbage collected objects as `Box<dyn GcCompat>`. [4] This allows us to store values of different types in one common type at the cost of an additional indirection. Now, the `GC_STATE` cannot simply store all objects as a `Vec`, as removing an object would change the indices of all successive objects. Hence it uses a `Vec<Option<_>>` to store all objects, and freeing an object simply sets it to `None`.

5.2.2 The smart pointer `GcCow<T>`

As an object will only have a unique index in the `GC_STATE` during its lifetime, we can use that index to refer to objects. This leads us to the definition of `GcCow<T>`:[5]

```
struct GcCow<T: GcCompat> {
    index: usize // index into the GC_STATE.
}
```

As `GcCow<T>` simply consists of a `usize`, it can implement the `Copy` trait. And additionally, copying a `GcCow<T>` is pleasantly cheap!

Now, the only way to obtain a `GcCow<T>` is by calling `GcCow::new(t)` for some `t: T`. This function, will store `t` in the `GC_STATE` and return a `GcCow` with the index of `t`. This guarantees that a `GcCow<T>` will always store the index to a `T`.

To get the inner value of a `GcCow<T>`, we can call the method `extract()`. It will look up the index in the `GC_STATE`, and then use the `Any` trait to downcast the `dyn GcCompat` back to `T`.
The type `GcCow<T>` has an important restriction, which is that the allocated object in the `GC_STATE` is immutable. Instead, if you intend to imitate mutation, you need to create a new `GcCow`. This is a deliberate design decision for the GC. Allowing mutation of objects inside of the `GC_STATE` can easily cause accidental effects: If you give someone a `Copy` of your `GcCow<T>`, they might mutate your object, as a `GcCow` only stores an index.

5.2.3 Cleaning up

The heart of our garbage collector is the mark and sweep algorithm. It will determine which objects are still reachable from a given `root` object, and it will then free all unreachable objects. The mark and sweep algorithm can be invoked by calling the function `mark_and_sweep<T: GcCompat>(root: T)`.

[4] This is called *dyn trait* in Rust. It allows you to use traits to perform dynamic dispatch.
[5] Additionally, `GcCow<T>` stores a `PhantomData<T>` to prevent the "parameter T is never used" compiler error.

As a full mark_and_sweep run takes a notable amount of time, the algorithm will only start if enough memory has been allocated since its last invocation.
Now, in order to determine what objects are still reachable from root, we only need to traverse the "points to"-graph. In this graph, every garbage collected object represents a node, and whenever an object contains a GcCow pointing to another object, this is represented as an edge in the graph.
This leads us to the central method points_to of the GcCompat trait.

```
pub trait GcCompat {
    fn points_to(&self, buffer: &mut HashSet<usize>);

    ...
}
```

Calling x.points_to(buffer) will find all objects that x points to, and insert their GC_STATE indices into buffer. How we implement the GcCompat trait for all relevant types will be explained in the next section.

5.3 Requirements for user-facing types

Only types implementing GcCompat can be used for garbage collection. And as we want to use garbage collection for all types that the user has access to, it is mandatory that we implement GcCompat for all user-facing types, including the types that the users define themselves.
Further, we have already established that all user-facing types should implement Copy so that the user does not have to think about move semantics and cloning.
In this section, we will explain how we made every user-facing type Copy and GcCompat. In order to implement these traits, we present different solutions for types defined by specr lang and *user-defined types*.

Additionally, we need to insert the generic bound T: Copy + GcCompat for all generic types T we encounter.[6] For example the specr lang the code ...

```
struct X<T>(T);
fn foo<T>() { ... }
```

... compiles to the following Rust code:

```
struct X<T: Copy + libspecr::hidden::GcCompat>(T);
fn foo<T: Copy + libspecr::hidden::GcCompat>() { ... }
```

5.3.1 Types defined by specr lang

As specr lang provides quite a few types, we will not present an exhaustive list of implementations. Instead, we will briefly showcase a few types defined in

[6]This Rust syntax is expressing that T has to implement both Copy and GcCompat.

libspecr, and discuss how they implement both GcCompat and Copy.

The most central implementation of GcCompat is the GcCow<T>: GcCompat implementation:

```
impl<T: GcCompat> GcCompat for GcCow<T> {
    fn points_to(&self, buffer: &mut HashSet<usize>) {
        buffer.insert(self.index);
    }
}
```

This effectively states that a GcCow<T> only points to a single object - *its object*. As users never access GcCow<T> directly, this implementation is only used by other types that internally contain a GcCow<T>.

Also, one can simply implement Copy for GcCow<T> as it effectively only stores a usize. All further types we will consider only store Copy primitive types and GcCow in order to remain Copy.

List<T>

As we want our List<T> type to be Copy, we will need to wrap a collection type in a GcCow. We decided against implementing our List<T> as a GcCow<Vec<T>>: As our garbage collector only supports immutable data, a tiny mutation of a Vec would require a new reallocation of the Vec. This is why our List<T> type is instead backed by a RRB-Vector (see [**rbb-vector**]) optimized for this use case. We use the implementation provided by the im crate:

```
struct List<T: GcCompat + Copy> {
    vec: GcCow<im::Vector<T>>
}
```

The GcCompat implementation of List<T> simply calls points_to on the internal vec. This adds the index referring to the im::Vector<T> to the buffer. Now, to complete the picture we need to also implement GcCompat for im::Vector<T>. The points_to method of im::Vector<T> needs to locate all garbage collected objects, that are pointed to by its elements. Hence it calls points_to for each element individually:

```
impl<T: GcCompat + Clone> GcCompat for im::Vector<T> {
    fn points_to(&self, buffer: &mut HashSet<usize>) {
        for t in self.iter() {
            t.points_to(buffer);
        }
    }
}
```

Int

libspecr provides the `Int` type, a mathematical (i.e. unbounded) integer. Internally `Int` contains a `BigInt` defined by the `num-bigint` crate ([Num]). We cannot *directly* use `BigInt` as it does not implement the `Copy` trait. Hence we use a `GcCow<BigInt>` instead.

Yet, there are two disadvantages of this solution:

- *You cannot allocate a `BigInt` in a const context*:
  For example `const BEST_NUMBER: Int = Int::from(42);` would not be possible. Yet definitions like these are common in the MiniRust code base and thus need to be supported.

- *Every integer operation causes a new `BigInt` allocation in the `GC_STATE`*:
  This causes an unreasonable blow up of the `GC_STATE` and loss of performance in arithmetically heavy sections of the code.

To solve both of these problems, `Int` provides both a "Big" variant to allow for unbounded integers, and a "Small" variant for numbers that fit on the stack.

```
enum Int {
    Big(GcCow<BigInt>),
    Small(i128),
}
```

The `GcCompat` implementation of `Int`, will call `points_to` on the `GcCow<BigInt>` in case of `Int::Big`, and otherwise will not add anything to `buffer`.

5.3.2 User-defined types

Now, let us come to the question: How do we implement `Copy` and `GcCompat` for types that the programmers using specr lang create themselves? The first part of this problem has a simple solution: `Copy` can be implemented by adding an attribute, given that your type only consists of other `Copy` types.
In practice this looks as follows:

```
#[derive(Copy, Clone)] // This attribute was inserted by specr-transpile
struct MyUserDefinedType {
    field1: CopyType1,
    field1: CopyType2,
}
```

[7]

Let us now consider the harder question: How do we implement `GcCompat` for user-defined types? In order to solve this problem, we have implemented a

[7]We need to implement `Clone` aswell, as `Copy` can only be implemented for types that already implement `Clone`.

derive macro (see the `gccompat-derive` crate) that automatically implements the `GcCompat` trait for any new type. What this derive macro will do is, it will traverse through the data structure, until it reaches a type for which `libspecr` already provided a `GcCompat` implementation. To make the transformations of this derive macro graphical, we present a few examples:

```rust
struct UserDefinedType {
    field1: u32,
    field2: List<SomeOtherType>,
    field3: OtherUserDefinedType,
}

// generated using gccompat-derive:
impl GcCompat for UserDefinedType {
    fn points_to(&self, buffer: &mut HashSet<usize>) {
        self.field1.points_to(buffer);
        self.field2.points_to(buffer);
        self.field3.points_to(buffer);
    }
}
```

For structs, the proc macro iterates over the fields of our struct, and checks what the fields individually point to.

In this particular example, the first field is an `u32`. The `points_to` implementation of `u32` does nothing - as `u32` points to nothing. Hence nothing gets added to the `buffer`.
But the second field stores a `List<T>`. As we have explained previously, a `List<T>` internally stores a `GcCow` that manages the actual data. The `points_to` implementation of `List<T>` will now add the index stored within the `GcCow` to the `buffer`.
Finally, for the third field, we encounter another user-defined type. So here, the traversal continues and we recursively call `points_to` on our `OtherUserDefinedType`.

Let us now also look at how `gccompat-derive` handles enums:

```rust
enum OneOrTwo<T> {
    One(T),
    Two { t1: T, t2: T },
}

// generated using gccompat-derive:
impl<T: GcCompat> GcCompat for OneOrTwo<T> {
    fn points_to(&self, buffer: &mut HashSet<usize>) {
        match self {
            OneOrTwo::One(t) => {
                t.points_to(buffer);
            }
            OneOrTwo::Two { t1, t2 } => {
                t1.points_to(buffer);
                t2.points_to(buffer);
            }
        }
    }
}
```

For enums, gccompat-derive generates a match block, whereas every variant of the enum is handled similarly to how a struct is handled. *side note*: This example also showcases how gccompat-derive handles generics!

Limitations

Our goal is to implement `Copy` and `GcCompat` for every user-facing type, but there are two elusive exceptions:

- Closures do not implement `GcCompat`, and
- Mutable references do not implement `Copy`

This does not mean that the use of either of them is forbidden in specr lang; in fact both of them are frequently used in the MiniRust codebase. But it has the implication, that they are excluded from garbage collection and hence can not be part of type definitions. For example this type definition will fail to compile:

```
struct Foo(&'static mut i32);
```

This is due to the fact, that Rust is not able to implement `Copy` for this type.

5.4 Enum indirection

This section explains how enum indirection (the attribute `#[specr::indirection]` is resolved in the specr lang compiler. First-of all, the indirection chosen for a `#[specr::indirection]` is `GcCow<_>`. This is the only option, as we need some form of indirection that implements `Copy`, thus all standard options like `Box`, `Rc` are not acceptable.
We will demonstrate the transformations required to implement enum indirection using an example:

```
enum Nat {
    Zero,
    Succ(#[specr::indirection] Nat),
}

fn succ(n: Nat) -> Nat {
    Nat::Succ(n)
}

fn pred(n: nat) -> Nat {
    match n {
        Nat::Zero => Nat::Zero,
        Nat::Succ(m) => m,
    }
}
```

compiles to

```
enum Nat {
    Zero,
    Succ(libspecr::hidden::GcCow<Nat>),
}

fn succ(n: Nat) -> Nat {
    Nat::Succ(libspecr::hidden::GcCow::new(n))
}

fn pred(n: nat) -> Nat {
    match n {
        Nat::Zero => Nat::Zero,
        Nat::Succ(m) => {
            let m = m.extract();
            m
        },
    }
}
```

The compiler does three things:

1. look for *#[specr::indirection]* and wrap the respective type within a `GcCow<_>`
2. Whenever we encounter a construction of the variant where we added an indirection before, we insert a `GcCow::new(_)` around the respective expression
3. Whenever we encounter a match of the variant where we added an indirection, we prepend the match block with `let m = m.extract();` This will shadow the variable `m: GcCow<T>` with the extracted variable `m: T` instead. [8]

One subtle yet important detail, is that this handling of `match` blocks needs to be done after all `argmatch`es were already resolved. This is due to the fact, that

[8]Declaring a new variable with the same name as an old variable will *shadow* that old variable, making it effectively unreachable until the new variable goes out of scope.

every argmatch results in another match block, so doing these transformations in the wrong order will make it miss some match blocks.

Limitations

Not all enum variants are used in a *fully-qualified* way. Consider this example:

```
enum Nat {
    Zero,
    Succ(#[specr::indirection] Nat),
}

use Nat::*;

fn one() -> Nat {
    Succ(Zero)
}
```

In this example, the variants `Succ` and `Zero` are used without prefixing them by `Nat`. As `Succ` and `Zero` are only defined as `Nat`, the algorithm is able to figure this out though. The problem arises, if there are multiple enum variants sharing the same name. If we extend our previous code by this declaration:

```
enum Foo {
    Succ(i32),
    Bar(u32),
}
```

we will encounter a naming collision for the variant `Succ`. To make this concrete, if we write something like this

```
let x = Succ(y);
```

specr-transpile cannot know whether it should wrap `y` in a `GcCow::new(_)` or not. If `Nat::Succ` is in scope a `GcCow::new` is necessary - but if `Foo::Succ` is in scope, a `GcCow::new` is not allowed. This will result in a compile-time error.

In the long run, it is probably advisable to implement full name resolution to mitigate this problem. We presume that this will be necessary for the specr lang to Coq compiler either way.

Chapter 6

Advances in the MiniRust specification

MiniRust puts a strong focus on precisely specifiying the semantics of *unsafe* Rust. Hence much effort went into figuring out the correct semantics of unsafe operations and the handling of memory. But as MiniRust is a rather small language, it is missing certain features that prevents it from representing a large set of Rust programs. In this Chapter we present contributions to the MiniRust specification that extend MiniRust to make it more expressive.

6.1 Intrinsics

The original set of MiniRust features was somewhat restricted. For example, MiniRust programs could not have any side-effects, which made it more complicated to observe whether the programs does what you expect it to do. Or to name another example, every memory allocation MiniRust were function locals, so there were no *dynamic memory allocations*.
To solve these problems we introduced *intrinsics*. Intrinsics are *built-in procedures* that a MiniRust program can invoke. They distinguish themselves to functions, as they won't be put on the stack-frame when called; and they are untyped, meaning that calling a intrinsic with the wrong type of argument will result in a runtime error.
Just like for functions, calling an intrisic is a terminator, i.e. the last instruction of a basic block.
We will now introduce the different intrinsics that we have added:

Exit

Previously, MiniRust programs had no way to gracefully terminate their execution. They either raised undefined behavior at some point, or ran in an infinite loop. This is why we introduced the `exit` intrinsic.

Alongside to the introduction of the `exit` intrinsic, we introduced *never-returning* functions. Previously, each MiniRust function was forced to provide a return local even though it did not intend to return. Let us see an example:

```
start fn never_returning() -/> {
  start bb0:
    _ = exit();
}
```

This function `never_returning` does not have a return local as expressed by the strikethrough arrow `-/>`. It does not need a return local, as it makes use of the `exit` intrinsic instead of returning.
Never-returning functions are used for `start` functions, and might play a role in expressing Rusts `!`-returning functions.

If a never-returning function actually *returns*, it will raise undefined behavior:

```
start fn bad_function() -/> {
  start bb0:
    return;
}
```

Dynamic memory

We added the intrinsics `allocate` and `deallocate`, which allow you to allocate pieces of memory and free them again. The `allocate` intrinsic will return a pointer to a freshly allocated piece of memory. It receives two integer arguments, the *size* of the allocation, and its *alignment*. Additionally, the `deallocate` intrinsic accepts the pointer as an argument, plus the *size* and *alignment*. The `deallocate` intrinsic will raise undefined behavior, if

- there is no live allocation starting at the given pointer
- the size or alignment do not match the allocation

Let us see a simple example showcasing dynamic memory:

```
start fn f0() -/> {
  let _0: (*layout(size=4, align=4))@align(8);
  start bb0:
    storage_live(_0);
    _0 = allocate(4, 4) -> bb1;
  bb1:
    _ = deallocate(load(_0), 4, 4) -> bb2;
  bb2:
    _ = exit();
}
```

This simple example, allocates memory and directly frees it again.

Printing

Previously, when running a MiniRust program the only feedback one receives is whether

- the program terminated normally, or
- what kind of undefined behavior was raised

This makes it very hard to understand whether the execution of the program happened as intended, in particular for programs that did not raise undefined behavior. In order to tackle this issue, we introduced the `print` and `eprint` intrinsics. They allow you to print a value to stdout or stderr respectively. Calling these intrinsics will not change the state of the `Machine`, but only generate the side-effect of printing a value. As intrinsics are untyped, `print` and `eprint` are able to print values of different types. We utilize these intrinsics to write tests, checking whether MiniRust behaves as intended.

```
start fn f0() -/> {
  start bb0:
    _ = print(42) -> bb1;
  bb1:
   _ = exit();
}
```

6.2 Globals

The Rust programming language provides certain features that require memory allocations to be live during the *whole execution* of the program. A typical example of this are Rusts *statics*.

```
static STATIC_FOURTY_TWO: u32 = 42;

fn main() {
    let number: u32 = STATIC_FOURTY_TWO;
}
```

This `static` declaration will introduce a static memory allocation storing the number 42. Whenever `STATIC_FOURTY_TWO` is referenced, the program will implicitly dereference a pointer into that static allocation.
Currently, MiniRust has no capabilities to express those static allocations. In order to solve this problem, we extended both the MiniRust syntax and the interpreter by *globals*.
A global is a static memory allocation, that we allocate in the `Machine::new` function, right before the program starts being executed. Using globals we can derive the following MiniRust program to represent our Rust program from above.

```
global(0) {
  bytes = [2a 00 00 00],
  align = 4 bytes,
}

start fn f0() -/> {
  let _0: u32@align(4);
  start bb0:
    storage_live(_0);
    _0 = load(deref<u32@align(4)>(global(0)));
    _ = exit();
}
```

As you can see there are quite a few differences between the Rust and MiniRust versions. First of all you might notice that globals contain *raw memory*. `global(0)` does not contain the number 42, but instead its encoded representation. [1] Further, the `global(0)` expresses that it is required to be allocated at alignment 4.[2] Additionally, you can see that MiniRust is quite explicit about globals being pointers. This is why `global(0)` will be `deref`erenced and loaded.

Now, this is not the complete picture of globals. There are Rust constructions that are simply not expressible by this.
Let us provide an example:

```
static X: u32 = 42;
static Y: &'static u32 = &X;
```

In this example, we again have our statically allocated 42, but additionally we have a new `static` that stores a reference to `X`. As a reference is just a kind of pointer, the global associated with `Y` simply needs to store the address of `X`. But as `X` will be allocated at program startup, its address has not been determined yet!

Hence, in order for `Y` to point to `X` we need a new feature, namely a *relocation*. In MiniRust this looks as follows:

```
global(1) {
  bytes = [00 00 00 00 00 00 00 00],
  align = 8 bytes,
  at byte 0: global(0), // relocation to `X` aka `global(0)`
}
```

This new global represents the `static Y`. You might notice that its `bytes` are left blank, and instead we have a ‘`at byte 0: global(0)`‘ annotation. This is a

[1] In order to make the examples in this section concrete, we consider a 64-bit little endian system.
[2] This follows from the fact that u32 requires an alignment of 4.

relocation. It expresses that the at the beginning of the allocation (`at byte 0`), we have to insert a pointer to `global(0)`.

`Machine::new` can now simply allocate all globals first; and then - given that all addresses are now determined - resolve all relocations.

So far we have talked about globals that store only integers - or only pointers. To fully grasp the power of globals, we have to consider one final example:

```
static X: (u32, u32) = (42, 42);
static Y: (u64, &'static u32) = (42, &X.1);
```

In this example, both statics are tuples. And by the design of this example, the resulting relocation pointing from `Y.1` to `X.1` can only be described by using *two offsets*:

- the offset describing which byte in `X` to address: This offset is 4 bytes, as this is the width of the preceding `u32`.[3]
- the offset describing where to put the pointer *within* `Y`: This offset is 8 bytes, as this is the width of the preceding `u64`.

This results to the following globals.

```
global(0) { // represents `X`
  bytes = [2a 00 00 00 2a 00 00 00],
  align = 4 bytes,
}

global(1) { // represents `Y`
  bytes = [2a 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00],
  align = 8 bytes,
  at byte 8: global(1) + 4,
}
```

The relocation `at byte 8: global(1) + 4` expresses both offsets.
Further, note that some of the bytes that make up `global(1)` are unchanged by the relocation, and only the second half of bytes is replaced by the pointer to `global(0)`.

6.3 Further Progress in MiniRust

In addition to the larger changes we have already covered, we also made a set of smaller changes to MiniRust over the period of working on this Master's Thesis. These contributions are not central to this thesis, but as they add up to a notable improvement for executing MiniRust, we decided to at least mention a few of them.

[3]These examples are designed to not have additional padding.

Operators

While trying to test control flow using `if` in MiniRust, we noticed that there is no practical method to generate meaningful booleans. The only options were using `true`, `false`, or transmuting an integer to a bool, using a pointer or a union.
This made it very hard to implement meaningful tests for MiniRusts control flow. In order to address this problem, we introduced Rusts `<=`, `>=`, `<`, `>`, `==`, `!=` operators for integer types. And while doing that we also quickly added

- the missing *remainder* operator for integers, and
- a pointer to pointer cast, allowing us to freely change pointer types from one to the other.

Machine initialization

As you might recall from the MiniRust introduction, a `Machine` stores the full state of a program at a given point in its execution. Previously, only the `step` function of `Machine` existed, but there was no way to construct a `Machine`. This is why we implemented `Machine::new(program: Program)`: First, it checks whether the program is *well-formed* and rejects the program if not. And then it sets up the initial stack frame and allocates the globals.

Use-after-free

When an allocation is freed in MiniRust, a flag in the allocation is set to indicate that. When accessing memory, this was overlooked and hence one was able to access already-freed memory. We quickly fixed that, and now it raises a `use-after-free` undefined behavior.

Negative Count Arrays

While proving the generic properties about the encoding, we noticed that MiniRusts arrays could have a negative number of elements, which would quickly cause problems.

Chapter 7

MiniRust Tooling

In Chapter 5 we implemented a specr to Rust compiler. Using this compiler, we can finally compile the MiniRust specification to a Rust crate. Next to simply executing MiniRust programs, this also allows us to implement tools in Rust that can build upon this MiniRust crate. In this Chapter we present the following tools:

- *minimize - a Rust to MiniRust compiler*: The whole idea of MiniRust is to define Rusts semantics in terms of MiniRusts semantics by compiling Rust to MiniRust while inheriting its semantics. Hence compiling Rust to MiniRust is a crucial step in this process. In combination with the MiniRust interpreter, this tool provides an alternative path to execute Rust code. This can be used to identify inconsistencies between Rust and MiniRust.
- *a MiniRust pretty printer*: MiniRust itself does not provide a syntax - it only provides an abstract syntax, i.e. a data structure representing the syntax. This makes working with MiniRust programs very hard, as you simply cannot read a program. For example, when generating a MiniRust program with `minimize`, we cannot simply *look* at it in order to determine whether the translation was correct.

  For those reasons, we developed a work-in-progress syntax for MiniRust, and implemented a pretty printer for it.

  Note that every piece of "MiniRust code" that we have shown in this Master's thesis was generated using this pretty printer!
- *MiniRust Testing*: Our test suites incorporate a collection of MiniRust programs. They assert that executing them using MiniRusts interpreter yields the desired results. We test that MiniRust is able to detect certain kinds of undefined behavior, and additionally that MiniRust programs without undefined behavior behave as expected.

Before we start diving into the presented tools, we have to answer one more question.

Running MiniRust programs - how to collect garbage?

The MiniRust specification is written as an interpreter. In order to run this interpreter, we first need to create a `machine` using `Machine::new(program)`, and then we can repeatedly call `machine.step()` to run the program step by step.

You might recall from Chapter 5 that we have introduced a garbage collector, yet we have never mentioned how and when we free leftover allocations by calling `mark_and_sweep`!

As the machine is intended to fully capture the state of a program during execution, we can use it as the `root` for our garbage collector. So, between the calls to `machine.step()` we free every garbage collected object that cannot be reached from the machine anymore, by calling `mark_and_sweep(machine)`.

7.1 Minimize

As alluded to before, Minimize is a Rust to MiniRust compiler.
Implementing such a compiler is generally a rather extensive task due to the fact that Rust is a very complex language. In order to compile Rust down to MiniRust, we need to rid Rust of most of its high-level features: Pattern matching, generics, traits, modules etc. are part of Rust, but are not part of MiniRust. Implementing such a compiler involves reimplementing a significant portion of the Rust compiler. So instead, we can actually use the Rust compiler to do the hard work for us. This is possible, as we can ask the Rust compiler to output an intermediate format, namely `MIR` - the mid-level intermediate representation. MiniRust itself is an idealized version of MIR. So translating MIR to MiniRust is a much more viable approach. It is worth emphasizing that MIR is an *intermediate format within the Rust compiler.* Implementing this compiler required us to take a deep dive into the intricacies of the Rust compiler. A notable amount of work on minimize consisted of pondering the internal API of rustc.
In order to obtain MIR, we made use of the component `rustc-dev`. It allows us to compile a given Rust program, and provides us access to the intermediate results of the Rust compiler.

As MiniRust and MIR are relatively similar languages, a detailed explanation of the whole translation procedure is rather redundant. Instead, we will highlight a few key differences between MiniRust and MIR, and how minimize handles them.

Generics

MIR does support generics, while MiniRust does not. This means that in order to translate MIR to MiniRust, we need to resolve generics. This process is called *monomorphization*. It works by generating a new function for every generic invocation of a generic function.

As we don't want to bother the reader by explaining yet another language - MIR, we will use a Rust example in order to explain how monomorphization works.

```
fn identity<T>(t: T) -> T { t }

fn main() {
    let x: u32 = identity::<u32>(42);
    let y: i32 = identity::<i32>(42);
}
```

... will be translated to ...

```
fn identity_u32(t: u32) -> u32 { t }
fn identity_i32(t: i32) -> i32 { t }

fn main() {
    let x: u32 = identity_u32(42);
    let y: i32 = identity_i32(42);
}
```

This translation is rather straight-forward. Yet, not in all scenarios it is this simple to infer which generic invocations a given function has. The problem becomes harder if generic functions start calling other generic functions, maybe even in a circular way.

```rust
fn main() {
    foo1::<u32>();
}

fn foo1<T>() {
    if some_condition() {
        foo2::<T>();
    } else {
        foo2::<i64>();
    }
}

fn foo2<T>() { foo1::<T>(); }
```

Figure 7.1: A more challenging example for monomorphization.

The key observation to solve this problem is that every function invocation with fully specified arguments can only call other generic functions with fully specificed arguments. This means if we start with the main function, which has no generic arguments and is hence fully specified, we can see which functions get called from there and so on. In the concrete example given above, we can do the following:

- Compile `main`.
  Add `foo1::<u32>` to the todo list.
- Compile `foo1::<u32>`.
  Add `foo2::<u32>` and `foo2::<i64>` to the todo list.
- Compile `foo2::<u32>`.
  Nothing to add, `foo1::<u32>` was already added!
- Compile `foo2::<i64>`.
  Add `foo1::<i64>` to the todo list.
- Compile `foo1::<i64>`.
  Nothing to add, `foo2::<i64>` was already added!
- The todo list is empty!

This algorithm is the central piece of minimize. It decides which functions get translated at all. That minimize uses this algorithm also implies that functions which do not get called, will never be compiled at all.

Start function

A MIR program starts with its `main` function, typically returning the unit type `()`. In MiniRust the starting function is required to have no return local, so returning () is not an option. Further, in contrast to MIR, a MiniRust program requires an explicit call to `exit()` to terminate the execution.

Instead of compiling MIRs `main` function in a special way to address these differences, minimize inserts a wrapper function that simply calls `main` and then terminates using `exit`.

```
// This is the translated `main` function.
fn f1() -> _0 {
  ...
}

// minimizes wrapper function
start fn f0() -/> {
  start bb0:
    _ = f1() -> bb1;
  bb1:
    _ = exit();
}
```

Intrinsics

In Chapter 6 we introduced *intrinsics* to MiniRust. Now, it would be useful if Rust programs that were given to minimize are able to call these intrinsics. Otherwise, minimize is unable to represent a signification portion of MiniRust programs.

Recall that minimize starts by giving the Rust program into the Rust compiler. So, if we try calling an intrinsic in our Rust program, the Rust compiler will reject it with "cannot find function ‘print‘".
In order to sidestep the Rust compiler, we implemented an external crate named `intrinsics`. This crate provides a function for each intrinsic. Now, by calling the functions of that crate the Rust compiler will be pleased. And after the Rust compiler has finished generating MIR, minimize will translate each call to a function of the `intrinsics` crate to the call of MiniRusts actual intrinsic instead.
This allows us to compile the following program using minimize:

```
extern crate intrinsics;
use intrinsics::{print, allocate, deallocate, exit};

fn main() { unsafe {
    let raw_ptr: *mut u8 = allocate(4, 4);
    let int_ptr: *mut u32 = raw_ptr as *mut u32;

    *int_ptr = 42;
    print(*int_ptr); // prints 42
    deallocate(raw_ptr, 4, 4);
    exit();
}}
```

7.2 Pretty printer

MiniRust only defines an abstract syntax which humans cannot read directly. In particular when writing `minimize`, this was pretty problematic, as I needed to read non-trivial MiniRust programs to see if the translation was done correctly. This motivated us to implement a simple pretty printer for MiniRust programs.

The implementation of the pretty printer works as follows: For every syntactical concept that MiniRusts abstract syntax provides there is a respective function that converts it into a String. This includes `Program`, `Function`, `BasicBlock`, `Statement`, `Terminator`, `ValueExpr`, `PlaceExpr`, `Type` and many more. These functions will recursively call each other, thus recursing through the syntax tree of the program. For example `fmt_program` will call `fmt_function` for every function; then `fmt_function` will call `fmt_basic_block` for all basic blocks etc.

It is worth noting that all the MiniRust code snippets that are contained in this document were generated by this pretty printer.

composite types

This pretty printer has one feature that is worth highlighting, which is the way that composite types (unions and tuples) are handled. In the abstract syntax of MiniRust, types are not declared and then used - but instead the full type definition needs to be written out whenever a given type is used. This is unnecessarily verbose, if you use nested types like tuples and unions often in your code.
So, what this pretty printer does is, whenever it encounters a new composite type, it associates it with a new name (`T0`, `T1`, `T2`, ...) and prints this name instead of the type definition. And after pretty printing the whole program, it will insert declarations for all composite types it encountered during the program. Let us see this in action:

```
tuple T1 (12 bytes) {
  at byte 0: i32,
  at byte 4: i32,
  at byte 8: i32,
}

tuple T0 (24 bytes) {
  at byte 0: T1,
  at byte 12: T1,
}

fn f0(_1) -> _0 {
  let _0: T0@align(4);
  let _1: T0@align(4);
  start bb0:
    _0 = move(_1);
    return;
}
```

In this example, the pretty printer will first read `f0`, while reading that it will encounter a new tuple type it will name `T0`. And then, while generating the type definition for `T0` it will encounter another tuple type it will call `T1`. And finally `T1` does not contain any more new types.

7.3 MiniRust Testing

As part of this thesis, we implemented two separate test suites.

- *minitest*: This "pure MiniRust" test suite contains hand-written MiniRust programs. It executes them using MiniRusts interpreter and compares their outcome with the expected outcome.
- *The minimize test suite*: This test suite consists of a collection of Rust programs. It works by using `minimize` to convert the Rust programs to MiniRust, and only then invoking MiniRusts interpreter.

Both test suites are equipped with a shell script named `cov.sh` that calculates the test coverage across the MiniRust codebase, and in case of the minimize test suite, also across the minimize codebase. This script allows us to detect blind spots of our test suite, and in particular cases of undefined behavior we have missed. The coverage is represented over a line-by-line basis, and expresses the number of times a particular piece of code was executed:

```
   | impl Type {
567|     pub fn size<M: Memory + libspecr::hidden::Obj>(self) -> Size {
567|         use Type::*;
567|         match self {
223|             Int(int_type) => int_type.size,
 16|             Bool => Size::from_bytes_const(1),
158|             Ptr(_) => M::PTR_SIZE,
145|             Tuple { size, .. } | Union { size, .. } | Enum { size, .. } => size,
                         ^107                 ^38                  ^0
 25|             Array { elem, count } => {
 25|                 let elem = elem.extract();
 25|                 elem.size::<M>() * count
   |             }
   |         }
567|     }
   | }
```

Figure 7.2: Sample output of `cov.sh`. This example shows that the `Type::size` function was evaluated 567 times during the execution of `minitest`. 223 of those were ints, and for example 107 of those were tuples.

This script uses LLVMs coverage tools.

minitest

This test suite runs MiniRust programs and checks that the interpreter yielded the expected outcome. It defines three functions

- `assert_ub(program: Program, ub_message: &str)`
- `assert_stop(program: Program)` and
- `assert_ill_formed(program: Program)`

These functions run a program using MiniRusts interpreter, and then check whether it yielded the desired outcome.

A typical minitest case looks as follows:

```rust
#[test]
fn null_ptr_example() {
    let program = make_null_ptr_program();
    assert_ub(program, "dereferencing null pointer");
}
```

As minitest is executing the MiniRust programs in a thread instead of spawning a new process, it cannot detect any `print`s that its tests generate. [1] Thus

[1] We might address this in the future by making the printing destinations configurable in MiniRust.

minitest cannot assert much for a gracefully terminating program, except for the fact that it indeed terminated. Hence minitest is mostly used to check whether certain programs have undefined behavior or not; and in particular which kind of undefined behavior. minitest covers all 35 cases of undefined behavior that MiniRust currently supports. Further, minitest is invoked in MiniRusts CI. So any change to MiniRusts specification has to first pass minitest.

Now, the reader might be wondering how we construct MiniRust programs to use for our test cases: MiniRust does not have a parser. In fact, except for the work-in-progress output format of our pretty printer, MiniRust does not even have a syntax. It only provides an *abstract syntax* - a data structure for our abstract syntax tree.
As constructing programs using only the abstract syntax quickly becomes infeasible, we built an abstraction - a program builder API - that allows us to construct MiniRust programs using less work.

We want to showcase this abstraction by an example. The following MiniRust program ...

```
start fn f0() -/> {
    let _0: u32@align(4);
    start bb0:
        storage_live(_0);
        _0 = 42;
        _ = exit();
}
```

... can be constructed using our program builder API as follows:

```
// We only have a single local `_0`.
// It uses the place type of u32, namely `u32@align(4)`.
let locals = [<u32>::get_ptype()];

// We define the basic block `bb0`.
let bb0 = block!(
    storage_live(0),
    assign(local(0), const_int::<u32>(42)),
    exit()
);

// The function `f0` does not return, has zero arguments, and consists of the given locals
let f0 = function(Ret::No, 0, &locals, &[bb0]);

// The program consists of a single function `f0`.
let p = program(&[f0]);
```

For the skeptical reader, who might not be convinced that such a program

builder is necessary or even helpful, we refer to the same program written using only the abstract syntax: A.1.
We presume that ultimately, when a stable syntax gets developed for MiniRust, this builder API will be superseded by a parser for MiniRust programs.

The minimize test suite

This test suite consists of Rust programs. These programs will be converted to MiniRust programs using minimize; then executed using MiniRusts interpreter; and finally we check whether they yield the desired result. Thus effectively, this test suite tests both `minimize` and the MiniRust interpreter.

A typical minimize test looks as follows:

```
// test.rs:
extern crate intrinsics;
use intrinsics::print;

fn main() {
    let mut x = 1;
    while x < 4 {
        print(x);
        x = x + 1;
    }
}

// test.stdout:
1
2
3
```

Figure 7.3: A typical minimize test

As this test suite executes programs by starting a new process, we can finally make use of the `print` intrinsic. This allows us to do interesting computations, and then verify its result.

Also, having tests being written in plain Rust has a few advantages in itself. First of all, changes to MiniRust can easily break test cases that are written in *MiniRust*, like test cases from `minitest`. These breaking changes also affect minimize, but when minimize is updated we do not need to rewrite its tests! Further, it is generally easier to write a test case for minimize than for minitest, as writing plain Rust is simpler than using the MiniRust program builder API. On the flipside, certain tests cannot be expressed as a minimize test, simply because the MiniRust program in question can not be generated by minimize.

And further, given a non-trivial Rust program, minimize might generate a very large MiniRust output. Now, if such a test fails, it is very hard to determine whether minimize or MiniRust is at fault for this failure.

Generally, we find that using both test suites balances their advantages and disadvantages nicely. Some tests are better expressed by minitest, and some by minimizes test suite.

Chapter 8

Formally verifying MiniRusts Value encoding

In section 3.6 we briefly introduced MiniRusts encoding system. To summarize, each MiniRust type provides an `encode` and a `decode` function. The `encode` function converts Values to their ”in-memory” representation as a list of AbstractBytes. The `decode` function does the converse. In order for these two functions to harmonize correctly, they must satisfy certain criteria.

To name a simple example: When we encode a value and then decode it back, we want to obtain the original value.
Properties like this are fundamental assumptions of MiniRust. If any of them did not hold it would be considered a specification bug.
Hence in order to guarantee that they indeed hold, we have formalized the encoding of types in Coq, and proved these desired properties for all MiniRust types.
In this Chapter we will first formally define the properties and then give some insights into our Coq implementation at `https://github.com/coq-minirust`.
But before defining what the properties express, we require to define a few basics.

8.1 Definitions

We will shortly introduce everything required to know to understand the desired properties of the encoding.

8.1.1 AbstractByte

MiniRusts memory does not directly store bytes (or `u8` in Rusts terms), but instead it needs to keep track of a few more details. To capture these pieces of extra information MiniRust defines the `AbstractByte` type.

```rust
enum AbstractByte {
    Uninit,
    Init(u8, Option<Provenance>),
}
```

An AbstractByte has two central differences from a `u8`:

- An AbstractByte might be uninitialized
- In addition to the `u8`, an AbstractByte might store a provenance. This will be relevant for the encoding of pointers, as pointers have to remember their provenance even when stored in memory.

In order to ease providing examples, we will for the rest of this Chapter assume that `Init` and `Uninit` are always in scope. This means we can describe a piece of memory with three bytes as for example `[Uninit, Uninit, Init(42, None)]`.

8.1.2 Types and Values

To make MiniRusts encoding, and also its desired properties, a bit more tangible, we will provide three example MiniRust types and their corresponding Values. This is not strictly necessary to define the properties, but it helps understand them better.
The following definitions are simplified versions of the original MiniRust specification. We refer to types.md for MiniRusts complete type definitions, and to values.md for the Value definition and the encoding.
We define a MiniRust type as follows:

```rust
enum Type {
    Bool,
    Int {
        size: Size,
        // `Signed` or `Unsigned`.
        signedness: Signedness,
    },
    RawPointer,
    Tuple {
        fields: List<(Offset, Type)>,
        size: Size,
    },
    Union {
        fields: List<(Offset, Type)>,
        size: Size,
        chunks: List<(Offset, Size)>,
    }
}
```

A `Type::Int` is specificed by its `Size` - represented as bytes or bits - and its signedness. For example the type `u32` would be represented as

```
Type::Int {
    size: Size::from_bits(32),
    signedness: Unsigned,
}
```

For the sake of simplicity, we have stripped `Type::RawPointer` of its pointee. The pointee does not affect the encoding of a raw pointer.

A tuple stores a sequence of fields. A field consists of a byte offset expressing where to store this field, and the type of the field itself. Additionally, a tuple type stores the full size it occpuies in memory. To present a simple example, the tuple type `(bool, bool)` could be expressed as follows:

```
Type::Tuple {
    fields: list![
        (Offset::from_bytes(0), Type::Bool),
        (Offset::from_bytes(1), Type::Bool)
    ],
    size: Size::from_bytes(2),
}
```

Tuples have a notion of *padding*. Padding are unused bytes, that appear if the total size of the tuple is larger than the sum of its fields. Padding is occasionally required to satisfy alignment constraints. For example the type `(u8, u16)` will insert a padding byte after the `u8` in order to place the `u16` at its required alignment of two.
The last type we are considering is `Type::Union`. Just like tuples, they store their fields and their full size. The difference to tuples is that the fields of a union may overlap. Overlapping fields for a tuple are rejected by the well-formedness checker!

In order to define which bytes are padding within a union, the union is decomposed into chunks. Every chunk is described by an offset and a size. Bytes that are within a chunk are considered to be relevant for the union, whereas bytes not contained in any chunk are considered to be padding.

In addition to the types, let us quickly introduce their respective values.

```
enum Value {
    Bool(bool),
    Int(Int),
    RawPointer {
        address: Int,
        provenance: Option<Provenance>,
    }
    Tuple(List<Value>),
    Union(List<List<AbstractByte>>),
}
```

Most of these cases are rather unsurprising.
The only interesting part of this definition is the `Value::Union`. While discussing `Type::Union` we mentioned that a union only preserves the bytes that are within one of its chunks. Now, as a union at runtime does not know which field it contains, it has to simply store all relevant bytes instead. It does so by storing a list of chunks, where each chunk itself is simply a list of AbstractBytes. This results in the type `List<List<AbstractByte>>`.

8.1.3 The Encodings

Now that we have introduced all relevant types and their corresponding values, we can talk about how the encoding of values in memory works in practice.

Recall that each type provides an `encode` function converting a `Value` to its byte-representation as `List<AbstractByte>`, and a `decode` function for the converse. In this subsection we will break down how these functions work for each type. We again refer to values.md for the precise definitions.

Before diving into the encodings we have to quickly address one small problem: Previously we have already explained that encoding of a Value cannot fail, because each Value needs to have at least one representation in memory. But, as MiniRust only provides a single Value definition covering all types at once, we run into problems when encoding a Value with the wrong type. For example, when trying to store an integer in a bool-local, it would invoke bools `encode` function with an integer value which cannot succeed and results in a runtime panic. Type errors like these will be caught by the well-formedness checker, hence we will assume that `encode` is only invoked with values that are *valid* for the given type.
With that out of the way, we can start defining the `encode` and `decode` functions!

Bool encoding

Encoding a boolean is rather straight forward: `encode(Value::Bool(true))` yields `[Init(1, None)]`, whereas `encode(Value::Bool(false))` yields `[Init(0, None)]`. So a boolean uses a single byte to represent itself in memory. This byte is either one or zero to differentiate between `true` or `false`.

Decoding on the other hand has a few more edge cases to handle: Given a list of AbstractBytes of the shape `[Init(x, y)]`, decoding will succeed only if `x=1` or `x=0`, yielding `Value::Bool(true)` and `Value::Bool(false)` respectively. For all other cases, decoding will fail. In particular this includes the cases where

- the list of AbstractBytes has zero or more than one element,

- the AbstractByte is uninitialized,
- the byte is neither 1 or 0

Note that decoding a boolean does not care, whether the `AbstractByte` is equipped with a provenance or not. So decoding `[Init(1, None)]` and `[Init(1, Some(prov))]` both succeed with `Value::Bool(true)`.

Integer encoding

Each integer type explicitly states how many bytes it is encoded in using its `size` field. If an integer does not fit into this space, encoding will raise a panic. It is the job of the MiniRust spec to ensure that integer values always fit into their type. Decoding fails if the number of AbstractBytes is incorrect, or it contains an `Uninit`. But again, decoding ignores the provenance of the given `AbstractByte`s.

The byte order (`LittleEndian` or `BigEndian`) in which an integer shall be encoded has to be specified by the `Memory`. As precisely defining integer encoding is more laborious than it is insightful, we refer to [PH16] for further details.

RawPointer encoding

Pointers are encoded by encoding their address as an unsigned integer, as explained above. The size of this integer is, like the byte order, specified by the `Memory`.
When encoding a pointer, all resulting `AbstractBytes` additionally inherit the provenance stored within our `Value::Pointer`. And conversely, when decoding a pointer, the `Value::Pointer` will adopt the unique provenance specified by all `AbstractBytes`. If the `AbstractBytes` do not agree on a single provenance, the resulting pointer will obtain `None` as a provenance instead.

Tuple encoding

Encoding a tuple works by recursively calling `encode` for all of its fields, given their respective types. The resulting AbstractBytes will be placed according to their byte offsets. Also, the padding - bytes that are not covered by any field - will remain `Uninit`.
Decoding a tuple works by calling `decode` for all fields, and combining the resulting Values into a list. Padding bytes are ignored in this process, they are even allowed to be `Uninit`.

Union encoding

Recall that unions consist of multiple chunks of data. The bytes between two consecutive chunks are considered to be padding.

As the value representation of a union already stores chunks of `AbstractBytes`, encoding them into a single list of `AbstractBytes` is a rather direct transformation.
When encoding a union value, we store all chunks at the byte offsets specified in the `Type::Union`. All padding bytes are set to `Uninit`. Conversely, decoding a union works by extracting the `AbstractBytes` within each chunk, and collecting them into a `List`. Similarly to the decoding of tuples, all padding bytes are ignored!
Note that the `fields` of a union are not relevant for its encoding.

8.1.4 Definedness

During these encoding definitions you might have noticed a certain trend: No decoding function ever requires an `AbstractByte` to be `Uninit`. It is always requires it to be `Init`, or it is irrelevant in case of padding bytes. Similarly, no decoding function every requires the provenance of an `AbstractByte` to be `None`.
In other words, both initializing `AbstractBytes` and enriching them with provenance never hinders a list of `AbstractBytes` from being decoded into a Value. This works as these two operations only make our `AbstractBytes` more *defined*. Let us formalize this notion.

Given two `AbstractBytes` `a` and `b`, we write `a` $\leq$ `b` to express that `b` is as least as defined as `a`.
We define it as follows:

- `Uninit` $\leq$ `a` for all `a: AbstractByte`.
  Everything is at least as defined as `Uninit`. Hence, `Uninit` is the lowest tier in our definedness hierarchy.
- `Init(byte, None)` $\leq$ `Init(byte, prov)` for all `byte` and `prov`.
  In other words, initializing an `AbstractByte` can never make it less defined.
- `a` $\leq$ `a` for all `a`.
  This expresses that $\leq$ is reflexive. Everything is at least as defined as itself.
- For all further `a`, `b`, the relation `a` $\leq$ `b` does not hold.

There are three level to definedness: Being uninitialized; initialized without provenance; and initialized with provenance. But note that `AbstractBytes` that are related under $\leq$ can never differ in their `u8`; or their provenance.

So `Init(byte1, None)` $\not\leq$ `Init(byte2, None)` if `byte1 != byte2`;
and `Init(byte, prov1)` $\not\leq$ `Init(byte, prov2)` if `prov1 != prov2`.

Thus given `a` $\leq$ `b`, the `AbstractByte` `b` never *changes* anything already defined by `a` it might only extends it further.

In order to cover the encoded representation of Values, we lift the notion of definedness to lists of `AbstractBytes`. Given two `List<AbstractByte>` `a` and `b`, the relation `a` $\leq$ `b` holds, if `a` and `b` have the same length, and they are pairwise related by $\leq$.
We have now captured what definedness means for encoded Values. Yet, the same concept is also applicable to unencoded Values.
We can defined it as follows:

- `Value::Pointer { address, provenance: None }` $\leq$ `Value::Pointer { address, provenance: prov }` for all `prov`.
- `Value::Tuple(vals1)` $\leq$ `Value::Tuple(vals2)` holds, if it holds pairwise for `vals1` and `vals2`.
- `Value::Union(chunks1)` $\leq$ `Value::Union(chunks2)` holds, if it holds pairwise for `chunks1` and `chunks2`.
- `a` $\leq$ `a` for all values `a`.
- For all further `a`, `b`, the relation `a` $\leq$ `b` does not hold.

Now let us address the final extension of $\leq$. Whenever the decoding of a list of `AbstractBytes` fails, the `decode` function will return `None`. Thus, if we intend to relate decoded Values using $\leq$, we have to address that they might be `None`. We define $\leq$ for `Option<Value>` as:

- `None` $\leq$ `a` for all `a`.
- `Some(a)` $\leq$ `Some(b)`, if `a` $\leq$ `b`.
- For all further `a`, `b`, the relation `a` $\leq$ `b` does not hold.

8.2 Properties

We are now ready to define the central properties that we want our encodings to satisfy. We start by presenting the *monotonicity properties*: They express that `encode` and `decode` are monotone with respect to $\leq$.

Monotonicity of `encode`

- Given two values `v1` and `v2` with `v1` $\leq$ `v2`, which are both valid for the well-formed type `ty`. Then `ty.encode(v1)` $\leq$ `ty.encode(v2)`.

This property expresses that, the more defined the input to `encode` is, the more defined is its output. To illustrate this property let us compare two pointer values `v1` and `v2`. Both share the same address but in comparison to `v1`, the pointer `v2` has a provenance. In this scenario `ty.encode(v1)` will consist only of `AbstractBytes` that have a `None` provenance, whereas `ty.encode(v2)` will have some provenance associated to each of them. Due to the fact that both pointers share the same address, the lists will not have any other differences. Thus it follows that `ty.encode(v1)` $\leq$ `ty.encode(v2)`.

Monotonicity of `decode`

- Given two lists of `AbstractBytes` `bytes1` and `bytes2` and a well-formed type `ty`. Then `ty.decode(bytes1)` $\leq$ `ty.decode(bytes2)`.

To make this property a bit more tangible, let us consider an example. Consider what happens when decoding `bytes2 = [Init(1, None)]` as a `bool`; it will result in a `Value::Bool(true)`. Let us now compare this to the less-defined `bytes1 = [Uninit]`. Then the expression `ty.decode(bytes1)` $\leq$ `ty.decode(bytes2)` evaluates to `None` $\leq$ `Some(Value::Bool(true))`, which does indeed hold.

Roundtrip I

- Given a value `val` which is valid for a well-formed type `ty`.
  Then `ty.decode(ty.encode(val)) == Some(val)`.

You might remember this property from the beginning of this Chapter. It effectively expresses that storing a value and then loading it back does not change that value. This property is the easiest to verify. We encourage the reader to test this property on a few MiniRust types.

The converse property is a little more nuanced:

Roundtrip II

- Given `bytes`, a list of `AbstractByte`, a value `val` and a well-formed type `ty`. If `ty.decode(bytes) == Some(val)`, then `ty.encode(val)` $\leq$ `bytes`.

The reader might have expected the outcome `ty.encode(val) == bytes`: As decoding `bytes` yields `val` one might have expected that encoding `val` again yields `bytes`.
To explain why equality does not hold in this case, consider `ty` to be any tuple type with at least one byte of padding. Further let `bytes` be any fully initialized list of `AbstractBytes`. Decoding `bytes` and then encoding it back will overwrite

the padding bytes by `Uninit`. Thus, even if all other `AbstractBytes` stay the same `ty.encode(val) == bytes` does not hold, but instead only `ty.encode(val)` $\leq$ `bytes` does.

8.3 Coq implementation

We will utilize this final section to shed some light on our Coq implementation of MiniRusts encoding properties. First, we will explain how we formalized this task in Coq; and then we will give a brief overview of our proof structure. The formalization can be divided into three parts.

- Data types and traits,
- Functions, and
- Properties,

8.3.1 Data types and traits

Formalizing MiniRusts data types in Coq was a rather straight-forward transformation. Every `enum` became an `Inductive`; every `struct` became a `Record`; and every `trait` became a `Class`.
To provide an example, the `enum AbstractByte` was formalized as follows:

```
Inductive AbstractByte : Type :=
 | Uninit : AbstractByte
 | Init : byte -> option Provenance -> AbstractByte.
```

... which is a pretty direct translation from ...

```
enum AbstractByte {
    Uninit,
    Init(u8, Option<Provenance>),
}
```

Another example we want to quickly mention is the `trait Memory`. With regard to encoding, it is responsible for providing the `Provenance` type, defining the `Endianness` and the size of a pointer. This is its formalization in Coq.

```
Class Memory := {
  PTR_SIZE : Size;
  ENDIANNESS : Endianness;
  Provenance : Type;
  P_EQ : Provenance -> Provenance -> bool;
  P_EQ_REFLECT : forall x y, Bool.reflect (x = y) (P_EQ x y);
  PTR_SIZE_GT0 : PTR_SIZE > 0;
}.
```

8.3.2 Functions

Formalizing MiniRusts functions in Coq proved to be a harder task. This was mostly due to the fact that they were ocassionally written in an imperative style.
Further, many recursive functions implemented in MiniRust could not be understood by Coq. Coq only accepts recursive functions, if it can guarantee that they terminate.
Two examples of this are the `encode` and `decode` functions:

```
Fixpoint decode (ty : Ty) (l : list AbstractByte) : option Value :=
 match ty with
  | TInt size signedness => decode_int size signedness l
  | TBool => decode_bool l
  | TPtr ptr_ty layout => decode_ptr ptr_ty layout l
  | TTuple fields size => decode_tuple fields size l decode
  | TArray elem count => decode_array elem count l decode
  | TUnion fields chunks size => decode_union fields chunks size l
 end.
```

As you can see, `decode_tuple` and `decode_array` get `decode` itself as recursive argument. Proving to Coq that this is valid, and can never cause infinite loops, due to the fact that we recurse through types, were non-trivial. This was mostly due to the fact that composite types like tuples that contain lists of tuples of types, makes it hard for Coq to understand this abstraction.

8.3.3 Properties

Defining the properties that we intend to prove was again a rather simple task. For example, let us look at how we defined the monotonicity property of `decode`:

```
Definition mono2 (ty: Ty) :=
  forall (l1 l2: list AbstractByte),
  (le l1 l2)
  -> le (decode ty l1) (decode ty l2).
```

The hardest part of defining the properties correctly was defining `wf`, the well-formedness requirement for types. The issue was that the well-formedness of a tuple type depends on the well-formedness of all of its fields types. This recursive definition was again problematic for Coq.
While proving the desired properties we noticed that we would need to prove another property first. Thus, we have added a fifth property `ENCODE_LEN`. `ENCODE_LEN` expresses that encoding a value generates a number of `AbstractBytes` equivalent to the `size` of the Type.
After defining all properties, we tied them together in the `Record Props`:

```
Record Props ty := {
  PR_WF : wf ty;
  PR_RT1 : rt1 ty;
  PR_RT2 : rt2 ty;
  PR_MONO1 : mono1 ty;
  PR_MONO2 : mono2 ty;
  PR_ENCODE_LEN : encode_len ty
}.
```

which allows us to define the central `Theorem` that we want to prove:

```
Theorem generic_props (ty: Ty) : wf ty -> Props ty.
```

8.3.4 Proofs

The proofs were the most involved part of this Coq implementation; they alone spanned over four thousand lines of code!

The general structure of the proof is an induction over types. This is necessary, as the properties can only be shown for composite types like tuples if we assume that their fields already satisfy them.

```
Theorem generic_props (ty: Ty) : wf ty -> Props ty.
Proof.
induction ty using ty_ind; intros Hwf.
- apply (bool_props Hwf).
- apply (int_props Hwf).
- apply (ptr_props Hwf).
- apply (tuple_props H Hwf).
- apply (array_props IHty Hwf).
- apply (union_props Hwf).
Qed.
```

As one might have guessed, Coq was not able to deliver a usable induction scheme for this problem, again due to the fact that Tuple has a weird recursive occurence. Thus we have implemented our own induction scheme `ty_ind`, which we proved by using another induction over the depth of our types.

Chapter 9

Conclusion

In this final Chapter, we summarize our progress on MiniRust. Additionally, we mention opportunties for future improvement.

Our first contribution to the MiniRust project consisted of the specr lang to Rust compiler.
In order to streamline further development on specr lang compilers, we presume that implementing name resolution is at some point advisable. The current system cannot rely on the fact that two equal identifiers refer to the same object, which makes it easy to introduce naming collisions during code transformations.

A simple yet restrictive alternative would be to disallow the programmer to introduce multiple items with the same name. This could be checked rather easily.

Next up, we extended MiniRusts specification to cover more Rust programs. It is now able to express termination of programs; the use of dynamic memory; side-effects like printing; static memory allocations; comparison of integers; and a few more.
MiniRust is still missing a significant portion of features to represent all Rust programs. For example enums, floating point numbers, unwinding, unsized types like slices and `dyn trait` come to mind.
Next, we implemented three tools for MiniRust. `minimize`: the Rust to MiniRust compiler; our pretty printer for MiniRust program; and our test suites. We presume that while `minitest` grows, at some point a parser for MiniRust programs will be valuable.

In addition to the previous contributions, we also proved certain properties about MiniRust in Coq. We proved that MiniRusts representation of values in memory is sound, but there is still a lot more to do!
The next step would be to verify the following: "Making an AbstractByte more defined can never introduce UB into a program". This more general statement does not only concern the encoding relation, but it concerns the full specification.

In order to simplify proving properties about MiniRust and its programs in Coq, there are plans to eventually implement a specr lang to Coq compiler. We predict that implementing such a compiler will be notably harder than the specr lang to Rust compiler. Specr lang and Rust are syntactically very similar, so that most of the code could remain unchanged when compiling. Restricting the features that specr lang inherits from Rust might make this second compiler significantly easier. Further, a system that allows to prove that certain recursive functions terminate, needs to be set in place.

Bibliography

[Knu84] Donald Ervin Knuth. "Literate programming". In: *The computer journal* 27.2 (1984), pp. 97–111.

[Pro] *WG14. Defect Report 260.* 2004. URL: `https://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_260.htm`.

[Stu+15] Nicolas Stucki et al. "RRB vector: a practical general purpose immutable sequence". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming.* 2015, pp. 342–354.

[PH16] David A Patterson and John L Hennessy. *Computer organization and design ARM edition: the hardware software interface.* Morgan kaufmann, 2016.

[DB17] Manjeet Dahiya and Sorav Bansal. "Modeling undefined behaviour semantics for checking equivalence across compiler optimizations". In: *Hardware and Software: Verification and Testing: 13th International Haifa Verification Conference, HVC 2017, Haifa, Israel, November 13-15, 2017, Proceedings 13.* Springer. 2017, pp. 19–34.

[Lee+18] Juneyoung Lee et al. "Reconciling high-level optimizations and low-level code in LLVM". In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018), pp. 1–28.

[Jun20] Ralf Jung. "Understanding and evolving the Rust programming language". In: (2020).

[Tea20] The Coq Team. *The Coq proof assistant.* 2020. URL: `https://coq.inria.fr`.

[KN23] Steve Klabnik and Carol Nichols. *The Rust programming language.* No Starch Press, 2023.

[Pre] *A minimal 'syn' syntax tree pretty-printer.* URL: `https://crates.io/crates/prettyplease`.

[A m] *A-Mir-Formality.* URL: `https://github.com/nikomatsakis/a-mir-formality/`.

[Uit] *A test framework for testing rustc diagnostics output.* URL: `https://crates.io/crates/ui_test`.

[Ub] *Behaviour considered undefined.* URL: `https://doc.rust-lang.org/reference/behavior-considered-undefined.html`.

[Num] *Big integer implementation for Rust.* URL: `https://crates.io/crates/num-bigint`.

[Cer] *Cerberus.* URL: `https://www.cl.cam.ac.uk/~pes20/cerberus/`.

[Fer] *Ferrocene Language Specification.* URL: `https://github.com/ferrocene/specification`.

[Im] *Immutable collection datatypes.* URL: `https://crates.io/crates/im`.

[Jun] Ralf Jung. *Pointers Are Complicated, or: What's in a Byte?*

[Mir] *Miri.* URL: `https://github.com/rust-lang/miri`.

[Syn] *Parser for Rust source code.* URL: `https://crates.io/crates/syn`.

Appendix A

Appendix

A.1 Constructing a MiniRust program from scratch.

```
let fn_name = FnName(Name::from_internal(0));
let local_name = LocalName(Name::from_internal(0));
let block_name = BbName(Name::from_internal(0));

// define the type `u32`.
let u32_type = Type::Int(IntType {
    signed: Unsigned,
    size: Size::from_bytes(32).unwrap(),
});

// define the place type `u32@align(4)`.
let u32_place_type = PlaceType {
    ty: u32_type,
    align: Align::from_bytes(4).unwrap(),
};

// define the `_ = exit();` terminator.
let exit_call = Terminator::CallIntrinsic {
    intrinsic: Intrinsic::Exit,
    arguments: list![],
    next_block: None,
    ret: None,
};

// define an integer constant `42`.
let const42 = Constant::Int(Int::from(42));
let const42 = ValueExpr::Constant(const42, u32_type);
```

```
// define the basic block bb0.
let bb0 = BasicBlock {
    statements: list![
        Statement::StorageLive(local_name),
        Statement::Assign {
            destination: PlaceExpr::Local(local_name),
            source: const42,
        }
    ],
    terminator: exit_call,
};

let mut blocks = Map::new();
blocks.insert(block_name, bb0);

let mut locals = Map::new();
locals.insert(local_name, u32_place_type);

// define our function.
let f = Function {
    args: list![],
    blocks,
    locals,
    ret: None,
    start: block_name,
};

let mut functions = Map::new();
functions.insert(fn_name, f);

// define the program.
let p = Program {
    functions,
    start: fn_name,
    globals: Map::new(),
};
```