Miri:
An interpreter for Rust's mid-level intermediate representation

Scott Olson*
Supervised by Christopher Dutchyn

April 12th, 2016

1 Abstract

The increasing need for safe low-level code in contexts like operating systems and browsers is driving the development of Rust[1], a programming language promising high performance without the risk of memory unsafety. To make programming more convenient, it's often desirable to be able to generate code or perform some computation at compile-time. The former is mostly covered by Rust's existing macro feature or build-time code generation, but the latter is currently restricted to a limited form of constant evaluation capable of little beyond simple math.

The architecture of the compiler at the time the existing constant evaluator was built limited its potential for future extension. However, a new intermediate representation was recently added[2] to the Rust compiler between the abstract syntax tree and the back-end LLVM IR, called mid-level intermediate representation, or MIR for short. This report will demonstrate that writing an interpreter for MIR is a surprisingly effective approach for supporting a large proportion of Rust's features in compile-time execution.

2 Background

The Rust compiler generates an instance of `Mir` for each function [Figure 1]. Each `Mir` structure represents a control-flow graph for a given function, and contains a list of "basic blocks" which in turn contain a list of statements followed by a single terminator. Each statement is of the form `lvalue = rvalue`. An `Lvalue` is used for referencing variables and calculating addresses such as when dereferencing pointers, accessing fields, or indexing arrays. An `Rvalue` represents the core set of operations possible in MIR, including reading a value from an lvalue, performing math operations, creating new pointers, structures, and arrays, and so on. Finally, a terminator decides where control will flow next, optionally based on the value of a boolean or integer.

3 First implementation

3.1 Basic operation

To investigate the possibility of executing Rust at compile-time I wrote an interpreter for MIR called Miri[3]. The structure of the interpreter closely mirrors the structure of MIR itself. It starts executing a function by iterating the statement list in the starting basic block, translating the lvalue into a pointer and using the rvalue to decide what to write into that pointer. Evaluating the rvalue may involve reads (such as for the two sides of a binary operation) or construction of new values. When the terminator is reached, it is used to decide which basic block to jump to next. Finally, Miri repeats this entire process, reading statements from the new block.

* scott@solson.me
[1] https://www.rust-lang.org
[2] Rust RFC #1211: Mid-level IR (MIR)
[3] https://github.com/tsion/miri

```
struct Mir {
    basic_blocks: Vec<BasicBlockData>,
    // ...
}

struct BasicBlockData {
    statements: Vec<Statement>,
    terminator: Terminator,
    // ...
}

struct Statement {
    lvalue: Lvalue,
    rvalue: Rvalue
}

enum Terminator {
    Goto { target: BasicBlock },
    If {
        cond: Operand,
        targets: [BasicBlock; 2]
    },
    // ...
}
```

Figure 1: MIR (simplified)

3.2 Function calls

To handle function call terminators[4], Miri is required to store some information in a virtual call stack so that it may pick up where it left off when the callee returns. Each stack frame stores a reference to the `Mir` for the function being executed, its local variables, its return value location[5], and the basic block where execution should resume. When Miri encounters a `Return` terminator in the MIR, it pops one frame off the stack and resumes the previous function. Miri's execution ends when the function it was initially invoked with returns, leaving the call stack empty.

It should be noted that Miri does not itself recurse when a function is called; it merely pushes a virtual stack frame and jumps to the top of the interpreter loop. Consequently, Miri can interpret deeply recursive programs without overflowing its native call stack. This approach would allow Miri to set a virtual stack depth limit and report an error when a program exceeds it.

3.3 Flaws

This version of Miri supported quite a bit of the Rust language, including booleans, integers, if-conditions, while-loops, structures, enums, arrays, tuples, pointers, and function calls, requiring approximately 400 lines of Rust code. However, it had a particularly naive value representation with a number of downsides. It resembled the data layout of a dynamic language like Ruby or Python, where every value has the same size[6] in the interpreter:

```
enum Value {
    Uninitialized,
    Bool(bool),
    Int(i64),
    Pointer(Pointer), // index into stack
    Aggregate {
      variant: usize,
      data: Pointer,
    },
}
```

This representation did not work well for aggregate types[7] and required strange hacks to support them. Their contained values were allocated elsewhere on the stack and pointed to by the aggregate value, which made it more complicated to implement copying aggregate values from place to place.

Moreover, while the aggregate issues could be worked around, this value representation made common unsafe programming tricks (which make assumptions about the low-level value layout) fundamentally impossible.

[4] Calls occur only as terminators, never as rvalues.

[5] Return value pointers are passed in by callers.

[6] An `enum` is a discriminated union with a tag and space to fit the largest variant, regardless of which variant it contains.

[7] That is, structures, enums, arrays, tuples, and closures.

4 Current implementation

Roughly halfway through my time working on Miri, Eduard Burtescu[8] from the Rust compiler team[9] made a post on Rust's internal forums about a "Rust Abstract Machine" specification[10] which could be used to implement more powerful compile-time function execution, similar to what is supported by C++14's `constexpr` feature. After clarifying some of the details of the data layout with Burtescu via IRC, I started implementing it in Miri.

4.1 Raw value representation

The main difference in the new value representation was to represent values by "abstract allocations" containing arrays of raw bytes with different sizes depending on their types. This mimics how Rust values are represented when compiled for physical machines. In addition to the raw bytes, allocations carry information about pointers and undefined bytes.

```
struct Memory {
    map: HashMap<AllocId, Allocation>,
    next_id: AllocId,
}

struct Allocation {
    bytes: Vec<u8>,
    relocations: BTreeMap<usize, AllocId>,
    undef_mask: UndefMask,
}
```

4.1.1 Relocations

The abstract machine represents pointers through "relocations", which are analogous to relocations in linkers[11]. Instead of storing a global memory address in the raw byte representation like on a physical machine, we store an offset from the start of the target allocation and add an entry to the relocation table which maps the index of the offset bytes to the target allocation.

In Figure 2, the relocation stored at offset 0 in `y` points to offset 2 in `x` (the 2nd 16-bit integer). Thus, the relocation table for `y` is `{0 => x}`, meaning the next N bytes after offset 0 denote an offset into allocation `x` where N is the size of a pointer (4 in this example). The example shows this as a labelled line beneath the offset bytes.

In effect, the abstract machine represents pointers as `(allocation_id, offset)` pairs. This makes it easy to detect when pointer accesses go out of bounds.

```
let x: [i16; 3] = [0xAABB, 0xCCDD, 0xEEFF];
let y = &x[1];
// x: BB AA DD CC FF EE (6 bytes)
// y: 02 00 00 00 (4 bytes)
//    └──(x)──┘
```

Figure 2: Example relocation on 32-bit little-endian

4.1.2 Undefined byte mask

The final piece of an abstract allocation is the undefined byte mask. Logically, we store a boolean for the definedness of every byte in the allocation, but there are multiple ways to make the storage more compact. I tried two implementations: one based on the endpoints of alternating ranges of defined and undefined bytes and the other based on a bitmask. The former is more compact but I found it surprisingly difficult to update cleanly. I currently use the much simpler bitmask system.

See Figure 3 for an example of an undefined byte in a value, represented by underscores. Note that there is a value for the second byte in the byte array, but it doesn't matter what it is. The bitmask would be 10_2, i.e. `[true, false]`.

```
let x: [u8; 2] = unsafe {
    [1, std::mem::uninitialized()]
};
// x: 01 __ (2 bytes)
```

Figure 3: Example undefined byte

[8] eddyb on GitHub
[9] https://www.rust-lang.org/team.html#Compiler
[10] Burtescu's reply on "MIR constant evaluation"
[11] Relocation (computing) - Wikipedia

4.2 Computing data layout

Currently, the Rust compiler's data layouts for types are hidden from Miri, so it does its own data layout computation which will not always match what the compiler does, since Miri doesn't take target type alignments into account. In the future, the Rust compiler may be modified so that Miri can use the exact same data layout.

Miri's data layout calculation is a relatively simple transformation from Rust types to a structure with constant size values for primitives and sets of fields with offsets for aggregate types. These layouts are cached for performance.

5 Deterministic execution

In order to be effective as a compile-time evaluator, Miri must have *deterministic execution*, as explained by Burtescu in the "Rust Abstract Machine" post. That is, given a function and arguments to that function, Miri should always produce identical results. This is important for coherence in the type checker when constant evaluations are involved in types, such as for sizes of array types:

```
const fn get_size() -> usize { /* ... */ }
let array: [i32; get_size()];
```

Since Miri allows execution of unsafe code[12], it is specifically designed to remain safe while interpreting potentially unsafe code. When Miri encounters an unrecoverable error, it reports it via the Rust compiler's usual error reporting mechanism, pointing to the part of the original code where the error occurred. Below is an example from Miri's repository.[13]

```
let b = Box::new(42);
let p: *const i32 = &*b;
drop(b);
unsafe { *p }
//       ~~ error: dangling pointer
//            was dereferenced
```

[12] In fact, the distinction between safe and unsafe doesn't exist at the MIR level.

[13] miri/test/errors.rs

6 Language support

In its current state, Miri supports a large proportion of the Rust language, detailed below. The major exception is a lack of support for FFI[14], which eliminates possibilities like reading and writing files, user input, graphics, and more. However, for compile-time evaluation in Rust, this limitation is desired.

6.1 Primitives

Miri supports booleans, integers of various sizes and signed-ness (i.e. `i8`, `i16`, `i32`, `i64`, `isize`, `u8`, `u16`, `u32`, `u64`, `usize`), and unary and binary operations over these types. The `isize` and `usize` types will be sized according to the target machine's pointer size just like in compiled Rust. The `char` and float types (`f32`, `f64`) are not supported yet, but there are no known barriers to doing so.

When examining a boolean in an `if` condition, Miri will report an error if its byte representation is not precisely 0 or 1, since having any other value for a boolean is undefined behaviour in Rust. The `char` type will have similar restrictions once it is implemented.

6.2 Pointers

Both references and raw pointers are supported, with essentially no difference between them in Miri. It is also possible to do pointer comparisons and math. However, a few operations are considered errors and a few require special support.

Firstly, pointers into the same allocations may be compared for ordering, but pointers into different allocations are considered unordered and Miri will complain if you attempt this. The reasoning is that different allocations may have different orderings in the global address space at runtime, making this non-deterministic. However, pointers into different allocations *may* be compared for direct equality (they are always unequal).

Secondly, pointers represented using relocations may be compared against pointers casted from in-

[14] Foreign Function Interface, e.g. calling functions defined in Assembly, C, or C++.

tegers (e.g. `0 as *const i32`) for things like null pointer checks. To handle these cases, Miri has a concept of "integer pointers" which are always unequal to abstract pointers. Integer pointers can be compared and operated upon freely. However, note that it is impossible to go from an integer pointer to an abstract pointer backed by a relocation. It is not valid to dereference an integer pointer.

6.2.1 Slice pointers

Rust supports pointers to "dynamically-sized types" such as `[T]` and `str` which represent arrays of indeterminate size. Pointers to such types contain an address *and* the length of the referenced array. Miri supports these fully.

6.2.2 Trait objects

Rust also supports pointers to "trait objects" which represent some type that implements a trait, with the specific type unknown at compile-time. These are implemented using virtual dispatch with a vtable, similar to virtual methods in C++. Miri does not currently support these at all.

6.3 Aggregates

Aggregates include types declared with `struct` or `enum` as well as tuples, arrays, and closures. Miri supports all common usage of all of these types. The main missing piece is to handle `#[repr(..)]` annotations which adjust the layout of a `struct` or `enum`.

6.4 Lvalue projections

This category includes field accesses, dereferencing, accessing data in an `enum` variant, and indexing arrays. Miri supports all of these, including nested projections such as `*foo.bar[2]`.

6.5 Control flow

All of Rust's standard control flow features, including `loop`, `while`, `for`, `if`, `if let`, `while let`, `match`, `break`, `continue`, and `return` are supported. In fact, supporting these was quite easy since the Rust compiler reduces them all down to a small set of control-flow graph primitives in MIR.

6.6 Function calls

As previously described, Miri supports arbitrary function calls without growing the native stack (only its virtual call stack). It is somewhat limited by the fact that cross-crate[15] calls only work for functions whose MIR is stored in crate metadata. This is currently true for `const`, generic, and inline functions. A branch of the compiler could be made that stores MIR for all functions. This would be a non-issue for a compile-time evaluator based on Miri, since it would only call `const fn`s.

6.6.1 Method calls

Miri supports trait method calls, including invoking all the compiler-internal lookup needed to find the correct implementation of the method.

6.6.2 Closures

Calls to closures are also supported with the exception of one edge case[16]. The value part of a closure that holds the captured variables is handled as an aggregate and the function call part is mostly the same as a trait method call, but with the added complication that closures use a separate calling convention within the compiler.

6.6.3 Function pointers

Function pointers are not currently supported by Miri, but there is a relatively simple way they could be encoded using a relocation with a special reserved allocation identifier. The offset of the relocation would determine which function it points to in a special array of functions in the interpreter.

[15] A crate is a single Rust library (or executable).

[16] Calling a closure that takes a reference to its captures via a closure interface that passes the captures by value is not yet supported.

6.6.4 Intrinsics

To support unsafe code, and in particular to support Rust's standard library, it became clear that Miri would have to support calls to compiler intrinsics[17]. Intrinsics are function calls which cause the Rust compiler to produce special-purpose code instead of a regular function call. Miri simply recognizes intrinsic calls by their unique ABI[18] and name and runs special-purpose code to handle them.

An example of an important intrinsic is `size_of` which will cause Miri to write the size of the type in question to the return value location. The Rust standard library uses intrinsics heavily to implement various data structures, so this was a major step toward supporting them. Intrinsics have been implemented on a case-by-case basis as tests which required them were written, and not all intrinsics are supported yet.

6.6.5 Generic function calls

Miri needs special support for generic function calls since Rust is a *monomorphizing* compiler, meaning it generates a special version of each function for each distinct set of type parameters it gets called with. Since functions in MIR are still polymorphic, Miri has to do the same thing and substitute function type parameters into all types it encounters to get fully concrete, monomorphized types. For example, in…

```
fn some<T>(t: T) -> Option<T> { Some(t) }
```

…Miri needs to know the size of `T` to copy the right amount of bytes from the argument to the return value. If we call `some(10i32)` Miri will execute `some` knowing that `T = i32` and generate a representation for `Option<i32>`.

Miri currently does this monomorphization lazily on-demand unlike the Rust back-end which does it all ahead of time.

[17] https://doc.rust-lang.org/stable/std/intrinsics/index.html

[18] Application Binary Interface, which defines calling conventions. Includes "C", "Rust", and "rust-intrinsic".

6.7 Heap allocations

The next piece of the puzzle for supporting interesting programs (and the standard library) was heap allocations. There are two main interfaces for heap allocation in Rust: the built-in `Box` rvalue in MIR and a set of C ABI foreign functions including `__rust_allocate`, `__rust_reallocate`, and `__rust_deallocate`. These correspond approximately to `malloc`, `realloc`, and `free` in C.

The `Box` rvalue allocates enough space for a single value of a given type. This was easy to support in Miri. It simply creates a new abstract allocation in the same manner as for stack-allocated values, since there's no major difference between them in Miri.

The allocator functions, which are used to implement things like Rust's standard `Vec<T>` type, were a bit trickier. Rust declares them as `extern "C" fn` so that different allocator libraries can be linked in at the user's option. Since Miri doesn't actually support FFI and wants full control of allocations for safety, it "cheats" and recognizes these allocator functions in essentially the same way it recognizes compiler intrinsics. Then, a call to `__rust_allocate` simply creates another abstract allocation with the requested size and `__rust_reallocate` grows one.

In the future, Miri should also track which allocations came from `__rust_allocate` so it can reject reallocate or deallocate calls on stack allocations.

6.8 Destructors

When a value which "owns" some resource (like a heap allocation or file handle) goes out of scope, Rust inserts *drop glue* that calls the user-defined destructor for the type if it has one, and then drops all of the subfields. Destructors for types like `Box<T>` and `Vec<T>` deallocate heap memory.

Miri doesn't yet support calling user-defined destructors, but it has most of the machinery in place to do so already. There *is* support for dropping `Box<T>` types, including deallocating their associated allocations. This is enough to properly execute the dangling pointer example in section 5.

6.9 Constants

Only basic integer, boolean, string, and byte-string literals are currently supported. Evaluating more complicated constant expressions in their current form would be a somewhat pointless exercise for Miri. Instead, we should lower constant expressions to MIR so Miri can run them directly, which is precisely what would need be done to use Miri as the compiler's constant evaluator.

6.10 Static variables

Miri doesn't currently support statics, but they would need support similar to constants. Also note that while it would be invalid to write to static (i.e. global) variables in Miri executions, it would probably be fine to allow reads.

6.11 Standard library

Throughout the implementation of the above features, I often followed this process:

1. Try using a feature from the standard library.
2. See where Miri runs into stuff it can't handle.
3. Fix the problem.
4. Go to 1.

At present, Miri supports a number of major non-trivial features from the standard library along with tons of minor features. Smart pointer types such as `Box`, `Rc`[19] and `Arc`[20] all seem to work. I've also tested using the shared smart pointer types with `Cell` and `RefCell`[21] for internal mutability, and that works as well, although `RefCell` can't ever be borrowed twice until I implement destructor calls, since a destructor is what releases the borrow.

But the standard library collection I spent the most time on was `Vec`, the standard dynamically-growable array type, similar to C++'s `std::vector` or Java's `java.util.ArrayList`. In Rust, `Vec` is an extremely pervasive collection, so supporting it is a big win for supporting a larger swath of Rust programs in Miri.

```
struct Vec<T> {
    data: *mut T,    // 4 byte pointer
    capacity: usize, // 4 byte integer
    length: usize,   // 4 byte integer
}

let mut v: Vec<u8> =
    Vec::with_capacity(2);
// v: 00 00 00 00 02 00 00 00 00 00 00 00
//    └─(data)──┘
// data: __ __

v.push(1);
// v: 00 00 00 00 02 00 00 00 01 00 00 00
//    └─(data)──┘
// data: 01 __

v.push(2);
// v: 00 00 00 00 02 00 00 00 02 00 00 00
//    └─(data)──┘
// data: 01 02

v.push(3);
// v: 00 00 00 00 04 00 00 00 03 00 00 00
//    └─(data)──┘
// data: 01 02 03 __
```

Figure 4: `Vec` example on 32-bit little-endian

See Figure 4 for an example (working in Miri today) of initializing a `Vec` with a small amount of space on the heap and then pushing enough elements to force it to reallocate its data array. This involves cross-crate generic function calls, unsafe code using raw pointers, heap allocation, handling of uninitialized memory, compiler intrinsics, and more.

Miri supports unsafe operations on `Vec` like `v.set_len(10)` or `v.get_unchecked(2)`, provided that such calls do no invoke undefined behaviour. If a call *does* invoke undefined behaviour, Miri will abort with an appropriate error message (see Figure 5).

[19] Reference counted shared pointer
[20] Atomically reference-counted thread-safe shared pointer
[21] Rust documentation for cell types

```rust
fn out_of_bounds() -> u8 {
    let v = vec![1, 2];
    let p = unsafe { v.get_unchecked(5) };
    *p + 10
//  ~~ error: pointer offset outside
//       bounds of allocation
}

fn undefined_bytes() -> u8 {
    let v = Vec::<u8>::with_capacity(10);
    let p = unsafe { v.get_unchecked(5) };
    *p + 10
//  ~~~~~~~ error: attempted to read
//            undefined bytes
}
```

Figure 5: `Vec` examples with undefined behaviour

Here is one final code sample Miri can execute that demonstrates many features at once, including vectors, heap allocation, iterators, closures, raw pointers, and math:

```rust
let x: u8 = vec![1, 2, 3, 4]
    .into_iter()
    .map(|x| x * x)
    .fold(0, |x, y| x + y);
  // x: 1e (that is, the hex value
  //        0x1e = 30 = 1 + 4 + 9 + 16)
```

7 Future directions

7.1 Finishing the implementation

There are a number of pressing items on my to-do list for Miri, including:

- A much more comprehensive and automated test suite.
- User-defined destructor calls.
- Non-trivial casts between primitive types like integers and pointers.
- Handling statics and global memory.
- Reporting errors for all undefined behaviour.[22]
- Function pointers.
- Accounting for target machine primitive type alignment and endianness.
- Optimizations (undefined byte masks, tail-calls).
- Benchmarking Miri vs. unoptimized Rust.
- Various `TODO`s and `FIXME`s left in the code.
- Integrating into the compiler proper.

7.2 Future projects

Other possible Miri-related projects include:

- A read-eval-print-loop (REPL) for Rust, which may be easier to implement on top of Miri than the usual LLVM back-end.
- A graphical or text-mode debugger that steps through MIR execution one statement at a time, for figuring out why some compile-time execution is raising an error or simply learning how Rust works at a low level.
- A less restricted version of Miri that is able to run foreign functions from C/C++ and generally has full access to the operating system. Such an interpreter could be used to more quickly prototype changes to the Rust language that would otherwise require changes to the LLVM back-end.
- Unit-testing the compiler by comparing the results of Miri's execution against the results of LLVM-compiled machine code's execution. This would help to guarantee that compile-time execution works the same as runtime execution.
- Some kind of Miri-based symbolic evaluator that examines multiple possible code paths at once to determine if undefined behaviour could be observed on any of them.

[22] The Rust reference on what is considered undefined behaviour

8 Final thoughts

Writing an interpreter which models values of varying sizes, stack and heap allocation, unsafe memory operations, and more requires some unconventional techniques compared to conventional interpreters targeting dynamically-typed languages. However, aside from the somewhat complicated abstract memory model, making Miri work was primarily a software engineering problem, and not a particularly tricky one. This is a testament to MIR's suitability as an intermediate representation for Rust—removing enough unnecessary abstraction to keep it simple. For example, Miri doesn't even need to know that there are different kinds of loops, or how to match patterns in a `match` expression.

Another advantage to targeting MIR is that any new features at the syntax-level or type-level generally require little to no change in Miri. For example, when the new "question mark" syntax for error handling[23] was added to rustc, Miri required no change to support it. When specialization[24] was added, Miri supported it with just minor changes to trait method lookup.

Of course, Miri also has limitations. The inability to execute FFI and inline assembly reduces the amount of Rust programs Miri could ever execute. The good news is that in the constant evaluator, FFI can be stubbed out in cases where it makes sense, like I did with `__rust_allocate`. For a version of Miri not intended for constant evaluation, it may be possible to use libffi to call C functions from the interpreter.

In conclusion, Miri is a surprisingly effective project, and a lot of fun to implement. Due to MIR's tendency to collapse multiple source-level features into one, I often ended up supporting features I hadn't explicitly intended to. I am excited to work with the compiler team going forward to try to make Miri useful for constant evaluation in Rust.

[23] Question mark syntax RFC
[24] Specialization RFC

9 Thanks

A big thanks goes to Eduard Burtescu for writing the abstract machine specification and answering my incessant questions on IRC, to Niko Matsakis for coming up with the idea for Miri and supporting my desire to work with the Rust compiler, and to my research supervisor Christopher Dutchyn. Thanks also to everyone else on the compiler team and on Mozilla IRC who helped me figure stuff out. Finally, thanks to Daniel Keep and everyone else who helped fix my numerous writing mistakes.