Shell scripting with Haskell

Franz Thoma

Berlin, 2017-02-24

Overview

- Shell scripting with high-level languages
- The `turtle` library
- Scripts & Dependency Management
- Parsing command line options
- A small application
- Conclusion

Shell scripting with high-level languages

Why use a high-level language for scripting

- **Abstraction**: Support for data structures, types and encapsulation helps allow cleaner semantics.
- **Flexibility**: High-level languages provide a rich set of both high-level and low-level libraries.
- **Scalability**: Module systems keep growing applications organized.
- **Robustness**: All of these make refactoring easier and applications more resilient.

Why use a statically typed language for scripting

- Dynamically typed languages are pretty popular in the scripting world as they are easy to hack away with.
- However, they share a number of problems with bare shell scripts: As scripts grow larger, the initial flexibility now makes the application increasingly harder to reason about.
- Statically typed programs are easy to refactor and extend

Why use Haskell for scripting

- Concise syntax, virtually no boilerplate
- Good library support, e.g. command line option parsers, `ncurses` bindings
- Can be interpreted using `runhaskell` or `stack runhaskell`

The `turtle` library

The `turtle` library

- `turtle` is an implementation of the UNIX command line environment in Haskell.
- The idea is to provide a set of recognizeable functions for accessing the file system, streaming data, and job control.

Demo

```
:set -XOverloadedStrings
import Turtle
import qualified Data.Text as Text
import qualified Filesystem.Path.CurrentOS as Path
```

```
projectDir <- pwd
print projectDir
cd =<< home
pwd
cd projectDir
pwd
view (ls ".")
let vi file = proc "vi" [file] empty
vi "README.md"
vi ".ghci"
```

Shell commands and their types

Turtle exposes some default shell commands:

- `echo :: Line -> IO ()`
- `cd :: FilePath -> IO ()`
- `mv :: FilePath -> FilePath -> IO ()`
- `cp :: FilePath -> FilePath -> IO ()`
- `rm :: FilePath -> IO ()`
- `pwd :: IO FilePath`

Building your own commands

The `proc` function allows calling external commands:

```
proc :: Text         -- Command
     -> [Text]       -- Arguments
     -> Shell Line   -- Lines of standard input
     -> IO ExitCode
```

Example:

```
vi :: FilePath -> IO ExitCode
vi file = proc "vi" [format fp file] empty
```

Shell streams

What about piping standard output to `less`?

```
less :: Shell Line -> IO ExitCode
less txt = proc "less" [] txt
```

The `Shell` type

`Shell a` is a stream of items of type `a`, with the possibility to execute `IO` actions.

- `stdin :: Shell Line`
- `input :: FilePath -> Shell Line`
- `yes :: Shell Line`
- `select :: [a] -> Shell a`
- `ls :: FilePath -> Shell FilePath`
- `cat :: [Shell a] -> Shell a`
- `view :: Show a => Shell a -> IO ()`

`Shell` composition

Function application/composition can be used to compose shell actions: `(.)` and `($)` act like unix pipes (but backwards):

```
less' :: FilePath -> IO ExitCode
less' = less . input
-- »cat <file> | less«
```

The bind operator `(>>=)` is the equivalent to shell expansions and `xargs`:

```
dircat :: FilePath -> Shell Line
dircat dir = ls dir >>= input
-- »cat $(ls <dir>)«
-- or »ls <dir> | xargs cat«
```

Scripts & Dependency Management

`runhaskell`

GHC has a script interpreter that can be used in a shebang line:

```
#!/usr/bin/env runhaskell

{-# LANGUAGE OverloadedStrings #-}

import Turtle

main = echo "Hello, World"
```

However, this fails unless `turtle` is installed globally in the user environment.

stack runhaskell

Stack has a remedy for the dependency problem:

```
#!/usr/bin/env stack
-- stack runhaskell --resolver=lts-8.0 --package=turtle

{-# LANGUAGE OverloadedStrings #-}

import Turtle

main = echo "Hello, World"
```

Parsing command line options

Auto-generated CLIs

```
> optparse/my-application.hs --help
My Application

Usage: my-application.hs

Available options:
  -h,--help                Show this help text
```

Turtle can generate this CLI for us:

```
{-# LANGUAGE OverloadedStrings #-}

import Turtle

main = do
    command <- options "My Application" (pure ())
    print command
```

Parameters and options

`turtle` provides an API for parsing parameters and options:

```haskell
data Options = Options
    { foo :: Bool
    , bar :: Maybe Text
    , baz :: Text }
    deriving (Show)

optionsParser :: Parser Options
optionsParser = liftA3 Options
    (switch "foo" 'f' "To foo or not to foo")
    (optional (optText "bar" 'b' "A bar option"))
    (argText "BAZ" "Some baz args")
```

```
> optparse/my-application-turtle.hs --help
Parse some options

Usage: my-application-turtle.hs [-f|--foo] [-b|--bar BAR] BAZ

Available options:
  -h,--help                Show this help text
  -f,--foo                 To foo or not to foo
  -b,--bar BAR             A bar option
  BAZ                      Some baz args
```

Simple CLIs

Sometimes only one or two simple parameters need to be passed. The `optparse-generic` library requires even less boilerplate to generate a CLI.

```haskell
{-# LANGUAGE DeriveGeneric, OverloadedStrings #-}

import Options.Generic

data Positional = Positional Text Int (Maybe Text)
    deriving (Show, Generic)

instance ParseRecord Positional

main = do
    command <- getRecord "My Application" :: IO Positional
    print command
```

```
> optparse/my-application-positional.hs --help
My Application

Usage: my-application-positional.hs TEXT INT [TEXT]

Available options:
  -h,--help                Show this help text
```

`bash` auto-completion

... is provided out of the box:

```
source <( my-application --bash-completion-script $(which my-application) )
```

A small application

Demo

`brick/select-file.hs`

Conclusion

Conclusion

Haskell has a rich ecosystem for scripting and small CLI applications:

- `turtle` for shell-like file-system access, external processes, and streaming
- `optparse-applicative` for declarative command line option parsing
- `brick` (and `vty`) as a lightweight `ncurses` textual interface
- `stack` with `stack runhaskell` for ad-hoc dependency management

Thank you

Questions?