wasmbin: a self-generating WebAssembly parser & serializer

rreverser.com/wasmbin-yet-another-webassembly-parser-serializer

October 17, 2020

If you prefer to skip right to the code, go here: https://github.com/GoogleChromeLabs/wasmbin

Backstory

Few months ago, I've been playing with various WebAssembly analysis and transformations in Rust and had a chance to try out several parsing / transformation / serialization toolchains.

As I was exploring this space, I've realised that what I really want is an API that would be high-level enough to allow me to work with the WebAssembly module AST as a natural Rust structure, yet low-level enough to preserve the binary contents of the module for any parts I don't modify in the process.

Here are some examples of toolchains I've tried and their pros/cons *for my particular goals*:

- wasmparser:
  - (+) Low-level access to the WebAssembly binary.
  - (+) Allows to lazily skip parts of the binary that you're not interested in parsing more deeply.
  - (+) Allows to parse individual items and not only module as a whole.
  - (+) Provides event-driven as well as structural API for reading.
  - (-) Allows parsing only from a byte slice, making the structural result hard to store / relocate without things like owning-ref.
  - (-) No support for serialization, making unsuitable for any transforms on its own.
  - (-) Some places could use newtypes a bit more actively (e.g. I'd love to see separate types for function ids / tables ids / etc., rather than all being generic numbers).
  - (-) The produced data structures lack standard basic trait implementations (e.g. PartialEq / Eq, Hash, etc.).

- walrus:
  - (+) Support for both parsing (based on wasmparser) and serialization.
  - (+) Very high-level IR, making most modifications easy without worrying about details. For example, one of the problems tricky to solve with lower-level APIs is when you add an import and suddenly all function IDs are shifted and all call instructions are pointing to wrong places - walrus takes care of this for you.
  - (+) All IDs are strongly typed, making them harder to mix up.
  - (+) Allows parsing from arbitrary sources, the result doesn't have a lifetime linked to the input and can be moved anywhere.
  - (+) Very well tested, has built-in debugging helpers like GraphViz output.
  - (-) Roundtrip is a non-goal and unsupported, entire module is rebuilt from IR upon serialization. On pre-optimised binaries, this also means that the result can be larger than the original, making it hard to use for writing custom optimisations.
  - (-) Due to the IR being very high-level, no standard trait implementations like Eq, Hash, Clone and so on, making it hard to store parsed items in other data structures.
  - (-) Working with function bodies can be sometimes awkward as they are represented in SSA-like form. You can't get access to the entire function body as a flat instruction list - instead, you're only given an entry instruction sequence, and a special visitor API that can walk from it to other connected instruction sequences. Similarly, on the other side of the API, there is a builder which you can use to build up instruction sequences and connect them together. This works well for many cases, but, for example, when I wanted to *just* make a copy of a function, I found it rather hard to make two APIs to correctly sync up with each other and gave up.
  - (-) No lazy parsing or a way to parse separate parts, it's all or nothing.

- parity-wasm
  - (+) Very close to a simple AST I wanted, with clear correspondence to the module structure.
  - (+) Support for both parsing and serialization.
  - (+) *(follows from two points above)* Preserves most of the module during round-trip.
  - (+) Provides most of the expected traits, others are easy to contribute.
  - (+) Allows parsing / serializing separate parts.
  - (+) Convenience helpers for common tasks (e.g. setting function as main, inserting section in a correct place etc.).
  - (+) Parsing from any source.
  - (-) No lazy parsing, entire module is parsed at the same time.
  - (-) Although gives access to "raw" representation of all fields, for some reason hides them behind function-based accessors, making them hard to use in pattern matching.
  - (-) Like in wasmparser, IDs are simple numbers rather than specialised newtypes.

Once again, I want to reiterate that since are my personal pet peeves that applied to the weird and fun use-cases I've been playing with, and by no means are indicative of how good one or another library is - all of them are successfully used by a lot of other people, and are likely to work for you too!

Let's build one more!

Now, one thing you should know about me (in case you haven't noticed from my blog yet) is that I'm a parsing nerd. So what these "downsides" did provide for me, was an excuse to try and build my own low-level parsing/serialization library for WebAssembly.

Moreover, as any parsing nerd, I **love** Rust macros. I've blogged about writing them before, I've used them in various projects, and generally consider them a powerful tool that can significantly increase readability and maintainability of code when used correctly.

So what I wanted to do for a while is to build a parser & serializer that would fully self-generate themselves using a macros. This way, I as a maintainer would only have to correctly describe all the structures and not bother with handwriting any implementations except for the core types. Think serde, but combining custom Deserialize & Deserializer for one specific format *(offtop: there are reasons why "just" using Serde wouldn't work here, but it's a bit out of scope of this post)*.

This brings us to custom derives. To recap, most of the Rust code out there relies on a powerful feature called `derive` - a special kind of macros that can be put on any user-defined type and used to auto-generate an accompanying implementation block.

For example, when you write something like:

```
#[derive(Debug, Hash, PartialEq, Eq)]
pub struct MyStruct {
    pub field: u32,
    pub another_field: bool,
    pub yet_another_field: Option<AnotherStruct>,
}
```

those `Hash` and `PartialEq` within the `derive` attribute get expanded into a code that recursively walks the structure fields and does the "right thing" to fold the results together:

```
impl ::core::hash::Hash for MyStruct {
    fn hash<__H: ::core::hash::Hasher>(&self, state: &mut __H) -> () {
        match *self {
            MyStruct {
            field: ref __self_0_0,
            another_field: ref __self_0_1,
            yet_another_field: ref __self_0_2 } => {
                ::core::hash::Hash::hash(&(*__self_0_0), state);
                ::core::hash::Hash::hash(&(*__self_0_1), state);
                ::core::hash::Hash::hash(&(*__self_0_2), state)
            }
        }
    }
}
...
impl ::core::cmp::PartialEq for MyStruct {
    fn eq(&self, other: &MyStruct) -> bool {
        match *other {
            MyStruct {
            field: ref __self_1_0,
            another_field: ref __self_1_1,
            yet_another_field: ref __self_1_2 } =>
            match *self {
                MyStruct {
                field: ref __self_0_0,
                another_field: ref __self_0_1,
                yet_another_field: ref __self_0_2 } =>
                (*__self_0_0) == (*__self_1_0) &&
                    (*__self_0_1) == (*__self_1_1) &&
                    (*__self_0_2) == (*__self_1_2),
            },
        }
    }
    ...
}
```

What if I could define all the necessary structures corresponding to the WebAssembly spec and put a custom derive in the end that would take care of generating the rest - parsing, serializing and visiting code?

```
#[derive(..., Wasmbin)]
```

And this is how Wasmbin was born. First of all, I've defined a couple of core traits:

```
pub trait Encode {
    fn encode(&self, w: &mut impl std::io::Write) -> std::io::Result<()>;
}

pub trait Decode: Sized {
    fn decode(r: &mut impl std::io::Read) -> Result<Self, DecodeError>;
}
```

`Encode` is for serializing the given object into an arbitrary write stream, and `Decode` is for reading it from an arbitrary read stream.

Then (not without a help of few other macros for templating) I've defined those traits on all necessary core types, according to the rules of the WebAssembly spec.

For example, the implementation for `u8`:

```
impl Encode for u8 {
    fn encode(&self, w: &mut impl std::io::Write) -> std::io::Result<()> {
        std::slice::from_ref(self).encode(w)
    }
}

impl Decode for u8 {
    fn decode(r: &mut impl std::io::Read) -> Result<Self, DecodeError> {
        let mut dest = 0;
        r.read_exact(std::slice::from_mut(&mut dest))?;
        Ok(dest)
    }
}
```

or integers:

```
macro_rules! def_integer {
    ($ty:ident, $leb128_method:ident) => {
        impl Encode for $ty {
            fn encode(&self, w: &mut impl std::io::Write) -> std::io::Result<()> {
                leb128::write::$leb128_method(w, (*self).into()).map(|_| ())
            }
        }

        impl Decode for $ty {
            fn decode(r: &mut impl std::io::Read) -> Result<Self, DecodeError> {
                const LIMIT: u64 = (std::mem::size_of::<$ty>() * 8 / 7) as u64 + 1;

                let mut r = std::io::Read::take(r, LIMIT);
                let as_64 = leb128::read::$leb128_method(&mut r)?;
                let res = Self::try_from(as_64)
                    .map_err(|_| DecodeError::Leb128(leb128::read::Error::Overflow))?;

                Ok(res)
            }
        }

        impl Visit for $ty {}
    };
}

def_integer!(u32, unsigned);
def_integer!(i32, signed);
def_integer!(u64, unsigned);
def_integer!(i64, signed);
```

Then I needed to do the same for container types. E.g. slices / vectors of "countable" items:

```
impl<T: WasmbinCountable + Encode> Encode for [T] {
    fn encode(&self, w: &mut impl std::io::Write) -> std::io::Result<()> {
        self.len().encode(w)?;
        for item in self {
            item.encode(w)?;
        }
        Ok(())
    }
}

impl<T: WasmbinCountable + Decode> Decode for Vec<T> {
    fn decode(r: &mut impl std::io::Read) -> Result<Self, DecodeError> {
        let count = usize::decode(r)?;
        std::iter::repeat_with(|| T::decode(r))
            .take(count)
            .collect()
    }
}
```

Once I was done with all the core & helper types, I could write a proc-macro to handle the custom derive. From here on, defining the Wasm structures was a breeze.

Let's take an `Import` node for the example:

```
#[derive(Wasmbin, ...)]
pub struct Import {
    pub path: ImportPath,
    pub desc: ImportDesc,
}
```

Under the hood, this auto-generates corresponding implementations by recursively walking over fields, just like other traits do:

```
impl Encode for Import {
    fn encode(&self, w: &mut impl std::io::Write) -> std::io::Result<()> {
        match *self {
            Import {
                path: ref __binding_0,
                desc: ref __binding_1,
            } => {
                Encode::encode(__binding_0, w)?;
                Encode::encode(__binding_1, w)?;
            }
        }
        Ok(())
    }
}

impl Decode for Import {
    fn decode(r: &mut impl std::io::Read) -> Result<Self, DecodeError> {
        Ok(Import {
            path: Decode::decode(r)?,
            desc: Decode::decode(r)?,
        })
    }
}
```

For enums, I've leveraged the recently added discriminant support for arbitrary enums. It's currently natively supported only on nightly Rust, but if I drop another custom proc-macro in front (`#[wasmbin_discriminants]`), then I can "polyfill" the support by reading the discriminants and stripping them away for the stable compiler:

```
#[wasmbin_discriminants]
#[derive(Wasmbin, ...)]
#[repr(u8)]
pub enum ImportDesc {
    Func(TypeId) = 0x00,
    Table(TableType) = 0x01,
    Mem(MemType) = 0x02,
    Global(GlobalType) = 0x03,
}
```

This way, I'm telling my derive code that the discriminant should be parsed as a `u8` and then matched to parse the corresponding variant (and vice versa during serialization). The example above generates code like following:

```
impl Encode for ImportDesc {
    fn encode(&self, w: &mut impl std::io::Write) -> std::io::Result<()> {
        match *self {
            ImportDesc::Func(ref __binding_0) => <u8 as Encode>::encode(&0x00, w)?,
            ImportDesc::Table(ref __binding_0) => <u8 as Encode>::encode(&0x01, w)?,
            ImportDesc::Mem(ref __binding_0) => <u8 as Encode>::encode(&0x02, w)?,
            ImportDesc::Global(ref __binding_0) => <u8 as Encode>::encode(&0x03, w)?,
            _ => {}
        };
        match *self {
            ImportDesc::Func(ref __binding_0) => Encode::encode(__binding_0, w)?,
            ImportDesc::Table(ref __binding_0) => Encode::encode(__binding_0, w)?,
            ImportDesc::Mem(ref __binding_0) => Encode::encode(__binding_0, w)?,
            ImportDesc::Global(ref __binding_0) => Encode::encode(__binding_0, w)?,
        }
        Ok(())
    }
}

impl DecodeWithDiscriminant for ImportDesc {
    type Discriminant = u8;

    fn maybe_decode_with_discriminant(
        discriminant: u8,
        r: &mut impl std::io::Read,
    ) -> Result<Option<Self>, DecodeError> {
        Ok(Some(match discriminant {
            0x00 => ImportDesc::Func(Decode::decode(r)?),
            0x01 => ImportDesc::Table(Decode::decode(r)?),
            0x02 => ImportDesc::Mem(Decode::decode(r)?),
            0x03 => ImportDesc::Global(Decode::decode(r)?),
            _ => return Ok(None),
        }))
    }
}
```

Now, you might notice that I've introduced another intermediate decode trait for enums. The reason here is that I've made my API work with arbitrary streams. It means that, once a "discriminant" (the byte which determines which variant should be parsed) is taken from the stream, it can't be put back.

This could be a problem for nested enums, where we don't know in advance which variant is going to be matched. For example, before the introduction of multi-value types, my `BlockType` definition looked like this:

```
#[derive(Wasmbin, ...)]
enum BlockType {
  Empty = 0x80,
  Value(ValueType),
}
```

Here, the autogenerated code needs to read a discriminant and match it against `0x80`. If it matches, then the variant is `Empty`, but if it doesn't, it needs to reuse the same discriminant to parse the nested `ValueType` instead:

```
impl Encode for BlockType {
    fn encode(&self, w: &mut impl std::io::Write) -> std::io::Result<()> {
        match *self {
            BlockType::Empty => <u8 as Encode>::encode(&0x80, w)?,
            _ => {}
        };
        match *self {
            BlockType::Empty => {},
            BlockType::Value(ref __binding_0) => Encode::encode(__binding_0, w)?,
        }
        Ok(())
    }
}

impl DecodeWithDiscriminant for ImportDesc {
    type Discriminant = u8;

    fn maybe_decode_with_discriminant(
        discriminant: u8,
        r: &mut impl std::io::Read,
    ) -> Result<Option<Self>, DecodeError> {
        Ok(Some(match discriminant {
            0x80 => BlockType::Empty,
            _ => {
                if let Some(res) =
DecodeWithDiscriminant::maybe_decode_with_discriminant(discriminant, r)? {
                    Self::Value(res)
                } else {
                    return Ok(None);
                }
            }
        }))
    }
}
```

If `ValueType` would implement only `Decode` directly, there would be no way to pass the same discriminant again (another alternative could be to allow only peekable streams, but it would be a user-facing inconvenience and not worth it for just passing one byte around).

Similarly, I can put `derive(Wasmbin)` on most other structs and enums that Wasm module consists of. For example, this is how the definition of the `ValueType` enum looks like:

```
#[derive(Wasmbin, ...)]
#[repr(u8)]
pub enum ValueType {
  I32 = 0x7F,
  I64 = 0x7E,
  F32 = 0x7D,
  F64 = 0x7C,
}
```

Or, probably the most interesting one, `Instruction` enum (showing only part of the list because the full one is **huge** and would require even more, potentially erroneous, code if not for auto-generation):

```
#[wasmbin_discriminants]
#[derive(Wasmbin, ...)]
#[repr(u8)]
pub enum Instruction {
  Unreachable = 0x00,
  Nop = 0x01,
  BlockStart(BlockType) = OP_CODE_BLOCK_START,
  LoopStart(BlockType) = OP_CODE_LOOP_START,
  IfStart(BlockType) = OP_CODE_IF_START,
  IfElse = 0x05,
  End = OP_CODE_END,
  Br(LabelId) = 0x0C,
  BrIf(LabelId) = 0x0D,
  BrTable { branches: Vec<LabelId>, otherwise: LabelId } = 0x0E,
  Return = 0x0F,
  Call(FuncId) = 0x10,
  CallIndirect { ty: TypeId, table: TableId } = 0x11,
  /* ... truncated as it's reallllly long */
}
```

Of course, there are few exceptions that can't be mapped as easily and need to be handled manually. For example, I've mentioned that the `BlockType` definition above was before the introduction of multi-value types to WebAssembly.

In order to accomodate for multi-value types, the spec definition was modified to the following:

> Block types are encoded in special compressed form, by either the byte 0x40 indicating the empty type, as a single value type, or as a type index encoded as a positive signed integer.
>
> ```
> blocktype ::= 0x40         => ε
>             | t:valtype    => t
>             | x:s33        => x (if x >= 0)
> ```

That is, now only a difference in the bit pattern determines whether something is a `ValueType`, or a beginning of a LEB128-encoded 33-bit signed integer that should be treated as a type index for the multi-value type... Yeah, there is no *good* way to automate this and it's easier to define encoding/decoding manually.

```
impl Decode for BlockType {
  fn decode(r: &mut impl std::io::Read) -> Result<Self, DecodeError> {
    let discriminant = u8::decode(r)?;
    if discriminant == OP_CODE_EMPTY_BLOCK {
      return Ok(BlockType::Empty);
    }
    if let Some(ty) = ValueType::maybe_decode_with_discriminant(discriminant, r)? {
      return Ok(BlockType::Value(ty));
    }
    let index = /* ... */;
    Ok(BlockType::MultiValue(TypeId { index }))
  }
}
```

But, aside from those edge-cases, the autogenerated implementations are covering majority of the spec and have been a huge boost to the maintainability and ease of future additions. For example, this is all it took to implement the `tail-call` proposal:

```
+#[derive(Wasmbin, Debug, Arbitrary, PartialEq, Eq, Hash, Clone, Visit)]
+pub struct CallIndirect {
+    ty: TypeId,
+    table: TableId,
+}
+
 #[wasmbin_discriminants]
 #[derive(Wasmbin, Debug, Arbitrary, PartialEq, Eq, Hash, Clone, Visit)]
 #[repr(u8)]
 pub enum Instruction {
     ...
     Call(FuncId) = 0x10,
-    CallIndirect {
-        ty: TypeId,
-        table: TableId,
-    } = 0x11,
+    CallIndirect(CallIndirect) = 0x11,
+    #[cfg(feature = "tail-call")]
+    ReturnCall(FuncId) = 0x12,
+    #[cfg(feature = "tail-call")]
+    ReturnCallIndirect(CallIndirect) = 0x13,
     Drop = 0x1A,
     ...
```

High-level API

Generally, you can access and modify fields directly on arbitrary type, as well as `::decode(...)` it from a stream or `.encode(...)` to one.

Aside from that, the top-level `Module` type has a couple of extra convenience helpers, with few of them - in particular, retrieving and inserting sections in the correct order - inspired by other libraries mentioned above:

Struct wasmbin::module::Module [-][src]

[+] Show declaration

Fields

```
sections: Vec<Section>
```

Methods

```
impl Module                                                                    [src]

pub fn decode_from(r: impl Read) -> Result<Module, DecodeError>                [src]

pub fn encode_into<W: Write>(&self, w: W) -> Result<W>                         [src]

pub fn find_std_section<T: StdPayload>(&self) -> Option<&Blob<T>>              [src]

pub fn find_std_section_mut<T: StdPayload>(&mut self) -> Option<&mut Blob<T>>  [src]

pub fn find_or_insert_std_section<T: StdPayload>(                              [src]
    &mut self,
    insert_callback: impl FnOnce() -> T
) -> &mut Blob<T>
```

Sections are represented as a vector of tagged enums rather than properties to guarantee order preservation.

Enum wasmbin::sections::Section [-][src]

```
[-]
    [+] Expand attributes
    pub enum Section {
        Custom(Blob<Custom>),
        Type(Blob<Type>),
        Import(Blob<Import>),
        Function(Blob<Function>),
        Table(Blob<Table>),
        Memory(Blob<Memory>),
        Global(Blob<Global>),
        Export(Blob<Export>),
        Start(Blob<Start>),
        Element(Blob<Element>),
        Code(Blob<Code>),
        Data(Blob<Data>),
    }
```

In WebAssembly spec, each section is preceded with a byte length to make it easier for parsers to read them lazily, skip them, or even read in parallel. In Wasmbin type system, this is implemented via a lazy `Blob` wrapper which you can see in the bodies of each `Section` variant above.

This means that, when you call a `Module`, then only raw byte vectors are read for each section, and then you can choose which sections you *actually* want to decode and when. For example, if you were to find an import for `env.memory` you could write code like following:

```
if let Some(imports) = wasmbin.find_std_section::<payload::Import>() {
  for import in imports.try_contents()?.iter() {
    match import.desc {
      ImportDesc::Mem(_) => {
        if import.path.module != "env" || import.path.name != "memory" { /* ... */ }
      }
      _ => {}
    }
  }
}
```

In this case, `Blob::try_contents` will lazily parse the contents on the first access and store them for future reads.

As an extra optimisation, if you don't modify the section, then during reserialization it will use the original raw bytes, but if some section is modified by accessing `Blob::try_contents_mut` instead, then it will recursively serialize the contents, too.

You can find the full API with several more traits and methods here: https://docs.rs/wasmbin/latest/wasmbin/module/struct.Module.html

Trade-offs

This post wouldn't be complete and fair without talking about trade-offs and downsides of this approach and the current implementation. Just to name a few major ones I keep in mind:

- The type structures are intentionally mirroring the WebAssembly spec rather than abstracting it away. On one hand, this is what allows to auto-generate parsing, serialization and visiting code for most structures, but, on another, it also means that any change to the spec structure has a higher risk of being a breaking API change in the crate. One example of this can be seen above in the `tail-call` proposal example - `CallIndirect` structure got extracted as it's now reused by several variants, so any users will have to make minor adjustment to their code to adapt on upgrade.
- The encoding / decoding code is currently designed to work with arbitrary `std::io::Read` / `std::io::Write` streams. On one hand, it makes it easier to integrate with any I/O sources, but, on another, this choice makes it slower and more memory-consuming for the common scenario of working with byte vectors.
- While most of the changes require *less* code to work, the expansive usage of macros (derive, proc-macros as well as regular ones) is likely to make the contribution curve steeper than in alternatives for people who are less accustomed to this language feature.

Feedback

Let me know if you find `wasmbin` useful or have any questions about the article, please do let me know either on the repo or Twitter!