Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Implementing Enums in MiniRust

Bachelor Thesis

Timon Meyer

Programming Language Foundations Lab
Swiss Federal Institute of Technology (ETH) Zurich

Supervision

Prof. Dr. Ralf Jung

E-mail address: timeyer@ethz.ch
ETH student ID: 21-922-935

Submission date: March 24, 2024

Abstract

Rust is a low-level programming language with a strong focus on safety. By design it eliminates many sources of mistakes that programmers can make using a combination of its rich type system and ownership model for data. However, it is lacking in a specification of its operational semantics on machine level. This is where MiniRust comes into play. MiniRust is a project by Prof. Dr. Ralf Jung to provide a reference interpreter and memory model that together give the operational semantics. Regrettably it still lacks many features, one of which was the enum datatype that is central to Rust. Enums feature prominently in types like `Option<T>` and `Result<T, E>` that signify either that a value might not exist or a result of an operation that might fail. This thesis adds the enum datatype to the MiniRust project, enabling it to model many more programs.

Acknowledgements

I want to thank Prof. Dr. Ralf Jung for his great support on how to present my work and his directions when I did something the unnecessarily hard way again.

I further want to thank all the members of the Programming Language Foundations Lab and Programming Methodology Group at ETH who came to my presentations on the topic and asked good questions that gave me a hint at where I needed to elaborate my explanations.

Contents

1. Introduction

This thesis is about implementing the operational semantics of Rust enums in MiniRust. Rust[1] as a programming language has become very popular due to its safety guarantees, while being fast and allowing to write low-level code at the same time. It does so by combining a powerful type system with an ownership model. The type system is heavily inspired from functional programming and its arithmetic data types. One class of those data types are sum types which in Rust are represented by the enum and features prominently in the standard library with the `Option<T>` and `Result<T, E>`. Since those types signify an optional value or a result of an operation which can fail, they are used everywhere.

Rust is missing a specification for its operational semantics on machine level. There is Ferrocene[2] which, for its qualified Rust toolchain for safety-critical systems, has a specification of the operational semantics of Rust. However, it is written in plain English and considers differences between the specification and the compiler to be a mistake of the specification. This is one of the reasons why the MiniRust[3] project exists. This project by Prof. Dr. Ralf Jung aims at providing the operational semantics of Rust by giving a reference interpreter and a memory model. While MiniRust is still lacking many features, enum data types are added as part of this thesis.

It starts in chapter 2 with a detailed explanation of what enums in Rust are and how they work, to then to give an overview of MiniRust and how it handles types. Using this information, chapter 3 will explain the model of enums together with the implementation of their semantics, giving the reasoning behind. Finally, chapter 4 will discuss some formal aspects of enums and their implementation in MiniRust. It ends with an overview on what Rust programs that use enums can now be modeled.

2. Background

2.1. Rust enums

Before enums can be added to MiniRust one first has to understand how enums work in Rust.

In Rust an enum is a type that has different variants. Each variant can have named or unnamed fields, as shown in listing 1. A variant without fields is called fieldless.

```rust
enum MyEnum {
    FieldlessVariant,
    UnnamedFields (Int),
    NamedFields { name: String, some_field: bool },
}
```

Listing 1: An example enum. `Int` is an arbitrary precision integer.

Each value of an enum is of exactly one variant. For example a value of `MyEnum` could be a `FieldlessVariant` or `UnnamedFields(42)` or some other value for a variant, but only one of those at a time.

This is the main difference between an enum and a union. In a union the underlying data is accessed as the type given by the member, but multiple members might be valid.

Each enum variant is identified by an enum-wide unique discriminant. This discriminant is encoded into enum values which then allows the program to discern of which variant a given value is. Therefore the set of possible values for each variant can be described by tuples of the discriminant and its fields. Then the set of possible values for the enum is the union of the values for each variant. This is sometimes called the disjoint union of the variants.

If in the example given in listing 1 the discriminants were assigned 0 for `FieldlessVariant`, 1 for `UnnamedFields` and 2 for `NamedFields` then the sets of values for the variants would be as follows:

$$\begin{aligned}
\mathcal{V}_{\text{FieldLessVariant}} &= \{(0)\} \\
\mathcal{V}_{\text{UnnamedFields}} &= \{(1, i) \mid i \in \mathbb{Z}\} \\
\mathcal{V}_{\text{NamedFields}} &= \{(2, s, b) \mid s \in \mathcal{V}_{String}, b \in \{0, 1\}\} \\
\mathcal{V}_{\text{MyEnum}} &= \mathcal{V}_{\text{FieldLessVariant}} \cup \mathcal{V}_{\text{UnnamedFields}} \cup \mathcal{V}_{\text{NamedFields}}
\end{aligned}$$

Rust's type system is heavily inspired by functional programming languages like Haskell which use algebraic data types for combining variables. The two most common classes of algebraic types are product types, which in Rust are implemented as structs and tuples, and sum types which in Rust are implemented as enums. The classes got their name because of how the set of all possible values is constructed. For product types it is the cartesian product and for sum types it is the disjoint union. Product types allow any combination of values for the types while sum types allow any value for a variant, but not multiple variants at once.

As the word "disjoint" indicates, Rust strictly enforces the invariant that each value is of exactly one variant. After a variant was constructed (for example with `MyEnum::UnnamedFields(42)`) there is no way for the programmer to change which variant is used for this value. But since the type of the value is the enum type there is no way to access the fields like in structs, where the name or index of the field can be used to access it. For a programmer there is no way of downcasting the enum type into one of its variants.

However, Rust provides pattern matching which can be used to match an enum value against the variants of the enum. This then allows the programmer to access the fields. There are many forms how pattern matching can be used as shown in listing 2 on the next page.

```rust
/// This function will try to access the 'Int' in
/// 'MyEnum::UnnamedFields' in four different ways.
pub fn access_the_integer(my_enum: MyEnum) {
    // most common: using a 'match' statement.
    match my_enum {
        MyEnum::UnnamedFields(i) => {
            println!("Got Int! {i}");
        },
        _ => {},
    }

    // if-let, useful when the other values do not matter.
    if let MyEnum::UnnamedFields(i) = my_enum {
        println!("Got Int! {i}");
    }

    // while-let, useful for polling until some end value.
    while let MyEnum::UnnamedFields(i) = my_enum {
        println!("Got Int! {i}");
        break;
    }

    // let-else
    // useful if only fields of this variant are used later.
    let MyEnum::UnnamedFields(i) = my_enum else {
        return; // we need to exit the function otherwise
    }
    println!("Got Int! {i}");
}
```

Listing 2: Accessing the field of the enum through various pattern matches.

2.1.1 Thesis related MIR

As the previous section showed, the Rust programming language seriously limits how the programmer can interact with enums. Modeling those very high-level interactions in a reference interpreter would be tough and indeed MiniRust's interpreter does not operate on such code. Instead it operates on a language that strongly resembles the Rust middle intermediate representation (MIR). Code generation in the Rust compiler runs through multiple stages. First the program code gets parsed into the high-level intermediate representation (HIR). This then gets lowered into the middle intermediate representation (MIR). Finally, the code gets lowered into bytecode generating backends like LLVM to produce the executable. On each intermediate representation the compiler performs various optimizations and transformations.

What does the Rust middle intermediate representation look like? Since it only concerns the code, the MIR of a program is a collection of functions. Each function is defined by basic blocks, in which all statements are being run one after each other. A statement may modify the execution environment, for example `Assign` assigns an `RValue` to a `Place`, effectively storing a value somewhere in memory. `RValue`s represent a single computation of a value, like mathematical operations or loading a value from a `Place`. Places describe how to access a value in a local variable. They give a local variable and then apply projections on it, which can modify the pointer to the value in the local variable and its type. Depending on the type those projections might be a field access or an offset.

`RValue`s only read from the execution environment, they do not write to it. The statements all modify the execution environment.

Control flow is handled by the terminators of the basic blocks. They describe which block to execute next, possibly depending on some condition. Examples for terminators are `Return`, which returns from this function, or `SwitchInt`, which switches to the next block, depending on the result of an integer-like `RValue`.

This very clear structuring of code into control flow, execution environment modification, computations and value accesses makes it a lot easier to argue about a given program. For this reason most Rust-specific optimizations are done on this level. This makes it a good basis for MiniRust, especially after considering that this representation can be accessed from the Rust compiler API which is used to translate Rust programs to MiniRust.

```
fn foo(_1: u32) -> u64 {
    let mut _0: u64; // define local _0 as return value

    // start basic block 0
    bb0: {
        // assign to the place of the local _0
        // the RValue of the cast of the local _1 to u64
        _0 = move _1 as u64 (IntToInt);
        return; // terminator returning from the function.
    }
}
```

Listing 3: Example MIR of a function that converts a u32 to a u64.

Listing 3 gives an example on what the MIR for a very simple function looks like when it is printed in a human-readable format. This highlights the main issue of the MIR: No high-level programming language would use such a syntax, as the intent of the program is hidden behind all the effects of individual statements.

The following paragraphs go into more detail for parts of the MIR that are relevant to enums, because they interact with them directly or indirectly.

Aggregate RValue
In Rust, Enum variants or structs are constructed from their fields in one go. This enforces that all fields are initialized at the moment of assignment.

Comparing this to initializing fields in a Java class, it could happen, that whoever writes the constructor forgets to initialize a field. This can lead to unexpected behavior later during execution. Rust's design prevents such programming mistakes.

SetDiscriminant statement

Earlier versions of the Rust compiler did actually split the aggregate assignments into individual field assignments during a transformation of the MIR called "deaggregation". Those field assignments were then followed by a statement to write the discriminant, `SetDiscriminant`. Later the deaggregation step got moved further down the compilation pipeline, but for example `async` coroutines still use the original SetDiscriminant statement to set the state of the coroutine.

Discriminant RValue

In order to have control flow that depends on the discriminant, it has to be read from the enum value. This is done using `RValue::Discriminant`.

SwitchInt terminator

The main control flow terminator of the MIR is `SwitchInt` which works similar to C's `switch case`. It switches to a branch determined by the value of the integer, `bool` or `char` it was given. It also has a fallback for the cases which are not explicitly specified.

Further down the compilation pipeline this might be compiled into a series of conditional branches or a branch table, depending on both computer architecture and the kind of value being switched on.

Variant downcast

The most important projection for enums is the `Downcast` projection which projects a specific variant onto this `Place`. This tells the compiler which variant should be considered for further projections, like for example accessing the fields of the enum variant.

Note that this downcast cannot be done by the programmer manually and has been generated by the compiler in an earlier stage, which ensures that the downcast is always correct. An example for this is Pattern matching on enum variants. When it reaches the MIR it has been transformed into a switch on the discriminant. In the subsequent blocks the enum value can be safely downcast based on the discriminant required to reach the block[1].

[1] This assumes that the enum value is correct if the discriminant could be read, leading to some issues described later in chapter 4.1.

Field access
After a `Place` of an enum value has been downcast into a variant, the fields can be accessed using a `Field` projection. Applying this field projection on the `Place` of an enum value without a `Downcast` is considered invalid MIR. The field projection would have no idea to which variant the accessed field belongs, and therefore also not know the type and location of the field.

2.1.2 Never Type

Sometimes code cannot return, like for example in listing 4. The loop will continuously execute `f` without end, and therefore the function will return nothing. With no value there is also no type, making it impossible to write function signatures. Rust's solution to this is the Never type `!`. It signals that any value with this type can never occur, meaning it cannot be instantiated or read in any way.

```rust
/// Calls the function f forever.
fn do_forever(f: impl Fn() -> ()) -> ! {
    loop {
        f();
    }
}
```

Listing 4: Function running the same function forever.

This is different from the more known unit type `()` returned by `f` in listing 4, of which there is exactly one value. In other languages like C the unit type is more commonly known as `void`.

While Never is not related to enums in Rust, it has the same semantics as an enum with no variants (zero-variant enum)[4], because to instantiate an enum means to instantiate a variant. Such an enum could also never be read. The two ways of reading from an enum value are by downcasting it into a variant which is impossible with no variants, or by calculating its discriminant. But with no valid discriminant this has to fail as well.

The only difference between a zero-variant enum and the Never type is that the Never type can be coerced into any other type.

2.1.3 Memory representation

In order for MiniRust to specify the state of the execution environment for the operational semantics it will have to deal with the bits and bytes that are manipulated in the computer executing the code. This poses the question on how to represent the mathematical concept of enums from chapter 2.1 as a combination of discriminant and fields in memory.

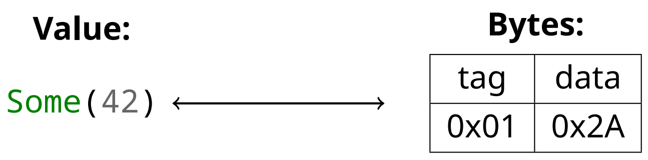
The data of the fields has to be encoded as usual, since Rust allows references to those fields. Dereferencing those references should not deal with the fact that those fields are part of an enum. The discriminant, however, can be encoded more freely. It may not overlap with data, but otherwise the compiler is free to do what it wants. The encoded discriminant is called the tag and the method of conversion is called tag encoding.

The first tag encoding used in the compiler is called the direct tag encoding. In this case the tag is just another field containing the discriminant. The example in listing 6 shows the memory representation of the specialization for `Option<T>` from the standard library shown in listing 5 for a byte `u8`. The discriminants are automatically assigned 0 for `None` and 1 for `Some(T)`. It is important to mention here that the actual memory representation is unstable and might be subject to change[5]. This example refers to the output of the Rust compiler at the time of this thesis[2].

```rust
enum Option<T> {
    None,    // None automatically gets discriminant 0.
    Some(T), // Some automatically gets discriminant 1.
}
```

Listing 5: `Option<T>` as in the standard library.

[2]This would be the nightly version of rustc 1.78.0 from 29th february, 2024.



tag	data
0x01	0x2A

Listing 6: Memory representation of `Option<u8>`

While this way of encoding enums is simple, it often is not space efficient. Consider the specialization of `Option` for references (`Option<&R>`): Direct tag encoding would use 64 bits to encode the reference and add a field of at least one bit for the tags of the two variants. Due to the alignment requirements of the reference this would blow up the size of the `Option<&R>` to 128 bits.

This can be improved by using the niche tag encoding. Rust references are not allowed to be `null`. The compiler uses this information to find out that the data of `Some(&R)` has a so-called niche with an invalid value, 0. Furthermore, `None` has no data at this place in memory. So the value 0 can be used as the tag for the `None` variant and be written where the pointer would be for `Some(&R)`. Reading the value of the reference now reveals which variant it is. A zero indicates `None`, while all other values belong to the variant `Some(&R)`. Without the separate tag field, the size of an `Option<&R>` is 64 bits.

More formally the niche tag encoding uses a niche, with enough invalid values to store the tags of the other variants, of a field where no other variant has data. This field stores the tag, and the variant with the data there is called the untagged variant. This means that to encode the untagged variant no tag is written, as in some sense the data is the tag. For the other variants the tag is computed using some arithmetic operations on their discriminant and written into the field of the tag. This is why the tagged variants may not have any data there, otherwise it would be overwritten.

Later, the discriminant is calculated by reading the tag and reversing the operations. Though if the value at the tag was valid for the data of the untagged variant, then its discriminant would be returned.

2.1.4 Undefined Behavior (UB)

Sometimes the program encounters a state which is invalid, or runs an operation on values for which it is undefined. The result is called Undefined Behavior, or UB in short, as many vastly different things are possible from this point on. Usually this happens because of some invariant that has been violated without throwing an error.

Rust prevents many sources of undefined behavior with its type system and ownership model. However, it can still be produced using code wrapped in an `unsafe { .. }` block that will be discussed in the next chapter.

The most important case of undefined behavior in Rust for this thesis is reading or writing an invalid value. This usually leads to reading a discriminant that does not belong to any variant of the type of the enum.

Dereferencing the pointer in listing 7 produces undefined behavior since `Option<bool>` uses a niche tag encoding with 2 for the tag of `None`[3]. 12 is neither the tag of `None`, nor a valid value for the boolean which uses 0 or 1, therefore this is an invalid value that would be assigned to `x`.

```
    let x = unsafe {
        let ptr: *const u8 = &12;
        *(ptr as *const Option<bool>)
    };
```

Listing 7: Unsafe example that leads to UB.

2.1.5 Unsafe Rust

Unsafe Rust is all the code that is wrapped in an `unsafe { .. }` block. The main difference is that unsafe code may dereference raw pointers.

Raw pointers, like for example `*const String`, are different from references like `&String` in that they are not subject to the ownership model and can be cast into any pointer type as shown in listing 8. Therefore it is unsound

[3]At the time of writing this, as mentioned before.

to arbitrarily dereference them. In fact, since most of Rust's undefined behavior has to do with reading or writing invalid values for their type, this is the only way to produce behavior considered undefined in Rust. Only raw pointers allow generally unsound behavior, like reading a value that has expired, reading or writing a value of the wrong type, or dereferencing a null pointer.

```rust
unsafe {
    let x: String = String::new();
    let x_ref: &String = &x;
    let x_ptr: *const String = &x;

    consume(x); // move x out of scope.

    // compiler will complain that x has gone out of scope.
    let _ = *x_ref;
    // compiler will accept this but it will produce
    // undefined behavior during execution.
    let _ = *x_ptr;
}
```

Listing 8: The main difference between a pointer and a reference.

For a high-level overview on unsafe Rust see the Rustonomicon[6]. This thesis will not go into more details on unsafe Rust as it does not offer any additional ways of interacting with enums.

2.2. Ferrocene

Ferrocene[2] by Ferrous Systems is a Rust toolchain qualified for safety-critical environments. It is based on Rust 1.68, released on March 9th 2023[7].

A prerequisite of the qualification is a formal specification, which is given in the Ferrocene Language Specification (FLS)[8]. It has been compiled from existing Rust documentation like the Rust Reference[9] and the Rustonomicon[6]. However, this specification only describes the behavior of the Rust compiler. Any difference between FLS and the Rust compiler is considered to be a mistake in the specification. This makes it less useful to define what behavior is a bug in the compiler and what is actually part of the Rust programming language. Furthermore, the Ferrocene Language Specification is written in English which makes it harder to produce a mathematical model based on it. Those two issues are some of the reasons why further work is being done on a Rust specification.

What does the Ferrocene Language Specification have to say about enums? Not much more than what was introduced in the background on Rust. It specifies the syntax, what discriminants can be assigned to variants, and what discriminants are used when they are not assigned manually. It further states that a valid value must have a discriminant specified by the enum type[10].

Additionally, it specifies how the pattern matching works and how enum values are initialized. But this is all, since those are the only enum related operations exposed to the programmer. There are no details about how those operations interact with a computer.

2.3. MiniRust

MiniRust[3] by Prof. Dr. Ralf Jung is a project that aims to develop a specification of the operational semantics of Rust, with a particular focus on unsafe Rust. The project is located on GitHub. It consists of the specification in /spec and additional tools like a test suite. Chapter 2.3.4 will discuss the MiniRust toolchain relevant to this thesis in some more detail.

It was mentioned in the introduction that the specification is given as a reference interpreter. This means that running Rust code on any computer should result in the same behavior as the one displayed by the interpreter given in the specification. At the heart of this interpreter is the `Machine` which contains the program, memory, thread state and other bookkeeping data. The machine can be executed step by step. On each step it chooses a thread to execute one instruction. The step may end the execution when the program terminates or undefined behavior occurs. The semantics of the instructions together with the scheduling give the operational semantics in step semantics.

In some ways MiniRust specifies a wider spectrum of behavior than Rust. This is because MiniRust's specification is not there to restrict Rust's possible future behavior. Sometimes, like in the memory representation, further optimizations might be added to the compiler and therefore only some guarantees are given.

2.3.1 Type Handling

In MiniRust, types are defined in terms of fundamental data types given in /spec/lang/types.md. They reflect the Rust data types in that there is a boolean data type, integers, pointers, tuples, arrays, unions, and, with this thesis, enums. The types used by the programmer are then defined through those data types.

Values of MiniRust types exists on two layers. First there are the values, on which MiniRust actually operates. This layer intends to capture the mathematical concepts without having to deal with the memory representation and other limitations of hardware. The definition of those values can be found in /spec/lang/values.md. In the case of enums the value is an instance of a variant, since there are no instances of the entire enum type. This is similar to the values of Rust enums described in chapter 2.1 which combine the discriminant with the fields. This thesis will refer to those variant instances when it mentions enum values in the context of MiniRust.

The second layer is what is actually stored in memory. MiniRust describes a memory interface in /spec/mem that allows it to model, for example, concurrent access. Instructions usually do not directly interact with this layer other than to store or load a value. The way those instructions encode and decode MiniRust's types from memory is called the representation relation and can be found in /spec/lang/representation.md.

MiniRust furthermore enforces well-formedness. For each type and value there is some validation. For example the fields of tuples may not overlap, or an integer has to be in the range of its type. Value validation runs before encoding. If the validation succeeds, then the encoding of this value is guaranteed to succeed as well. But the bytes in memory might encode an invalid value. This is detected upon loading the value which leads to the interpreter stopping with undefined behavior, as loading an invalid value is undefined behavior as described in chapter 2.1.4.

Those well-formedness checks as well as some more for program syntax are described in /spec/lang/well-formed.md. Chapter 3.1.4 will discuss the well-formedness requirements of enum types.

2.3.2 Place expressions

Similar to Rust's `Place` and its projections there are place expressions in MiniRust. These evaluate to a pointer and a type.

Those expressions include getting the place of a local variable, a field of a tuple, or an element of an array.

Place expressions are used to load or store values from and to memory. It is here that the two layers of MiniRust's type handling come together. When `Statement::Assign` assigns a value to a place, it first checks that the value is well-formed and then uses the representation relation to encode it into bytes, which then are written to memory. `ValueExpr::Load` evaluates a place expression and tries to load a value from the pointer using the representation relation of the type that the place was evaluated into.

2.3.3 Tuples

MiniRust tuples will come up a lot, as they represent both Rust tuples and structs. For structs, the Rust compiler in the earlier stages of the compilation process turned all named field references into references by index. After this, all structs and tuples are treated equivalently in the MIR.

MiniRust uses tuples to describe both Rust tuples and structs. They are defined by their size, alignment and fields. Fields are described by their types and offsets. All bytes outside the fields but within the size are considered padding, and will be uninitialized when writing the tuple to memory, as defined by the representation relation of tuples.

Listing 9 gives examples of structs and tuples, all of which will be converted to a MiniRust tuple with a size of 8 bytes, an alignment of 4 bytes and the two fields. At offset 0 is the `u16` and at a 4 byte offset is the `i32` due to its alignment[4].

```
struct Unnamed(u16, i32);
struct Named { field_one: u16, field_two: i32 }

let x = (0u16, 42i32);
```

Listing 9: Examples of structs and tuples that will be converted to the same MiniRust tuple type.

[4]At least in the version of the compiler at the time of writing.

2.3.4 Toolchain

The implementation of enums is spread over multiple parts of the MiniRust toolchain. Here is an overview of the relevant parts for this thesis:

The `specification` in /spec of the MiniRust repository specifies the interpreter, its syntax and step semantics. This is what this thesis from now on refers to when it talks about MiniRust.

It is written in a language called `specr lang`. This programming language has been defined for this project with a syntax very similar to Rust, but has additional features like garbage collection, so the specification does not have to deal with Rust's ownership model.

`Specr-transpile` can be used to turn the specification into a Rust library crate that can further be used to create MiniRust programs and execute them. This program is located in the MiniRust-Tooling repository, where the library providing the data types used in the specification, libspecr, is located as well.

The MiniRust library crate produced by specr-transpile is used in the following two tools:

`Minitest` is located at /tooling/minitest in the MiniRust repository. It contains many test cases to verify that the MiniRust reference interpreter works as intended. Those test cases could also be considered to be examples of Rust behavior.

`Minimize` is located at /tooling/minimize in the MiniRust repository. This is a transpiler that translates Rust programs to MiniRust. It does so by using the rustc Rust compiler interface to lower the code into MIR and then mapping the MIR to MiniRust code. Afterwards the MiniRust programs are executed.

`Minimize` comes with its own test suite of Rust programs and their expected output (standard out and standard error) that it can run to test the transpiler.

The implementation of the translation of Rust enums to MiniRust is an important part of this thesis, since many Rust programs use enums in, for example, iterators.

3. Implementation

3.1. MiniRust

The following chapters describe the implementation of enums and some other related parts in MiniRust.

3.1.1 Enum type

An enum in MiniRust is defined as a 6-tuple $(\sigma, \alpha, \tau_\delta, \Delta, V, D)$. The first two parts denote the size σ and alignment α of the enum. The next two parts, Δ and τ_δ, are the set of every discriminant for this enum and their type, since discriminants in Rust can be of any integer type and both signed and unsigned.

Furthermore, the relation between the discriminants and the variants is given in V. The tag encoding from the Rust memory representation described in chapter 2.1.3 was split into two parts. The variants come with a so-called tagger to encode their discriminant. The discriminator D is used to decode the discriminant in the enum.

3.1.2 Enum variants

Discriminants can be somewhat arbitrarily assigned[11] to the variants, so MiniRust needs a general mapping from discriminant to variant. This mapping is given by $V : \Delta \to (\tau, T, \rho)$ where τ is the variant type that describes the data of the variant.

The tagger $T : \rho \to (\tau_{\mathbb{Z}}, \mathbb{Z})$ describes at which offsets $\rho \subset \mathbb{N}$, given by the variant, an integer with type $\tau_{\mathbb{Z}}$ has to be written to encode the discriminant. With a niche tag encoding, as discussed in chapter 2.1.3 about the Rust memory representation, the untagged variant would write nothing while some new experimental tag encoding might write two or even more integers.

Note that the concept of a variant type deviates quite heavily from Rust, where the variants contain fields. MiniRust allows the type of the variant to be any type as long as it has the same size as the enum, for reasons given later.

Using a tuple gives the semantics of the fields in a semantically equivalent way to Rust, see chapter 4.2 for a detailed discussion. Therefore tuples should be used to represent Rust enum variants. There is no reason to have any differences between accessing fields in a tuple and in a variant.

Using something other than a tuple for the variant type is allowed in MiniRust and just specifies more behavior than possible with Rust.

Listing 10 shows the implementation of the variants in the specification. It uses `Map` for relations, since it keeps track of its keys, eliminating the need to store the sets Δ and ρ separately. The `Map` can also be turned into an iterator of all its key-value pairs, which is particularly useful to iterate over the entries in the tagger in the representation relation.

```
pub enum Type {
    ..
    Enum {
        variants: Map<Int, Variant>,
        discriminant_ty: IntType,
        discriminator: Discriminator,
        size: Size,
        align: Align,
    }
}

pub struct Variant {
    pub ty: Type,
    pub tagger: Map<Offset, (IntType, Int)>,
}
```

Listing 10: The MiniRust enum definition.

3.1.3 Discriminator

The final element to discuss is the discriminator D. It is defined as a decision tree to decide which discriminant is encoded in the memory for an enum value. The tree nodes can be any value from $\mathcal{D}$ given below.

$$\begin{aligned}\mathcal{D} := &\{invalid\} \cup \\ &\{known(\delta \in \Delta)\} \cup \\ &\{branch(\tau_{\mathbb{Z}}, \mathbb{N}, \mathbb{Z} \to \mathcal{D})\}\end{aligned}$$

The two leaf nodes $invalid$ and $known(\delta)$ signify that the evaluation has come to an end. $invalid$ signifies that no valid discriminant is encoded and therefore the entire enum value is invalid. Reading a field or the discriminant from such an enum is undefined behavior. $known(\delta)$ signifies that the discriminant is known to be $\delta \in \Delta$.

Finally $branch(\tau_{\mathbb{Z}}, \mathbb{N}, \mathbb{Z} \to \mathcal{D})$ branches on some integer that is read using the integer type $\tau_{\mathbb{Z}}$ at the offset $\in \mathbb{N}$. Given this integer, some branch is selected by the child mapping $\mathbb{Z} \to \mathcal{D}$. Figure 3.1 gives an example of a complex discriminator where x denotes the value that was read to continue on this branch.

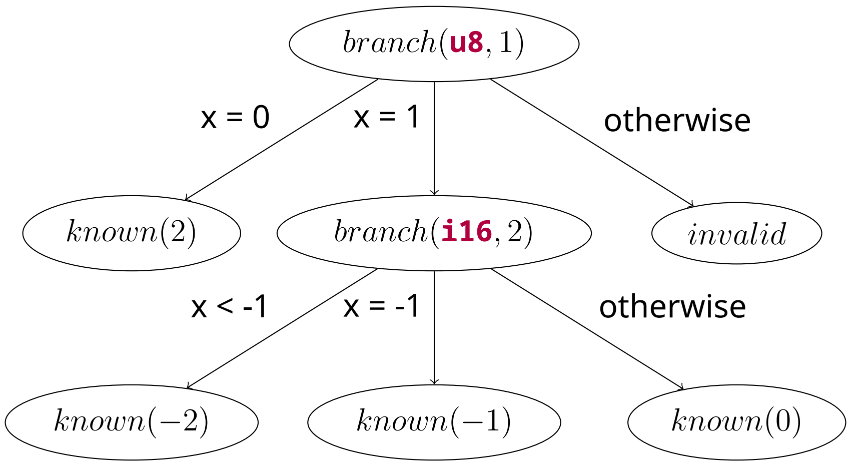


Figure 3.1: Example discriminator

The split of the tag encoding into the discriminator and taggers was originally suggested in a document about enum encoding by Jakob Degen[12]. However, the discriminator design underwent some changes. In the original sketch invalid values are encoded as missing entries in the branch. But this does not work for zero-variant enums, since a branch would require reading from a non-existent value. Further the branching now happens on arbitrary integer types instead of only bytes. This way, the enum definition does not have to care about the integer endianness.

Listing 11 shows how the discriminator is implemented. $branch(\tau_{\mathbb{Z}}, \mathbb{N}, \mathbb{Z} \to \mathcal{D})$ was made to work in practice by having the child mapping be from ranges to subtrees. This way a branch on a 64-bit value does not always require a map with 2^{64} entries, which would fill up the entire address space of a modern computer. Often the number of variants is small and a large chunk of values leads to the same result.

Secondly, a fallback was introduced for all values that have no entry in the map. This is to ensure that the branching is fully defined for any value that is read.

```rust
pub enum Discriminator {
    Invalid,
    Known(Int),
    Branch {
        offset: Offset,
        value_type: IntType,
        fallback: Discriminator,
        /// A left-inclusive right-exclusive range of
        /// values that map to some Discriminator.
        children: Map<Range, Discriminator>,
    },
}
```

Listing 11: The discriminant decision tree.

3.1.4 Enum type well-formedness

Not every enum that can be represented in MiniRust is valid, for various reasons described in this chapter. To check that the enum type is valid, the following well-formedness checks are done:

The size must be a multiple of the alignment.
This must hold for every type in MiniRust, enums included.

All discriminants can be represented using the discriminant type.
Otherwise some discriminants could not be assigned to variables of the discriminant type. As the biggest integer type is 128 bits long, this means that there is an upper and lower limit on which discriminants can be represented in MiniRust. However the same limitations apply to Rust.

The variant types must be well-formed.
Otherwise, downcasting an enum value into a variant could return an invalid type.

The variant types must have the same size as the enum.
This is mostly for practical reasons, because if some variant type was smaller,

the question of which enum bytes it covers would arise. When using tuples as mentioned before, the size can easily be set to be the size of the enum, which just generates padding if necessary. This limits what types other than tuples can be directly used for the variant types. Since this is additional specification of behavior, this limitation does not affect the semantics needed for Rust.

The variant type must not have a larger alignment than the enum.
A larger alignment for the variant type would mean that it might get misaligned when allocating an enum value.

The tagger must be well-formed.
Each value written must be representable by its given integer type, which in turn must be well-formed. Additionally each value must be written in the bounds of the enum.

Please note the absence of an alignment check for the integer that is written. Rust allows tags to be at an arbitrary offset. One example where this happens is shown in listing 12.

```rust
/// Inner is directly tagged with i16.
#[repr(i16)]
enum Inner {
    V1 = -32767,
    V2 = -32768,
}

#[repr(C, packed)]
struct WeirdNicheAlign {
    x: u8,
    /// Because of the packed layout inner has offset
    /// of 1, and it has a niche for 'Option' to use.
    /// 'None' will write a 16-bit tag at an offset of 1.
    inner: Inner
}
```

Listing 12: Enum with a Niche that has alignment 1 for a 16-bit value.

The discriminator must be well-formed.
The discriminator is recursively checked. `Discriminator::Invalid` is always well-formed. `Discriminator::Known` must return a discriminant with a corresponding variant.

For `Discriminator::Branch` the following must hold for its parts: The integer type must be well-formed. The value read must be in bounds of the enum, again the offset does not have to be aligned for the integer type. And the fallback must be well-formed.

Each entry in the child mapping is checked for four things: The entire range must be valid and representable given the integer type. Additionally, the range may not overlap with any other range in the children. And lastly, the resulting discriminator is checked for well-formedness.

3.1.5 Enum values and representation relation

A value of an enum is given as a tuple (δ, v) where δ is the discriminant of the variant and v is the value of the variant type.

Encoding the enum value is simple. First the value v gets encoded. Afterwards, the discriminant is encoded according to the tagger by writing the specified integers at their offset.

This obviously requires that the tagger does not overwrite any data. Otherwise, reading from fields where the tagger wrote something could result in invalid values. Currently, it is up to the user to verify this, as this is not covered by the well-formedness checks. The reason for not having a well-formedness check for this, is that MiniRust, at the moment, does not offer a method to find out which bytes of some type encode data and which are just padding. The overhead of implementing this was not worth it at the time of this thesis, considering that the Rust compiler already ensures that the tag and the data does not overlap, except in the untagged variant during niche encoding.

Decoding an enum from memory is a two-step process. First the discriminator is evaluated to find the discriminant. If it returns invalid, then the bytes do not encode a valid enum and the decoding fails with undefined behavior. After the discriminant has been figured out, the bytes can be decoded as the variant type. This step can still fail, for example when only data of an enum has been modified and is now invalid.

Now that the enum type and its representation relation have been defined, the operations can be defined as well. They will closely follow the operations laid out in the MIR.

3.1.6 Variant constructor

An enum value can be constructed using `ValueExpr::Variant`, mirroring the MIR `RValue::Aggregate` for enums. It takes a discriminant, the data for the variant value and the type of the enum. Why is the type needed when the previous chapter defined an enum value as only its discriminant and value of the variant paired together? Evaluating a MiniRust `ValueExpr` returns not only the value but its type too.

3.1.7 GetDiscriminant and SetDiscriminant

Getting the discriminant is done by using `ValueExpr::GetDiscriminant`, which evaluates the discriminator of the enum on the `PlaceExpr` it was given. This currently only requires that the bytes, which are read for the discriminant, are valid. Please note that the validity requirements for reading the discriminant on its own are still undecided[13], for a detailed discussion on this see chapter 4.1.

Similarly `Statement::SetDiscriminant` only runs the tagger on the given `PlaceExpr` for the specified discriminant. It does not enforce any validity requirements at all, as it can be used to produce a valid enum value from scratch as well. However `ValueExpr::Variant` should be used for this. `Statement::SetDiscriminant` mainly exists because of the MIR statement described in chapter 2.1.1.

3.1.8 Variant Downcast

In order to access data inside an enum value, MiniRust needs to know which variant should be accessed. `PlaceExpr::Downcast` takes a `PlaceExpr` of the enum value and the discriminant of the variant into which the enum is downcast.

Evaluating a `Downcast` makes it evaluate its inner `PlaceExpr` and return the same pointer combined with the variant type. As mentioned before, this variant type might be a tuple holding the fields. On the tuple a field access can be done the same way as the Rust MIR does on the enum type with a known variant. Therefore no additional logic to access the fields of a variant has to be added. However, a field access on variant types that are not a tuple does not have to be well-defined. Variant types will be discussed in more detail in chapter 4.2.

3.1.9 Switch terminator

Previously, MiniRust only supported `If` terminators that branch on a boolean. But with the introduction of enums a `Switch` terminator was introduced that branches on integers. The MiniRust specific `If` terminator was removed in favor of it. The old behavior is achieved with a boolean to int cast and switching on the resulting value.

The `Switch` terminator evaluates a `ValueExpr` and tries to find the target for the value in its case map. If no target was found, then the terminator jumps to a designated fallback. This mirrors the behavior of the MIR `SwitchInt` terminator described in chapter 2.1.1.

3.2. Minimizer

The following two sections describe the implementations necessary for converting Rust code containing enums to MiniRust. Most of the translation is trivial, as MiniRust is modeling Rust MIR. But the way Rust's compiler rustc stores types makes the enum type translation a bit harder.

3.2.1 Enum type minimization

For the variant types, `minimize` creates a tuple type with all the fields and the same size and alignment as the total enum, as hinted at earlier. Chapter 4.2 gives a more formal reasoning about the validity of this. This leaves the tag encoding from chapter 2.1.3 to be translated. Recall that Rust uses two different tag encodings at the time of this thesis, direct and niche tag encoding.

For direct tag encoding the translation is trivial. The tagger for each variant writes its discriminant in the tag field. The discriminator in turn reads the tag field to determine the discriminant. Its fallback is `Discriminator::Invalid`, as all variants have an entry. Note the Rust compiler ensuring that the tag field is part of the enum and not the variants, and is ensured to be separate from all the data.

For niche tag encoding the translation gets more involved. Recall that one variant, called the untagged variant, has data at the tag field and the other variants write a value that is invalid for that data. The tagger of the untagged variant does nothing. For the other variants, the tag is computed during minimization and the tagger is configured to write it into the field of the tag.

For the discriminator, the tag encoding stores which values are valid for the field of the tag, including all the valid values of the data in the untagged variant. This makes it harder to figure out which values are from data and which are from tagged variants. So the discriminator is set up to branch as follows: It maps the computed tags to the tagged variants. Furthermore it maps all values outside the valid range to `Discriminator::Invalid`. That leaves all data values, for which the fallback points to the discriminant of the untagged variant, as shown in figure 3.2.

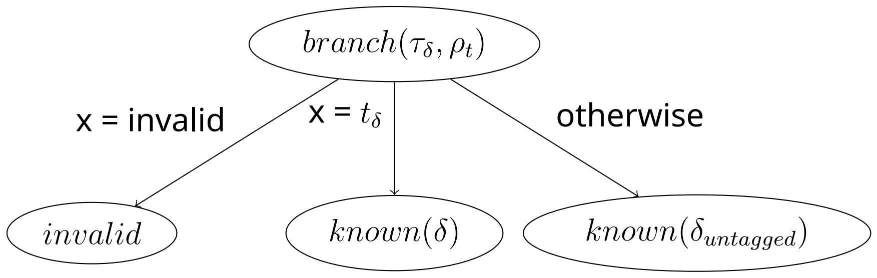


Figure 3.2: Niche encoding discriminator.
τ_δ: discriminant type, ρ_t: tag field offset, t_δ: tag for discriminant δ

Note that in the end, taggers produced by `minimize` have zero or one entry and discriminators have only one top-level branch. This might make the discriminator and tagger design seem overgeneralized but, since the Rust memory representation is unstable[5], this feels like a good precaution.

3.2.2 Further changes and implementations

The translation of the following MIR is a trivial one-to-one mapping:

- `RValue::Aggregate` of enums
  $\rightarrow$ `ValueExpr::Variant`
- `RValue::Discriminant`
  $\rightarrow$ `ValueExpr::GetDiscriminant`
- `StatementKind::SetDiscriminant`
  $\rightarrow$ `Statement::SetDiscriminant`
- `ProjectionKind::Downcast`
  $\rightarrow$ `PlaceExpr::Downcast`

Furthermore the Never type `!` is translated into a zero-variant enum. The only difference between the two types is the type coercion, as discussed in chapter 2.1.2. MiniRust has no concept of type coercion at the moment.

And lastly, the translation of the `SwitchInt` terminator has been expanded to support switching on integers and not only booleans, since switching on a discriminant is now possible. The only remaining type to switch on that is unsupported is `char`, which is not implemented in MiniRust at all.

4. Evaluation

For every feature implemented, unit tests were added to ensure that this feature works as intended. Unit tests for the specification look similar to the one in listing 13, which ensures MiniRust detecting the discriminator returning a discriminant, for which there is no variant.

This test shows various helper methods that provide a build interface for MiniRust programs to make them somewhat human-readable. For a simple program, with only a main function and no branches, only the local variables and statements of the function are required. The program is checked for well-formedness before being run, which in this case fails at validating the local variable with the enum type upon checking the type for well-formedness.

For `minimize`, the integration tests are given as rust programs and their output. The output is given in separate files for standard out and standard error. An example program for a test, that checks that iterators now work, can be seen in listing 14.

Running the `minimize` test will translate the program to MiniRust, run it and then compare standard out and standard error with the content of their respective file. If one or both of the files does not exist, then there should be no output for that stream. The program in listing 14 should print 162 followed by a newline character to standard out, which would be the content of the file for standard out. The file for standard error does not have to exist, since this program should output no warnings or errors.

```rust
#[test]
fn ill_formed_discriminator() {
    let enum_ty = enum_ty::<u8>( // Type of discriminant
        &[],                     // Variants, here none
        discriminator_known(1),  // Discriminator:
        size(0),                 // known(1) is invalid since
        align(1)                 // no variant has discr. 1.
    );
    let locals = &[enum_ty];
    let stmts = &[];
    let prog = small_program(locals, stmts);
    assert_ill_formed(prog);
}
```

Listing 13: MiniRust unit test to ensure that the discriminator cannot reach discriminants which have no variant.

```rust
struct RepeatN {
    val: bool,
    repetitions: u8,
}

impl Iterator for RepeatN {
    type Item = bool;
    fn next(&mut self) -> Option<Self::Item> {
        ..
    }
}

fn main() {
    let iter = RepeatN { val: true, repetitions: 3 };
    let mut sum = 0;
    for i in iter {
        sum += i;
    }
    print(sum);
}
```

Listing 14: An example `minimize` test program.

The next subchapters will go over some formal aspects around enums and the issues that arise from them.

4.1. GetDiscriminant validity

It was mentioned that the requirements for a valid discriminant read are still being discussed by the Rust team[13]. The debate is inconclusive since the requirements used in the Rust compiler depend on what it is doing. Nevertheless, here are explanations for two positions to show the problem of finding suitable validity requirements.

One could require the entire enum value to be valid. But this is not compatible with enums that have a deconstructor for some variants. It might happen that data got moved out of a variant field. The bytes of this field would no longer be guaranteed to be initialized to some valid value. But then, if the enum needs to be deconstructed for some variant, it would need to check whether that variant is present in the enum value. This is currently done with a discriminant read on the now invalid value.

Another way would be to do what MiniRust currently does: Require the bytes, which are read to determine the discriminant, to be initialized and encode some valid discriminant. But some optimizations and the code generation in the Rust compiler rely on the fact that, when the discriminant can be read, then the entire enum is valid. For example, a switch on the discriminant is often considered enough to ensure that the entire enum value is valid, and therefore its fields can be safely read from and written to.

In conclusion the validity requirements used in the compiler seem to contradict each other at the moment, picking whatever is the most useful for the specific job. Luckily for MiniRust, this does not really matter, as MiniRust is able to detect undefined behavior and just aborts when it occurs.

With this out of the way, the next two subchapters deal with the generalization of behavior in MiniRust and why it is valid.

4.2. Using variant types instead of fields

The sections about the implementation should already have hinted on how using tuples for the variant types yields the same behavior as the fields in Rust enum variants. This subchapter aims to give a more formal reasoning for why this is equivalent, and then show how arbitrary variant types are more general, in the sense that MiniRust can handle more semantics than Rust.

To show this, let us first consider using tuples for the variant types. This is equivalent to fields when it comes to Rust MIR semantics since the operations that can be done on Rust enums are the following:

Reading or writing the discriminant. Which does not concern itself with how the data is structured. All it cares about is where in the bytes the data is stored, so as not to overwrite it, and what values the encoded data can assume for a niche tag encoding.

Further it should be clear from the `minimize` implementation in chapter 3.2.1 that every current Rust tag encoding can be expressed as discriminators and taggers.

Constructing a variant given the fields. This can be mapped to MiniRust by first constructing a tuple with the fields and then constructing a variant using this tuple.

Projecting to a field in a variant. For this to be allowed in the MIR, the variant has to be known. While a comment in the Rust compiler documentation[14] mentions that a single-variant enum would not have to be downcast, since the variant is always known, the compiler in practice still does a downcast. This is probably because the downcast is a pure reinterpretation of the layout and does not generate code. Therefore it is easier to just not treat this special case separate, as there is no loss of performance in the compiled program. For the purposes of this thesis, this case is ignored. It is very likely that a field access without downcast will never occur, since very few enums have only one variant and the only benefit would be a smaller MIR, easily offset by the amount of work to treat this special case.

In MiniRust the downcast would yield the place with the tuple type on which the field access is well-defined in a semantically equivalent way.

Downcasting an enum value into a variant. In Rust this only happens if it is followed by a field projection. Furthermore, it is not well-formed MIR to downcast when a previous downcast projection already determined the variant. This means that in Rust a downcast leads to an in-between type that cannot be treated like the whole enum and essentially only allows a field projection on it.

In MiniRust, however, this gives the variant type which is well-defined and allows field projections in the case of a tuple as well as the usual projections on the variant type.

In conclusion, using tuples as variant types covers the entire range and behavior of Rust enum variants in a semantically equivalent manner. Furthermore, it is obvious that a wider range of enum variant semantics can be represented. A variant with variant type `bool` is not the same as using a single-field tuple containing the boolean. The `bool` does not need a field access after the downcast to read the value, as it is already a boolean. This means that MiniRust has well-defined behavior for a wider range of variants than Rust but covers Rust's semantics completely.

4.3. Additional representable enums in MiniRust

MiniRust can not only handle more semantics when it comes to variant types, the generalization of tagger and discriminator allow additional memory representations. Consider the enum in listing 15. One byte is not enough to store the discriminants of all 500 variants, at least 9 bits are required. Because of the alignment of `u16` there is a byte of padding after each of the `u8`[1].

[1]Again, according to the output of the Rust compiler at the time of this thesis.

```
enum TwoTags {
    Data(u8, u16, u8, u16),
    Empty1,
    Empty2,
    ..,
    Empty499
}
```

Listing 15: Enum which can be represented smaller in MiniRust than in Rust

The two bytes can be used to store the tags of the 500 variants by writing the upper byte of the discriminant in the first byte of padding and the lower into the second. The discriminator can deal with such an enum by branching first on the first byte of padding and then on the lower.

Rust would have to add another 16 bits to the size of the enum as the tag currently needs to be stored in a single field. Obviously, the discriminator and tagger could be set up to do so as well. In conclusion, MiniRust can represent enums in a wider range of representations, some of which use less space than what Rust currently can do. But there is nothing stopping the Rust compiler team to add further tag encodings and cover such cases.

4.4. Minimize

This subchapter discusses the two main issues that have arisen while writing integration tests for `minimize`.

First, `minimize` currently does not "remember" whether it compiled a specific type already. Since fields between enum variants are allowed to overlap, it is possible to write an enum of compact size whose representation gets huge, if the variant types are not reused. Listing 16 gives an example of such an enum constructed using macros where each nested enum is present in four variants. `Minimize` will compute every inner enum four times which after 27 iterations means that $4^{27} = 2^{54}$ enum types have been translated. Using only one byte for each enum type, which is obviously too little, this would result in around 18 Petabytes of data. If the types were reused it would only be 26 enum translations.

```rust
fn test_big_enum() {
    macro_rules! fooN {
        ($cur:ident $prev:ty) => {
            enum $cur {
                Empty,
                First($prev),
                Second($prev),
                Third($prev),
                Fourth($prev),
            }
        }
    }

    fooN!(Foo0 ());
    fooN!(Foo1 Foo0);
    ..
    fooN!(Foo27 Foo26);

    let _foo = Foo27::Empty;
}
```

Listing 16: How to create a huge enum that `minimize` cannot translate on a normal computer.

The second issue that arose seems to only concern `Option<NonZeroINT>` where `NonZeroINT` can be any of the integer types, for example `NonZeroU8`. Trying to construct a `None` variant of this type gets optimized into a constant scalar which contains the bytes of the value. The problem is that MiniRust currently does not support creating constant values from bytes directly. The closest thing would be to load the value from a global allocation, but this is not semantically equivalent. There is no loading from a pointer happening in Rust. There are three ways for `minimize` to deal with such a value that come to mind:

First it could try to decode the enum value from the bytes. This requires a lot of effort to make it work in `minimize` and a general rework of constant translation is probably needed.

Second it could move the bytes into a global allocation and load from this allocation instead of the constant value. This changes the semantics of the program in a nontrivial manner which makes the solution undesirable.

The third option is to just not support `Option<NonZeroINT>`. This is the case at the moment until some more experimentation reveals more types that run into this issue, or a more satisfactory solution is found.

The reason for this behavior remains unclear. It seems to happen during constant evaluation, but it is weird that it only seems to concern `Option<NonZeroINT>`. All other tested types use a `RValue::Aggregate` even for constants.

5. Conclusion

This thesis implements enums in MiniRust in a way that completely covers the range of possible enum behavior in Rust. The generalization of discriminant encoding and decoding ensures that when Rust implements further optimizations on this topic, then only `minimize` needs to be updated to support this new behavior.

Furthermore, `minimize` has been extended to support transforming most Rust enums with two notable exceptions mentioned in the previous chapter about the performance of `minimize`. However those only affect niche cases, almost all real-world use cases are now covered. This unlocks many powerful tools that the Rust standard library includes, like for example iterators.

The implementation was contributed over multiple pull requests to the MiniRust repository:

- Enum variant `ValueExpr`
- Enum memory representation and well-formedness
- Enum `Downcast`
- Enum `GetDiscriminant` and `SetDiscriminant`
- Switch terminator
- Alignment-check fixes (mainly around enums)
- Changes to discriminant handling
- `Minimize` support for everything enum related but niche tag encoding
- `Minimize` support for niche tag encoded enums

In the end little work is still required around enums, and a lot of work around MiniRust in general.

5.1. Future Work

Starting with the work required around enums, the enum constant scalars for `Option<NonZeroINT>::None` need some way of translation. Some ideas on how were given in chapter 4.4, but they do not seem practical. Next, the type translation in `minimize` could be optimized to reuse results. And finally, it has been mentioned in chapter 3.1.5 about the representation relation that a well-formedness check for the tagger not overwriting data might make sense in future.

Furthermore there are still a lot of small parts missing in MiniRust and `minimize` which are required for a complete representation of Rust. For example `String`s and `char` are not implemented at all. Minimizing `for i in 0..42 {}` fails only because of the constant range. Implementing boolean to integer casting and negation was implemented concurrently to the work on this thesis. Small things like this can be found everywhere in the MiniRust project.

And finally a possible extension goal for this thesis was to make the coding interface for MiniRust test cases nicer. At the moment every reference to local variables is done by giving the index in the array of local variables defined earlier, which makes it easy to make a mistake. Also, well-formedness does not tell the programmer where the mistake is, only that the entire program is not well-formed.

Bibliography

[1] The Rust Team. Rust, . URL https://www.rust-lang.org/. [Accessed on February 8, 2024].

[2] Ferrous Systems. Ferrocene, . URL https://ferrous-systems.com/ferrocene/. [Accessed on March 7, 2024].

[3] Ralf Jung and contributors. Minirust. URL https://github.com/minirust/minirust. [Accessed on February 8, 2024].

[4] The Rust Team. Never type, . URL https://doc.rust-lang.org/stable/reference/types/never.html. [Accessed on March 13, 2024].

[5] The Rust Team. Type layout, . URL https://doc.rust-lang.org/stable/reference/type-layout.html. [Accessed on March 13, 2024].

[6] The Rust Team. The rustonomicon, . URL https://doc.rust-lang.org/nomicon/. [Accessed on March 7, 2024].

[7] The Rust Release Team. Announcing rust 1.68.0, . URL https://blog.rust-lang.org/2023/03/09/Rust-1.68.0.html. [Accessed on March 10, 2024].

[8] Ferrous Systems. Ferrocene language specification, . URL https://public-docs.ferrocene.dev/main/specification/. [Accessed on March 10, 2024].

[9] The Rust Team. The rust reference, . URL https://doc.rust-lang.org/stable/reference/. [Accessed on March 7, 2024].

[10] Ferrous Systems. Ferrocene language specification, enums, . URL https://public-docs.ferrocene.dev/main/specification/types-and-traits.html#enum-types. [Accessed on March 17, 2024].

[11] The Rust Team. Enumerations, . URL https://doc.rust-lang.org/stable/reference/items/enumerations.html. [Accessed on March 13, 2024].

[12] Jakob Degen. Enum layout and discriminant, 2022. URL https://hackmd.io/@2S4Crel_Q9OwC_vamlwXmw/By4FoVud9. [Accessed on February 13, 2024].

[13] Jakob Degen Ralf Jung and contributors. Decide on when mir discriminant() operation is ub. URL `https://github.com/rust-lang/rust/issues/91095`. [Accessed on March 10, 2024].

[14] The Rust Team. Place in rustc-middle::mir::syntax, . URL `https://doc.rust-lang.org/stable/nightly-rustc/rustc_middle/mir/syntax/struct.Place.html`. [Accessed on March 12, 2024].