WIP

Overview

This projet provides a validation framework for various implementations of PMT in various languages. The primary targeted languages are:

- C,
- go,
- rust,
- typescript,
- python,
- zig.

The solution is extensible to other languages.

Definitions

Validating an implementation of a PMT involves three actors:

1. The PMT implementation to be validated: *Implementation* (for short),
2. The validation framework: *Framework*,
3. A test runner implemented in the same language as the PMT implementation to be tested: **Runner**.

Implementation

- Responsibility: State of the art implementation of the PMT in the targeted language.

Framework

- Responsibility: Offer an uniform way to execute operations over an *Implementation* and validate its states.

Runners

- Responsibility: Glues the *Framework* and the *Implementation*. Its role is to request test instructions from the *Framework*, call the *Implementation* accordingly then pass the result back to the *Framework*.

Design considerations

Languages interoperability

Generally, languages offer interoperability with other language with via the so called `FFI` foreign function interface. `FFI` is closely related to the language memory model and many low level details like function arguments passing for function calls, etc.

The defacto standard for `FFI` is the `C` ABI which literarily means that all languages have to pass (and retrieve) arguments to(from) functions the `C` way and organize struct fields the `C` way (to name few).

Given that the `C` language has no memory management, allocated objects have to be manually freed to prevent memory leak. Language with a garbage collector

ABI has an impact over API. API in `C` are provided through so called `.h` files that defines `structs`, `enum` and functions signatures.

Here are some details for each targeted languages:

- C: obviously `C` uses the `C` ABI. In order to use a library build for interoperability with `C` a client only need the binary of that library and ideally its associated `.h`. Provided those two elements
- go,
- rust,
- typescript,
- python,
- zig.

Implementation

By definition a PMT *Implementation* defines its API, the set of operations to interact with the tree. From an implementation to another that set of operations may vary, see [main document](./main.md) for a comparison between known implementations. Because of those variations in operation sets some test cases may fail for some and succeed for others.

Framework

The *Framework* is implemented in Rust. It is delivered as a library that exposes its interfaces with the `C` ABI in order to be interoperable with as many language as possible.

Runners and bindings

For *Runners* to call the *Framework* we also provide thin bindings to C for go, python and zig. The Rust *Implementation* can obviously go the native way. The TypeScript *Implementation* is a special case where the `wasm` interface is envisioned. We might consider https://docs.deno.com/runtime/manual/runtime/ffi_api and go the `C` way everywhere.

Tests workflow

It's implemented by the *Runners Runners* implementations are implemented in the same language as the *Implementation*. That leaves room for more flexibility.

Runners take the form of this pseudo code:

```
pub fn run_test(implementation, framework, test_case) {
    commands = framework.get_commands(test_case);
    for command in commands {
        run_command(command, implementation);
    }
}

pub fn run_all_test(framework) {
    tests = framework.get_all_tests();
    for test in tests {
        run_test(implementation, framework, test)
    }
}
```

```
func run_command(command *C.Command, tree *trie.Trie) {
    /// TODO : implement each command who can be executed on the trie
    switch command.id {
    case C.Insert:
        panic("Insert not implemented")
    case C.Remove:
        panic("Remove not implemented")
    case C.Commit:
        panic("Commit not implemented")
    case C.CheckRootHash:
        panic("CheckRootHash not implemented")
    case C.RevertTo:
        panic("RevertTo not implemented")
    case C.Get:
        panic("Get not implemented")
    case C.Contains:
        panic("Contains not implemented")
    case C.GetProof:
        panic("GetProof not implemented")
    case C.VerifyProof:
        panic("VerifyProof not implemented")
    }
}
```

Implementation example :

- Golang

- Python
- Rust
- TypeScript
- C
- Zig

If needed write your runner

Implement the interface / trait that glued your implementation with the test framework