Simplifying WebAssembly Interactions

afsal thaj

Why WebAssembly ?

- Fast, portable, and secure by design
- Runs in a lightweight, sandboxed environment
- Enables polyglot programming and language interoperability
- Supported by major platforms and browsers
- Evolving towards a component model for modular, introspectable systems

And one thing that I love - Introspection!

My appreciation for introspection began with **functional programming**

Seeing that same introspection system-wide through **WebAssembly** is deeply satisfying.

🔍 ***Introspection*** – The host sees what your program actually does

🛡️ ***Limits*** unexpected or malicious behavior. The host decides the implementation.

🧩 ***Designed, not patched*** – A step beyond reactive security fixes

101

A WA **component** is a self-contained unit of code that can be independently compiled, shared, and executed.

101

WIT (WebAssembly Interface Types): A specification used to define how WebAssembly components communicate with each other.

101

Wasmtime: High-performance WebAssembly runtime. A real world example of how the whole component model and WIT are intended to work in practice

Interact with WASM components

Let's invoke functions in a WASM component, but without using the new Rib

We will use `wasmtime` cli to do this

As of now dev release required to invoke functions in a component

Steps to follow

- We will build a **simple component** and it's called `command` component
- We will walk through a simple **WIT** file that's part of the component
- A quick skim through the component code written in rust
- Try running the component using **wasmtime** and try invoking functions

Let' see this in action

Raw Interaction

- Gets difficult with complex types
- Limited interaction capability. Example: How about passing the result of one component to the other?
- Prone to making errors, less productivity

Solution

- Interacting with components with any runtimes has been difficult in the whole webassemby ecosystem
- So we really need a solution that can be easily integrated with the great runtimes of Webassembly!

And here it is!

A REPL which can be integrated with any runtimes backed by the type-safe Rib language

Rib REPL in wasmtime

A forked wasmtime has already got this integration to showcase the power of Rib REPL!

```
>>>
>>> let x = instance()
()
>>> x.add({title: "foo", priority: low, deadline: some("foo")})
ok({id: "1", title: "foo", priority: low, status: backlog, created-timestamp: 0, updated-timestamp: 0})
>>>
>>> x.delete("foo")
ok
>>> :type x.delete("foo")
result<_, string>
>>> let result = x.process-user(backlog, low, "foo")
()
>>> :type backlog
enum { backlog, in-progress, done }
>>> :type low
enum { low, medium, high }
>>>
```

Let's see this in action (Demo)

```
~/p/r/w/wasmtime  $  target/debug   ./wasmtime repl bidder_composed.wasm                    26s  Thu 15 May 18:30:19 2025
~/p/r/w/wasmtime  $  target/debug                                                          4482ms  Thu 15 May 18:30:32 2025
~/p/r/w/wasmtime  $  target/debug                                                          4482ms  Thu 15 May 18:30:32 2025
~/p/r/w/wasmtime  $  target/debug                                                          4482ms  Thu 15 May 18:30:33 2025
~/p/r/w/wasmtime  $  target/debug                                                          4482ms  Thu 15 May 18:30:33 2025
```

Rib REPL in wasmtime - parser errors

```
>>>
>>> let my-record = {foo: 1, "bar" : 42}
Parse error at line: 1, column: 26
Unexpected `"`
Expected letter
```

Rib REPL in wasmtime - compiler errors

```
>>>
>>> let x = instance()
()
>>> x.add({title: 1, priority: low, deadline: some("foo")})
[compilation error]
[position] 26
[cause] type mismatch at path: `title`. expected string
[help] invalid argument to the function `app:myapp/bid-api@0.1.0.{add}`
>>>
```

The errors without REPL

```
~/p/r/w/wasmtime  $  target/debug  ./wasmtime --invoke 'add({title: 1, priority: low, deadline: some("foo")})'
Error: failed to run main module `bidder_composed.wasm`

Caused by:
    0: while interpreting parameters in invoke "add({title: 1, priority: low, deadline: some("foo")})"
    1: invalid value type at 12..13
! ~/p/r/w/wasmtime  $  target/debug
```

What we achieved?

- Quick ***typesafe*** interactions with WASM components
- ***Method call syntax*** to invoke functions
- ***Autocomplete*** all the way
- **Inspect types** anytime
- Distinguish components, functions etc properly
- Descriptive ***compilation errors***

Rib

Onboard quickly

It's easy to use, and syntax is intuitive

- Simply start `wasmtime repl`, create an instance and rely on auto completes
- You know Rib if you know basic Rust syntax for most of it
- Sticks on to ***wasm wave syntax*** for wasm values

Consistent Method Call Syntax

```
// x is a component instance, cart is a resource instance
>>> let x = instance();
>>> let cart = x.cart(user-id);
>>> cart.add-item({product-id: 1, product-name: apple, quantity: 1, price: 1000})
```

Distinguish components with type parameters

```
>>> let x = instance[foo:bar]()
>>> let y = instance[bar:baz]()
>>> x.foo()
>>> y.bar()
```

Rib REPL - Distinguish functions within a component

baz and ***qux*** are interfaces having function ***bar***

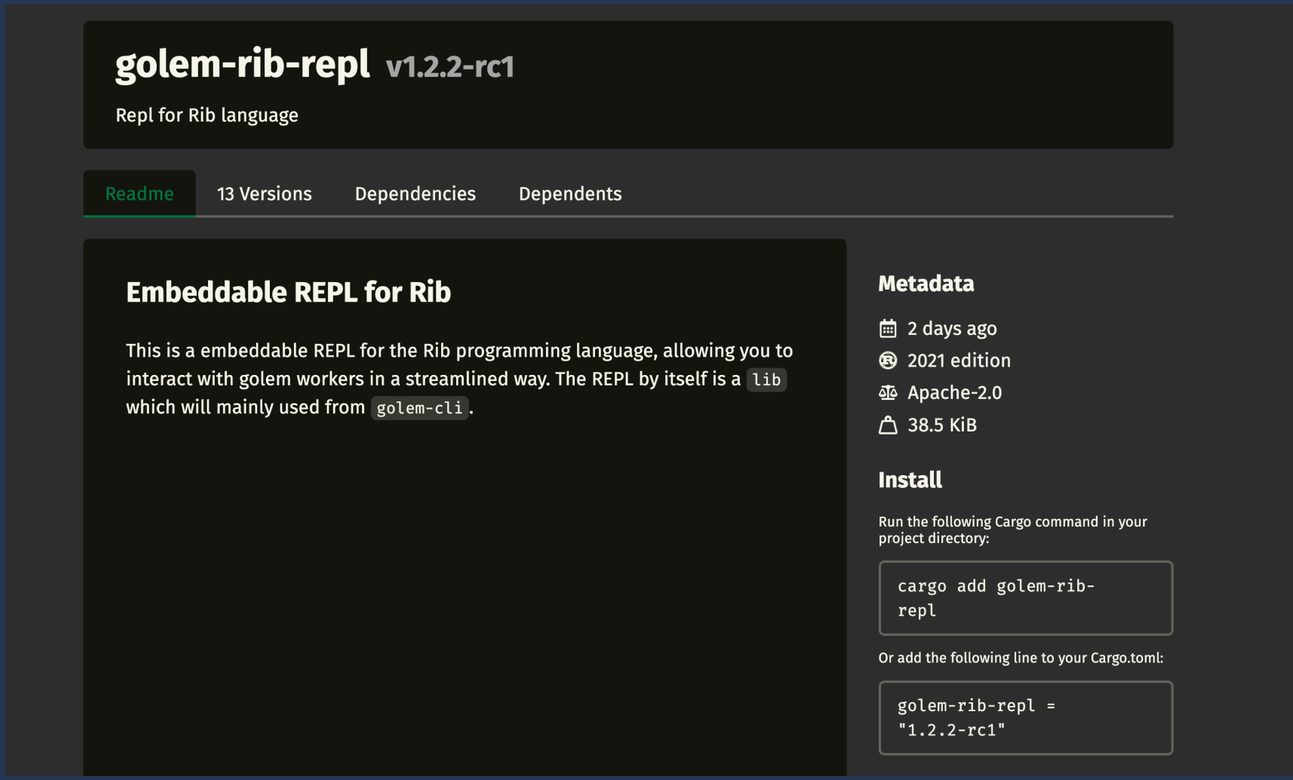
```
>>> let x = instance()
>>> x.bar[baz]()
>>> x.bar[qux]()
```

Embeddable

Is **Rib REPL** tied to `wasmtime`?

No! `Rib` REPL embeddable - that you integrate with any system/runtime clients.

We will see an example of it outside wasmtime



How about stateful applications?

- We used wasmtime repl and interacted with components that are not stateful
- We can somehow make it work though by having persistence layer within your app
- Or an ever running project serving http APIs

But I would rather simply switch to

Golem OSS

Why?

- I can write the application with an in-memory map
- I can't get into the boring DB and persistence layer
- I don't need to re-engineer what golem has done in running a component forever

Let's see how quick it is!

Bootstrap an app with golem

```
golem server run
golem app new shopping-cart rust
cd shopping-cart
golem component new rust my:shopping-cart
golem repl
```

Let's see this in action

Golem CLI with Rib REPL

```
golem repl
// interactive shell allowing users to select components
>>> let worker = instance("my_cart")
>>> cart.add-item({product-id: "foo", name: "foo", price: 42, quantity: 42})
>>> cart.get-cart-contents()

[{product-id: "foo", name: "foo", price: 42, quantity: 42}]
```

More on Rib Syntax

Rib Syntax and Grammer

https://learn.golem.cloud/rib covers most of it

- grammer
- types
- if-else
- pattern match
- list comprehension/reduction
- ranges and many more

Rib

Rib

Rib is the language used by the API Gateway that enables users to write programs capable of manipulating worker responses, which are WebAssembly (WASM) values.

Rib uses the WAVE (WebAssembly Value Encoding) syntax for encoding values.

Rib Grammar

`rib-expr` below defines the grammar of **Rib**.

```
rib_expr          ::= rib_expr_untyped (":" type_name)?
rib_expr_untyped  ::= simple_expr rib_expr_rest
simple_expr       ::= simple_expr_untyped (":" type_name)?
simple_expr_untyped ::=
      list_comprehension
    | list_reduce
    | pattern_match
    | let_binding
    | conditional
    | flag
    | record
```

On This Page

Rib Grammar
Return type of a Rib script
Core Language Elements
Identifier
Primitives
Numbers
Boolean
String
Lists
Options
Results
Tuples
Record
Variants
Enums
Flags
Expressions and Syntax Features
Assign to a variable
String Interpolation

Examples

```
>>>
>>> let x = some(1)
()
>>> match x { some(x) => x , none => 0 }
1
>>> match x { some(x) => x }
[compilation error]
position: 355
cause: non-exhaustive pattern match: the following patterns are not covered: `none`
help: to ensure a complete match, add missing patterns or use wildcard (`_`)
>>>
```

Examples (cont)

```
let worker =
  if result.user-id > 1000 then "group0" else "group1"
```

Examples (cont)

```
let limits = 1..10;

for i in limits {
  cart.add-items(i);
  yield "success";
}
```

Rib in API gateway of golem

Golem API gateway

- golem provides API gateway to quickly expose your component functions as http APIs
- Create an API definition which is a set of routes and each route has an embedded rib
- This rib is responsible for invoking component functions with optional transformation of input and output

Example Rib script in API definition

```
let user: string = request.path.user;

let worker-name = "user-${user}";

let worker = instance(worker-name);

let input = request.body;

worker.add-user(user)
```

Generating OpenAPI based on Rib script

- API gateway compiles Rib with valid global variables to be `["request"]`.
- Hence it knows statically the http request body, the path variables in the request route,
- Easily generate an open API spec.
- And any http calls that don't satisfy these requirements will become a BadRequest

Rib Implementation overview

- Parser implemented using combine library in Rust:
- Find the grammer in https://learn.golem.cloud/rib
- Rib compiler consist of independent 15+ compilation phases
- Examples: `type_pull_up`, `type_push_down_phase`, `type_check` and `unification`
- It scans script repeatedly for some of these phases, until it reaches a fix-point of inference defined by the sate of

Type inference FixPoint

```
pub fn type_inference_fix_point<F, E>(
  mut scan_and_infer: F,
  expr: &mut Expr
) -> Result<(), E>
where
    F: FnMut(&mut Expr) -> Result<(), E> {
  ...
}
```

Rib Implementation overview (contd)

- Rib Intermediate Representation (IR): **https://github.com/golemcloud/golem/blob/main/golem-rib/src/compiler/ir.rs**
- Stack based interpreter
- Configurable compiler and interpreter to customize the behaviour
  - Example customizations are golem API gateway, Rib REPL

Immediate things to fix and improve

- Precise and better compilation errors, at the unification phase
- std functions and user defined functions
- Make rib-repl as lightweight as possible before interacting with wasmtime authors.
- And resolve known issues here

Find more details at

https://learn.golem.cloud/rib

https://github.com/golemcloud/golem/tree/main/golem-rib

https://github.com/golemcloud/golem/tree/main/golem-rib-repl