Rust Shell

Technical Specification

Ivan Yurov
Boston University – CS575

Platform

Rust Shell (Rush) is built with Rust programming language which provides access to low level functions along with safety and abstractions of higher level language. This project heavily utilizes monadic types, such as **Option**, **Result** as well as custom algebraic data type to wrap command during processing.

The target OS for Rush is originally **macOS**, but considering that both **macOS** and **Linux** are Posix systems, it should compile for **Linux** too. The only possible caveat is how it manages to access environment and user related info:

- User's name
- User's home dir
- Hostname

Input

Rush provides advanced editing capabilities such as cursor moving, inserting and removing characters at any position in the string. To implement that it takes control over input stream and processes each keystroke in a loop. All code related to reading user command located within **Prompt** module. Public interface contains single function **prompt::ask**, that returns a String. Steps that repeat at each iteration include:

- Capturing a key stroke
- Pattern matching against predefined key codes along with guards
- Performing action inside of matched block
- Rerendering current line

Input loop ends when user hits Enter, when loop breaks current command is returned by the function. In order to maintain cursor position, function uses local variable cmd_offset. Each call to rendering routine moves cursor programmatically in order to render content in appropriate place and returns it to the position where user left it.

Pattern matching clauses have guards that allows for protection against various overflow/underflow conditions. For example, if user hits RIGHT arrow while cursor is already at the end of the line, it doesn't match:

```
match getch() {
  KEY_RIGHT if cmd_offset < command.len() => cmd_offset += 1,
}
```

Main clause guards ASCII codes from from 32 to 126, which is too restrictive and does not support extended tables of UTF8, but perfectly fine for demonstration purpose. To store current command, it uses String, which is mutable and growable in Rust.

History

Previously performed commands are stored in stack that is built on **Vector<String>**. Prompt::ask accepts the reference of history stack as an argument. Once user accesses history by pressing UP and DOWN arrows, current command is stored in temp variable. When user reaches the beginning of the history, temp command is getting restored. Only actual commands go to history, not any service commands such as ‘help’, ‘!!’ or ‘exit’. As long as Vector type is located in heap, the length of command history is virtually unlimited.

Command

Code responsible for command execution is located within Command module. Public API presented by **command::run** function.

Tokenization

In order to split command into executable chunks the program splits by & first. The collection of joined commands becomes an iterator. Each joined command is being split by | sign. The collection of pipelined commands is being executed using ‘fold’ combinator, it allows to provide previous command struct to the next command in order to route Stdio streams.

```
for segment in cmd.split("&").map(|s| s.trim()) {
  let command = segment.split("|").map(|p| p.trim()).fold(None, |prev, piped| {
    let mut c = Command::new(piped, prev);
    c.run();
    Some(c)
  });
  // Printing of stdout/stderr is omitted
});
```

After creating Command struct it’s wrapped into Optional type, this is important to process first command correctly, cause it doesn’t have piped Stdin stream.

Spawning

Command struct constructor splits command by whitespaces, there first element of collection becomes exec and the rest is arguments. Using pattern matching again it finds out if exec is one of internal command type. If that’s the case it wraps it into CommandType::Internal ADT, and External otherwise. Previous command passed as reference wrapped in Optional type and Box. That allows for referencing instead of putting actual object into stuct because it wouldn’t be possible to have recursive structure as static analyzer cannot predict its size.

```
enum CommandType {
  Internal(String),
  External(String)
}
struct Command {
  pipe_in: Option<Box<Command>>,
  exec:    CommandType,
  args:    Vec<String>,
  stdout:  String,
  stderr:  String
}
```

Actual command execution happens when method run called on structure. Using pattern matching against custom ADT CommandType it figures out which command type we are going to run and spawns the process or finds internal command implementation by its name.

```
match self.exec {
  CommandType::External(ref cmd) => {
    match Cmd::new(&cmd)
     .args(&self.args)
     .stdin(Stdio::piped())
     .stdout(Stdio::piped())
     .spawn() {
      // Working with process: pass stdin, read stdout
    }
  },
  CommandType::Internal(ref int) => match int.as_ref() {
    "cd" => {
      // Perform CD command
    }
  }
}
```

Stdio

Only one mode of STDIO is supported with current implementation. Input for consecutive piped command provided right after process spawning, the output stored inside of command structure after command completion. It allows to render output with ncurses in synchronous fashion, but also brings some caveats. Such as running programs that require continuous access to stdin/stdout, in current implementation such process can't finish until user provides some input.