Implementing Enums in MiniRust

Bachelor Thesis Project Description

Timon Meyer supervised by Prof. Dr. Ralf Jung

Department of Computer Science, ETH Zürich
Zürich, Switzerland

The MiniRust[1] project tries to specify the operational semantics of the Rust programming language[2] by giving an interpreter that will produce the same behaviours as the Rust program would. The object language of the interpreter is a subset of Rust that resembles the middle intermediate representation (MIR) used by the Rust compiler. The interpreter itself is described in a language called specr designed for this purpose.

The goal of this thesis is to add support for enums in MiniRust. Enums are rusts version of sum types implemented as tagged unions. They use a discriminant to describe which variant of the union is stored in a value representing the enum.

To ensure type safety, the Rust compiler only allows access to fields of a certain variant by a process called pattern matching. In the following example pattern matching is used both in the `if` condition, the `let-else` and the `match` statement to access the field of a possible `A::Variant1`.

```rust
enum A {
    Variant1(NonZeroU64),
    Variant2
}

fn foo(a: A) -> u64 {
    if let A::Variant1(v) = a {
        return v as u64;
    }

    let A::Variant1(v) = a else {
        return 0;
    }

    match a {
        A::Variant1(v) => v as u64,
        A::Variant2 => 0
    }
}
```

Listing 1: Example on how to use enums

Under the hood they all do roughly the same: They compute the discriminant and then execute the according block.

Adding a tag as the discriminant, that has to be large enough to represent every variant might be very space inefficient, especially when working with `Option`s containing pointers where we would add 7 bytes of padding for alignment. (We have 8 bytes for the pointer and at least 1 bit for the tag). To combat

[1] https://github.com/RalfJung/minirust
[2] https://www.rust-lang.org/

this Rust tries to place the discriminant in the biggest so called ‘niche‘, which is a range of invalid values for a subset of bits which are used as discriminant. In Listing 1 `A` has a niche at value 0, which could be used to represent `Variant2` and therefore reduce the enum size to 8 bytes. The Rust compiler `rustc`[3] uses various layout optimizations to create and merge niches and therefore reduce the enum size.

This however may make computing the discriminant more complex. Since the exact method is only required when generating the final code the Rust MIR abstracts this operation away. The match statement from Listing 1 would be close to the following:

```
let mut _0: u64; // return value

bb0: {
    // the discriminant was set using ‘discriminant(a) = ...;‘
    _1 = discriminant(a);
    switchInt(move _1) -> [0: bb2, otherwise: bb1];
}

bb1: {
    _2 = ((x as Variant1).0: NonZeroU64);
    _0 = NonZeroU64::get(move _2) -> bb3;
}

bb2: {
    _0 = const 0_u64;
    goto -> bb3;
}

bb3: {
    return;
}
```

Listing 2: Example MIR output of match

While MiniRust also doesn’t care about the exact way Rust computes the discriminant because that is not part of the Rust specification, it still needs to find it for the interpreter and cares about the data layout of the enum. This is where a very general decision tree model by Jakob Degen[4] would fit nicely into the picture as it allows us to define the way to find the discriminant from a layout perspective and not instruction-level perspective.

This leads us to the core goals of this bachelor thesis:

- Implement enum encoding and decoding in MiniRust.
  For the discriminant evaluate both the decision tree model and the way the niche model[5] is implemented in `rustc` for discrimination (i.e. `switchInt`).
- Implement enum discriminant projection to set/get the discriminant.
- Implement enum field projection. We somehow need to access the fields, so MiniRust needs a way to project the fields on new variables. It might make sense to first implement variant projection like the Rust MIR as a `PlaceExpr` which then returns a struct. Mind that those variant downcasts only exist in the MIR as a result of pattern matching and cannot be used in the base language.

[3] https://github.com/rust-lang/rust/tree/master/compiler
[4] https://hackmd.io/@2S4Crel_Q9OwC_vamlwXmw/By4FoVud9
[5] https://github.com/camelid/type-layout

- Find out what is considered undefined behaviour (UB) by the Rust community and compiler and build test cases around those findings.

Should the time still allow the following extension goals are available:

- Add support for enums in the minimizer.

  The minimizer is a transpiler from Rust to MiniRust. We can directly take the layout computed by Rust and convert it to the MiniRust representation.
- Make MiniRust programs for test purposes easier to write by hand. Currently, writing a MiniRust program by hand (i.e., not using Minimize) is quite cumbersome since a lot of entities (local variables, basic blocks, functions) are identified by an ID (a natural number), and one has to carefully use the correct number everywhere. It is easy to pick the wrong number which leads to confusing results. Since we are already often using helper methods to generate these entities we should be able to make them also return a generated ID which we can store in properly named variables for later reference without needing to write the bare ID.