Using Hesh, the Haskell-Extensible Shell

Chris Forno

2015-04-18

Hesh makes writing scripts in Haskell easier. Here's an example:

```
1  #!/usr/bin/env hesh
2  -- Backup a PostgreSQL database to an encrypted file.
3  
4  main = do
5    args <- System.Environment.getArgs
6    case args of
7      [database] -> do
8        today <- $(date +%Y-%m-%d)
9        let file = database ++ "-" ++ today ++ ".sql.gpg"
10       $(pg_dump $database) |> $(gpg -e -r $EMAIL) /> file
11     _ -> do
12       progName <- System.Environment.getProgName
13       $(echo "Usage: $progName <database>") /> "/dev/stderr"
```

Let's look at how this differs from standard Haskell.

- On line 1, hesh is used like runhaskell to start the script without previous compilation.
- On line 5, the call to `System.Environment.getArgs` has no accompanying qualified import.
- On line 8, `$()` is used to execute a command and read its output into a variable (like Bash's `$()`).
- On line 10, `$()` is used again, this time with a variable substitution (`$database`). However, instead of reading the result of the command into a variable, standard output is piped (`|>`) to another command. The output of that command is redirected (`/>`) to a file.

Hesh as a Shell

Hesh is not an interactive shell.[1] It's intended only for scripts. It's designed to reduce the verbosity of Haskell for shell-script-style tasks enough to make it a viable alternative to Bash.[2]

Hesh implements some of the functionality you'd expect from a shell, such as process spawning, I/O redirection, and variable substitution. For the rest, you can rely on Haskell and its libraries.

Spawning Processes

Spawning a process with `$()` behaves somewhat similar to what you'd expect from other shells, but is designed to be minimal.[3] It interprets its contents in 3 steps:

1. separate into tokens on whitespace (excluding quoted whitespace)
2. expand variables
3. spawn the command specified by the first token with all folowing tokens as arguments

`$()` returns either a `String`, a `CreateProcess`, or `()` (all in the `IO` monad), depending on the context. This—along with some proprocessing Hesh does before compiling your script—helps make sure it behaves as you'd expect in most cases.

Quoting

If you want a string containing whitespace to be passed as a single argument to a program, you must quote it with double-quotes (`""`). To use a quote inside double quotes, escape it with a backslash (`\"`).

Variable Expansion

`$()` expands any variable reference of the form `$variableName` or `${variableName}` (the latter allows for placing variables next to each other or within a string of text, like `${var1}${var2}` or `computer${suffix}`). If you want to include a literal `$`, escape it with a backslash (`\$`).

[1] Hesh is on its way to becoming an interactive shell, but there are non-trivial obstacles to making it one.

[2] If it wasn't obvious, Hesh is not Bash-compatible, even though it borrows some of its syntax.

[3] `$()` does not currently support nesting.

Variables must be valid Haskell identifiers (excluding infix operators),[4] such as `$varName` or `$val'`. For convenience, if the variable begins with an uppercase character, it's assumed to be an environment variable name and substituted with the corresponding environment variable.[5]

Note that variables are expanded *after* tokenization. This means that you don't have to worry about a variable containing whitespace: it will still be passed as a single argument (unlike with Bash). For example, in:

```
let var = "a filename with spaces"
in $(ls $var)
```

the `ls` program will be called with a single argument, not with 4 arguments.

Argument Lists

If you already have a list of arguments you'd like to pass to a program, you can use `cmd` instead of `$()`. `$()` itself invokes `cmd` after tokenization and variable expansion. The following 2 examples are equivalent:

```
$(ls a b c)
```

```
cmd "ls" ["a", "b", "c"]
```

Redirecting I/O

`|>` Pipes the output of the process on the left to the input of the process on the right.[6] This behaves the same as Bash's `|` (but is named differently to prevent clashing with Haskell's `|`).

`/>` Redirects the output of a process to a file. This behaves the same as Bash's `>` (again named differently to prevent clashing with Haskell's `>`).

`!>` Redirects stderr of a process to a file. This behaves the same as Bash's `2>` (which is not a valid identifier in Haskell).

`&>` Redirects both stdout and stderr of a process to a file. This behaves the same as Bash's `&>`.

[4] Allowed variable names start with any lowercase Unicode character followed by any number of Unicode alphanumeric characters and/or apostrophes (`'`).

[5] Allowed environment variable names start with any uppercase Unicode character followed by any number of Unicode alphanumeric characters and/or underscores (`_`). If you want to reference an environment variable that doesn't begin with an uppercase character, you'll need to read it into a Haskell variable via `System.Environment.getEnv` first.

[6] As with Bash, piping does not affect stderr.

`</` Read a file as input to a process. This behaves the same as Bash's `<`.

All of these functions are available via the `Hesh` library (which is automatically imported for use in Hesh scripts), so you can also use them in your non-Hesh Haskell programs.[7]

Each output operator (`/>`, `!>`, `&>`) has an append version (`/>>`, `!>>`, `&>>`) that will append to the given output file instead of overwriting it (like bash's `>>`).

Hesh as a Compiler

You can use Hesh to compile native binaries. In fact, that's what Hesh is doing each time it evaluates your script:

1. pre-process the script
2. generate a Cabal file
3. run Cabal build in a temporary directory[8]
4. execute the resulting binary

Preprocessing

Hesh's first feature was its automatic Cabal file generation. Without it, your scripts would be limited to the base libraries. From there, the preprocessor evolved hand-in-hand with Hesh's shell functions in order to make writing scripts feel as natural as possible. The preprocessor does the following:

1. desugars `$()` into `[sh||]` quasiquotes[9]
2. finds all uses of qualified names and adds them to the import list (e.g. `System.Environment.getArgs`)
3. looks in the Hackage database for any modules imported and adds them to the generated Cabal file
4. adds a type signature to `$()` in certain common contexts

[7]Note that `$()` is syntactic sugar for Hesh's `sh` quasiquoter, so you have to convert all instances of `$(command)` to `[sh|command|]` if you use the Hesh library outside of Hesh.

[8]A new temporary directory is created every time the script changes. Hesh used to also use a Cabal sandbox to try to mitigate dependency problems, but the result was found to be too expensive while developing and testing scripts.

[9]Desugaring activates the TemplateHaskell and QuasiQuotes Haskell extensions. You can skip this desugaring by passing the `--no-sugar` or `-n` option to Hesh.

Hesh and Hackage

The first time you run Hesh, it will take a while. That's because it's parsing the Hackage database (the one created by `cabal update`) and converting it into a more convient form for future runs.[10] Hesh uses this database to look up which package a module belongs to.

Hesh uses a very simplistic method of looking up packages and specifying package version constraints. In particular, its behavior is undefined if more than 1 package exports a module with the same name. In order to disambiguate between packages that export the same module, you can use package-qualified imports.[11] For example, to import the vector package implementation of Data.Vector, use `import "vector" Data.Vector`.

[10]The modules from the Hackage database is cached in `modules.json` in your Hackage path (usually `packages/hackage.haskell.org` in your `.cabal` directory).

[11]See http://haskell.org/ghc/docs/latest/html/users_guide/syntax-extns.html#package-imports for more details.