HsOptions: Haskell command-line flag processing library

Jose Raymundo Cruz Henriquez
Advisor: Prof. Matthew Fluet
Rochester Institute of Technology

May 7, 2014

1 Introduction

HsOptions is a *Haskell* library that supports command-line flag processing.

It is similar to *getOpt()*, but for Haskell, and with a lot of neat extra features. Typically, an application specifies what flags it is expecting from the user – like `--user_id` or `--file <filepath>` – somehow in the code; *HsOptions* provides a declarative way to define the flags inside the Haskell source code.

Most flag processing libraries requires all the flags to be defined in a single point, such as the main file, but *HsOptions* allows the flags to be scattered around the code, promoting code reuse and scalability. A module defines the flags it needs and when this module is used in other modules it's flags are handled by HsOptions.

HsOptions is completely functional, because no global state is modified. The only IO actions performed are to get the command-line arguments and to expand the configuration files.

Another important feature of *HsOptions* is that it can process flags from text files as well as from the command-line. This feature is available with the use of the special use of the *−−usingFile <filename>* flag.

Contents

2 Project description

The project is defined as a library for command-line flags processing. A major requirement is the ability to configure and set values for the flags using configuration files. These are text files that are treated the same way the command-line input stream is. Multiple configuration files can be inserted in the command-line input stream by using a special keyword, this is explained in details in the solution section.

Many applications can be dynamically configured, through the use of configuration files and/or command-line flags. A library for command-line flag processing would allow for the declarative specification of flags/options and relationships between flags (e.g., enabling both option A and option B is an error; if option A is enabled, then option B defaults to a particular value, unless explicitly set to a different value).

The goal of this project is to build a library that will handle the most common use cases regarding command line flags/options and that will add advanced functionality that will help reduce the development time and bad practices significantly. API and infrastructure design is important in this project since the library will be built from zero. A strong design should allow the library to support all desired features in a concise manner.

Functional programming is the primary tool of this library, not just because the project targets Haskell (a functional programming language) but because this programming paradigm have great benefits such as more concise code, programs easy to analyze and think about, consistency and immutability. The next section describes the features that functional programming provides that contribute to the development of this project.

3 Functional Programming features used

- First class and Higher-order functions

  Functions are first class members and can be passed in as arguments to other functions or can be returned as a result of a computation. This will allow high level customization of the core functions that this project will provide such as options with custom parsers and custom validation of values.

- Pure functions

  Each function is pure and will not produce any side effect when executed. This will allow modularization of the library and reuse of the flags definitions on separate modules. Also will provide an easy setup for combining modules without worrying of any order issue.

- Recursion

  Recursion is one of the major benefits of functional programming. Will be very useful when parsing the configuration files for this project as the intrinsic behavior of a hierarchical configuration files structure is a great fit for a recursive parsing and evaluation.

- Lazy evaluation

  This feature will help the speed up the run-time of the application. It will evaluate lazily each statement and only compute the required values. Will come in handy when evaluating a set of flags an error is found and the execution must be terminated as it will be expensive to keep evaluating once an error is found such as invalid syntax.

- Type system

  The system is type checked which means that type errors will be found at compile time. This will help reduce run-time errors and will help build a more robust system.

4 Solution

4.1 Project Features

HsOptions provides a lot of features to make flag handling easier for the developer. This section summarizes these features and gives a general overview on how each feature would be useful. For a more details see the *Tutorial Introduction: API* section.

- **Defining flags**

  A flag can be defined using the *make* method. It takes the name of the flag, the help text and a function that parses the input string value to the corresponding type of the flag. For example: `appIdFlag = make ("app_id", "the id help", [parser intParser])`.

- **Process flags**

  Flags are processed with the *processMain* method. It takes the description of the program, a list of all declared flags, and several callback functions for success, error and help. This method should be called in the main method of the Haskell program.

- **Get flag value**

  The *get* method is used to get the value of a flag. It takes the result object returned from the *processMain* method and a declared flag and will return the value of that flag inside the result argument. For example: `let id = get result appIdFlag`.

- **Optional and Required flags**

  All flags are required by default. To mark a flag optional the *isOptional* flag configuration is used. For example: `userLastName = make ("last_name", "help", [maybeParser stringParser, isOptional])`.

- **Configuration files**

  Flags can be parsed from command-line input or from text file. To send a flag value stream from a configuration file use the *usingFile* special flag. If you run the program with `runhaskell Program.hs --usingFile file.conf` then flag values will be parsed from *file.conf*. Multiple files can be included, and flags are parsed from both command-line and files in the left to right order.

- **Default value**

  A default value can be set to a flag using the *defaultIs* configuration. If no value was sent to the flag then this default value is used.

- **Flag alias**

  Flag can have many names/aliases. This is configured using the *aliasIs* flag configuration as such: `id = make ("id", "the id", [parser intParser, aliasIs ["uid", "user_id"]])`.

- **Dependent defaults** With dependent defaults you can configure a default value of flag depending on a context using the *defaultIf*. This context is a function that takes the process result as argument and returns a *Maybe value*, *Nothing* meaning no default value for the given result and *Just* being the default value. It is useful to set a default value to a flag depending on the value of other flags.

```
log_output = make ("log_output", "the output",
                   [parser stringParser,
                    defaultIf (flags -> if flags `get` debugFlag
                                        then Just "debug.log"
                                        else Nothing)
                   ])
```

- **Optionally required**

  A flag can be marked as required under certain circumstances using the *requiredIf*. This circumstance is specified by a predicate function that takes the process results as argument and returns true or false. If the predicate returns true, and the user did not provide a value for the flag then a *Flag is required* error is reported.

```
log_output = make ( "log_output"
                  , "where to save the log."
                  , [ maybeParser stringParser
                    , requiredIf (\ flags ->
                          flags `get` log_memory == True
                        )
                    ]
                  )
```

- **Flag parsers**

  Multiple flag parsers were created for the most used Haskell types, such as *intParser, stringParser, charParser*, etc. These parsers define the type of the flag and how to parse a string value to the type. See the tutorial section for a list of all parsers.

- **Flag operations**

  Several operations can be used when sending flag values in the command-line. Append, prepend and inherit. Append *(+= or +=!)* will add the new flag value to the end of the previous flag value. Prepend *(=+ or =+!)* will do it at the start. Inherit is a way to reference the previous value using the *$(inherit)* variable. It is useful to expand the flag value in any way. For example:

```
$ runhaskell Prog.hs --user bat \
                     --user +=! man \
                     --user = no-$(inherit)-no.
```

  ... will expand *user* to the value *no-batman-no*

4.2 Example

This program defines two flags (*user_name* of type *String* and *age* of type *Int*) and in the *main* function prints the name and the age plus 5. It also adds the alias *u* to the flag *user_name*.

```
-- Program.hs
import System.Console.HsOptions

userName = make ( "user_name"
                , "the user name of the app"
                , [ parser stringParser
                  , aliasIs ["u"]
                  ]
                )
userAge = make ("age", "the age of the user", [parser intParser])

flagData = combine [flagToData userName, flagToData userAge]

main :: IO ()
main = processMain "Simple example for HsOptions."
                   flagData
                   success
                   failure
                   defaultDisplayHelp

success :: ProcessResults -> IO ()
success (flags, args) = do let nextAge = (flags `get` userAge) + 5
                           putStrLn ("Hello " ++ flags `get` userName)
                           putStrLn ("In 5 years you will be " ++
                                     show nextAge ++
                                     " years old!")

failure :: [FlagError] -> IO ()
failure errs = do putStrLn "Some errors occurred:"
                  mapM_ print errs
```

You can run this program in several ways:

```
$ runhaskell Program.hs --user_name batman --age 23
Hello batman
In 5 years you will be 28 years old!
```

```
... or:
    $ runhaskell Program.hs --user_name batman --age ten
    Some errors occurred:
    Error with flag '--age': Value 'ten' is not valid


... or:
    $ runhaskell Program.hs --help
    Simple example for HsOptions.
        --age        the age of the user
    -u  --user_name  the user name of the app
        --usingFile  read flags from configuration file
    -h  --help       show this help
```

4.3 Tutorial Introduction: API

4.3.1 Defining flags

A flag is defined using the *make* function. It takes the name of the flag, the help text and the parser. The parser specified how to parse the string value of the flag to the correct type. A set of default parsers are provided in the library for common types.

To define a flag of type *Int*:

```
age :: Flag Int
age = make ("age", "age of the user", [parser intParser])
```

To define the same flag of type *Maybe Int*:

```
age :: Flag (Maybe Int)
age = make ("age", "age of the user", [maybeParser intParser])
```

The function *maybeParser* is a wrapper for a parser of any type that converts that parser to a *Maybe* data type, allowing the value to be *Nothing*. This is used mostly for optional flags.

Instead of *intParser* the user can specify his custom function to parse the string value to the corresponding flag type. This is useful to allow the user to create flags of any custom type.

4.3.2 Process flags

To process the flags the *processMain* function is used. This function serves as a middle man between the real *main* and the flag processing. Takes 5 arguments:

- The description of the program: used when printing the help text.
- A collection of all the defined flags
- Three callback functions:
  - Success callback: called with the process results if no errors occurred
  - Failure callback: called if any error while processing flags occurred
  - Display help callback: called if the user sent the *–help* flag

This is an example of how to call the *processMain* function:

```
import System.Console.HsOptions

-- flags definitions
name = make ("name", "the name of the user", [parser stringParser])
age = make ("age", "the age of the user", [parser intParser])

-- collection of all flags
all_flags = combine [flagToData age, flagToData name]

-- real main
main = processMain "Example program for processMain"
                   all_flags
                   successMain
                   defaultDisplayErrors
                   defaultDisplayHelp

-- new main function
successMain (flags, args) = putStrLn $ flags `get` name
```

In this example, the provided implementations for the failure and the display help callback were used (*defaultDisplayErrors* and *defaultDisplayHelp*), so that we do not need to define how to print errors or how to print help.

As mentioned before, if no errors were found then *successMain* function is called. The argument sent is a tuple (*FlagResults*, *ArgsResults*). *FlagResults* is a data structure that can be used to get the flag's value with the *get* function. *ArgResults* is just a list of the non-flag positional arguments.

If there was any kind of errors while processing the flags the *display errors* callback argument is called with the list of *FlagError* as argument. The user can specify a custom function so he can handle the errors as he wishes.

The third callback, *display help*, is called when the user sent the special help flag (*–help* or *-h*). It takes the program description and all the information of the flags as a list of (*flag_name*, *[flag_alias]*, *flag_helptext*). The *defaultDisplayHelp* is a default implementation that prints the helptext in a standard format, usually this is the way to go unless the user wants to print the help text in a custom format.

4.3.3 Get flag value

A flag value is obtained by using the *get* function. It takes the *FlagResults* and a defined flag as a parameter, and it will look for the value of the flag inside the *FlagResults*. In a way you can think of *FlagResults* as a data structure that can be queried with flags to retrieve flag values.

The *FlagResults* are obtained by processing the flags with the *processMain* function.

The return type of *get* is the type of the flag, so if the flag is *Flag Int* then *get* returns an *Int* (so the flag value is typed).

For a given flag:

```
repeat = make ("repeat", "how many times to repeat", [parser intParser])
```

... we can grab it"s value after processing like this:

```
success :: (FlagResults, ArgsResults) -> IO ()
success (flags, args) = do let r = flags `get` repeat
                           putStrLn $ "The value of repeat is " ++ show r
```

4.3.4 Optional and Required flags

By default all flags are marked as required. If you want to make an optional flag then two things are required:

- First, the type of the flag must be *Flag (Maybe a)*, so that the flag can be *Nothing* if it was not provided and *Just value* if it was.
- Second, the flag must be configured using the *isOptional* flag configuration.

Example:

```
-- optional flag
database :: Flag (Maybe String)
database = make ("db", "the database",
                 [maybeParser stringParser, isOptional])

-- required flag
app_id :: Flag Int
app_id = make ("app_id", "application to run", [parser intParser])

-- combine all flags
all_flags = combine [flagToData database, flagToData app_id]

-- main
main = processMain "Sample" all_flags success
                   defaultDisplayErrors defaultDisplayHelp

-- success main
```

```
  success (flags, _) = do putStrLn $ "database: "
                                  ++ show (flags `get` database)
                          putStrLn $ "app_id: " ++ show (flags `get` app_id)
```

This is the expected behavior when getting the flag value:

```
    $ runhaskell Program.hs
    Errors occurred while parsing flags:
    Error with flag '--app_id': Flag is required

... as you can see only `app_id` is required, but not `database`.

    $ runhaskell Program.hs --app_id = 123
    database: Nothing
    app_id: 123

... value for `database` is `Nothing`.

    $ runhaskell Program.hs --app_id = 123 --db = local
    database: Just "local"
    app_id: 123
```

4.3.5 Configuration files

Flags can be processed not only from command-line input, but also from configuration text files. These text files are included at any point in the command-line stream by using the special flag `--usingFile <filename>`.

When the flag processor encounters a *usingFile* it reads the content of the file and runs the processor again with this content, consuming the *usingFile* flag and replacing it with all the new flags found inside the configuration file.

A configuration file can itself include other configuration files as well, by using the *usingFile* flag inside the file, so a tree of files can be created (a file can have a parent file, and a grandparent file, or a file can include multiple files to combine them together).

If there is any kind of error while reading the file, or there is a syntax error inside the file then that error is reported to the user. This is an example of a configuration file that has comments, and that includes two more files.

```
# combined.conf
```

```
--database = localdb
--usingFile = file1.conf
--usingFile = file2.conf
jack
jill
batman

# file1.conf
--flagA = 3

# file2.conf
--flagB = 42
```

So if we have a *Program.hs* that is configured with the flags *database*, *flagA* and *flagB*, and that prints the remaining positional arguments, then this is the output of the program for the following scenarios:

```
$ runhaskell Program.hs --usingFile combined.conf
database: localdb
flagA: 3
flagB: 42
args: ["jack","jill","batman"]
```

We can send more arguments, or modify flags, after or before including the file:

```
$ runhaskell Program.hs superman --usingFile combined.conf robin
database: localdb
flagA: 3
flagB: 42
args: ["superman", "jack","jill","batman", "robin"]
```

As you can observe *superman* and *robin* are respectively at the start and end of the positional arguments, that is because first *superman* is found in the input stream, then the *usingFile combined.conf* which gets evaluated and parsed, and when this is complete then the processor moves to *robin* which is captured as the last positional argument.

Here is another example on how we can override and extend the flags. We will change the *flagA* to 1024 and will append the value *.local* to the *database* flag.

```
$ runhaskell Program.hs --usingFile combined.conf \
                        --database +=! ".local" \
                        --flagA = 1024
database: localdb.local
flagA: 1024
flagB: 42
args: ["jack","jill","batman"]
```

4.3.6 Default value

There are two types of default flag values, a default value when the flag was not provided by the user, and another default value for when the user provided the flag but not the flag value. The flag configurations are *defaultIs* and *emptyValueIs*.

A default value can be configured for a flag by using the *defaultIs* flag configuration. It takes the value that the flag will have in case the flag is not provided by the user.

Example:

```
database = make ("database", "the db connection",
                 [ parser stringParser , defaultIs "local.sqlite"])
```

So for example:

```
  $ runhaskell Program.hs
  database: local.sqlite

... if you set the value then the default is ignored:

  $ runhaskell Program.hs --database production.sqlite
  database: production.sqlite

... but, it should be noted that if you send the flag, but not it's value,
    then an error will occur, as the system assumes you meant to set a
    value to the flag:

  $ runhaskell Program.hs --database
  Some errors occurred:
  Error with flag '--database': Flag value was not provided
```

```
... if you want to add a default value for the flag value is empty use
    the ‘emptyValueIs‘ flag configuration:
```

```
database = make ("database", "the db connection",
                 [ parser   stringParser
                 , defaultIs  "local.sqlite",
                 , emptyValueIs "prod.sqlite"])
```

```
$ runhaskell Program.hs --database
database: prod.sqlite
```

The combination of *defaultIs* and *emptyValueIs* makes it possible to define flags such as booleans. So we could set up a flag such as `--debug` (*Bool*) that will take the value *False* if missing and will take the value *True* if the user sent `--debug` without him having to say `--debug = True`.

4.3.7 Common configurations

There are some common patterns that occurs while configuring flags. These patterns can be put into a function for code reuse.

Boolean flag

A default behavior for boolean flag is that if the flag is missing then it's value is *False* and if the flag is present, even with a missing flag value, then it's value is *True*. For this the *boolFlag* flag configuration was created.

```
debug = make ("debug", "debug flag", boolFlag)

-- This is equivalent to:
debug = make ("debug", "debug flag", [ parser  boolParser
                                     , defaultIs  False
                                     , emptyValueIs True
                                     ])
```

This is because *boolFlag* is defined as such:

```
boolFlag :: [FlagConf Bool]
boolFlag = [ parser  boolParser
           , defaultIs  False
           , emptyValueIs True
           ]
]
```

4.3.8 Flag alias

Creates a flag configuration for the aliases of the flag.

Sets multiple alias for a single flag. `(--user_id alias: ["u", "uid")`. These aliases can be used to set the flag value, so `--user_id = 8` is equivalent to `-u = 8`.

They are set using the *aliasIs* flag configuration:

```
user_id = make ("user_id", "the id",
                [parser intParser, aliasIs ["u", "uid"]])
```

4.3.9 Dependent defaults

Creates a flag configuration that will define a default value for a flag based on a condition. This condition is a function that takes in the current *FlagResults* and returns *Nothing* if the there is no default value or the default value (*Just*) if there is one.

If the function returns a value, and the user did not send the flag in the input stream, then the default value associated with this function is used as the default value for the flag.

The dependent default value is configured by using the *defaultIf* function. It takes as arguments the *default value getter function* that given the *FlagResults* tries to return a default value.

Example:

```
userName = make ("user_name", "the user", [parser stringParser])

movie = make ( "movie"
             , "the movie of the user"
             , [ parser stringParser
               , defaultIf (\ flags ->
                     if flags `get` userName == "neo"
                     then Just "matrix"
                     else if flags `get` userName == "bruce"
                          then Just "batman"
                          else Nothing)
               ]
             )
```

This is the output for different scenarios:

```
$ runhaskell Program.hs --user_name other
```

```
  Some errors occurred:
  Error with flag '--movie': Flag is required

... none of the predicates matched, the flag is required by the user

  $ runhaskell Program.hs --user_name batman
  user_name: bruce
  movie: batman-begins

... the first dependent default matched, so it's value is used.

  $ runhaskell Program.hs --user_name neo
  user_name: neo
  movie: batman-matrix
```

This configuration is useful in scenarios where a flag's default value depends on the value of one or more flags.

4.3.10 Optionally required

You can mark a flag optionally required by using the *requiredIf* flag configuration.

This flag configuration needs a *predicate* function that given the current *FlagResults* returns *True* or *False* depending if the flag should or should not be required.

For example it is useful to make a flag required if another flag was set to a particular value:

```
log_memory = make ( "log_memory"
                  , "if set to true the memory usage will be logged"
                  , boolFlag)


log_output = make ( "log_output"
                  , "required if 'log_memory' is true"
                  , [ maybeParser stringParser
                    , requiredIf (\ flags ->
                        flags `get` log_memory == True
                      )
                    ]
                  )
```

... after the flags are processed then the optionally required condition is checked. If the configured predicate returns true an error is reported to the user:

```
  $ runhaskell Program.hs
  log_memory: False
  log_output: Nothing

... if you send the ‘log_memory‘ the conditional predicate will return
    ‘True‘ and the flag will be required:

  $ runhaskell Program.hs  --log_memory
  Some errors occurred:
  Error with flag '--log_output': Flag is required

... if you send the value for ‘log_output‘ then an error should not occur:

  $ runhaskell Program.hs  --log_memory --log_output /tmp/memorylog.tmp
  log_memory: True
  log_output: Just "/tmp/memorylog.tmp"
```

4.3.11 Global validation

A global validation rule is a function that will be evaluated with the *FlagResults* after the processing stage and will determine if the current state is valid.

It is the last stage of flag processing. If there is a validation error then this error is reported to the user. This validation is done by using the *validate* function that takes a function that returns a *Maybe String*, *Nothing* being a passing result and *Just err* being failing result with an *err* error message.

For example:

```
flagData = combine [ flagToData debugMemory
                   , flagToData debugDisk
                   , validate (\fr ->
                       if get fr debugMemory && get fr debugDisk
                       then Just $ "debug_memory & debug_disk can't be on"
                                ++ " at the same time"
                       else Nothing)
                   ]
```

An error will be produces if the application is run with a negative *user_id*.

4.3.12 Flag parsers

Parsers: Flag parser configurations.

- **intParser**. Parses a flag value to an integer.
- **floatParser**. Parses a flag value to a float.
- **doubleParser**. Parses a flag value to a double.
- **charParser**. Parses a flag value to a char.
- **stringParser**. Parses a flag value to a string.
- **boolParser**. Parses a flag value to a boolean.
- **arrayParser**. Parses a flag value to an array.

Parser wrappers:

- **toMaybeParser**.

  Takes a *parser* as argument and wraps it so it becomes a *Maybe a* parser.

  Used to convert an existent parser to an optional parser.

  ```
  intPaser :: FlagArgument -> Int
  toMaybeParser intParser :: FlagArgument -> Maybe Int
  ```

  If the flag was missing or the flag value was missing then the new parser will return *Nothing*, otherwise the wrapped parser is called.

  It comes handy when you create a flag of type *Maybe a* and you want to use one of the existent parsers:

  ```
  user_id :: Flag (Maybe Int)
  user_id = make ("user_id", "help", [parser (toMaybeParser intParser)])
  ```

  Since this seems to be a common pattern the *maybeParser* method was created that combines the *parser* function with the *toMaybeParser*. The previous example is equivalent to:

  ```
  user_id :: Flag (Maybe Int)
  user_id = make ("user_id", "help", [maybeParser intParser])
  ```

4.3.13 Flag operations

Flag operations allows the user to set the value of a flag based on the previous value set. This is useful in situations where configuration files are used, so that a child configuration file can extend the value of a flag set in a parent configuration file.

Operations are specified when setting a value for a flag. This is the syntax to set a flag: `--flag_name [operation] flag_value`. If the *[operation]* is not set then the `assign (=)` operation is implied.

Assign

This is the default operation. Sets the value of the flag, overwriting any previous value if there was any. This is the default operation unless the user changed it in the flag configuration.

```
Example:
    $ runhaskell Program.hs --file = "/home/user/" --file = "/tmp"
    file: "/tmp"
```

Inherit keyword

The *$(inherit)* keyword can be used in the flag value and will be expanded to the previous value of the flag (or to empty string if no previous value).

```
Example:
  $ runhaskell Program.hs --file = "/home/user" --file = "$(inherit)/local/tmp"
  file: "/home/user/local/tmp"

... and with no previous value:

  $ runhaskell Program.hs --file = "$(inherit)/local/tmp"
  file: "/local/tmp"
```

Append

It's a specialization of the *$(inherit)* keyword to append the current value of the flag to the previous. There are two ways to append, using the `+=` symbol or the `+=!` symbol.

They are the same except that `+=` puts a space between previous value and current value (if there is a previous value for the flag).

```
They are equivalent to:

  -- space in between
  --file += /local/tmp   <=> --file = "$(inherit) /local/tmp"

  -- no space in between
  --file +=! /local/tmp  <=> --file = "$(inherit)/local/tmp"

Example `(+=)`:

  $ runhaskell Program.hs --warning = "1 2" --warning += "3"
  warning: "1 2 3"

Example `(+=!)`:

  $ runhaskell Program.hs --warning = "warn-1,2" --warning +=! ",3"
  warning: "warn-1,2,3"
```

Prepend

It's a specialization of the *$(inherit)* keyword to prepend the current value of the flag to the previous. There are two ways to prepend, using the `=+` symbol or the `=+!` symbol.

They are the same except that `=+` puts a space between previous value and current value (if there is a previous value for the flag).

```
They are equivalent to:

  -- space in between
  --file =+ /local/tmp   <=> --file = "/local/tmp $(inherit)"

  -- no space in between
  --file =+! /local/tmp  <=> --file = "/local/tmp$(inherit)"

Example `(=+)`:

  $ runhaskell Program.hs --warning = "1 2" --warning =+ "0"
  warning: "0 1 2"

Example `(=+!)`:

  $ runhaskell Program.hs --warning = "warn-1,warn-2" --warning =+! "warn-0,"
  warning: "warn-0,warn-1,warn-2"
```

Change flag default operation

By default a flag's default operation is the `assign (=)` operation. So if the user sends a flag and it's value without explicitly using an operation this is the operation used.

Now if you want to change this behavior for a given flag you can do so by using the *operation* flag configuration. This takes an operation as an argument and sets this as the default operation for the flag:

```
warning = make ("warn", "warnings to print",
                [parser stringParser , operation append])
```

```
Now if you run the program like this:

  $ runhaskell Program.hs --warn 1 --warn 2 --warn 3
  warn: "1 2 3"

You can overwrite this default if you specify the operation
in the command line:

  $ runhaskell Program.hs --warn 1 --warn 2 --warn 3 --warn = 0
  warn: "0"
```

The available operations for the flag are these:

- *assign* `(=)`
- *append* `(+=)`
- *append' (no space)* `(+=!)`
- *prepend* `(=+)`
- *prepend' (no space)* `(=+!)`

5 Other libraries comparison

A feature comparison of *HsOptions* was made against the most common command-line parsing libraries used in the Haskell community. The two libraries taking into consideration were **hflags**[1] and **options**[2].

Feature	HsOptions	HFlags	Options
Typed flags	X	X	X
Auto Help text	X	X	X
Optional/Required flags	X	X	
Default value	X	X	X
Flag Alias	X	X	X
Dependent default	X		
Global validation	X		
Optionally required	X		
Stateless	X		X
Flag operations	X		
Read from files	X		
Flag value inheritance	X		
Distributed	X	X	

The comparison table shows that HsOptions rises above the other two. You can observe the list of functionality in HsOptions is significantly greater as compared to the rest. The basic functionality is supported (define and parse flags, get flag values, etc), but HsOptions goes beyond and adds high level features such as dependent default and configuration files.

To see benefits and details on each of these features take a look at the *Tutorial Introduction: API* section.

6 Design decisions and problems faced

Several design decisions were made while developing this library, these decisions are documented in this section along with why one design is better than another and the pros and cons of each. Also it should be mentioned that each of this design decision were made at a given point on this project life cycle and may be changed in the future if it makes sense. This information is presented in chronological order.

6.1 March 12, 2014

The API of the make flag function was changed. Now the user needs to explicitly state that a flag is either required or optional, thus changing the underlying Haskell Flag Type and the behavior of the flag.

```
This was the previous API:
  -- make :: (name, helpText, parser)
make :: (String, String, String -> Maybe a)

This is the new API:
  -- make :: (name, helpText, parser)
make :: (String, String, Maybe String -> Either FlagError a)
```

The change was on the parser function. The reason of the change was that optional flags needed to be implemented in the code, but the previous API had a flaw which was that the parser just said if a string value could be parsed to the correct flag type or not (Just value or Nothing), but if Nothing was returned we had no more information, just that it failed, without knowing why. The new design allows us to differentiate when a flag was missing or a flag value was invalid. If a flag is required then a Missing Flag returns an MissingFlagError, but if the flag is optional a Missing Flag does not return any error, as the flag is optional and the program should not complain if the flag was not passed in, as it is an optional flag.

In terms of users perspective, this is how the API changed when creating a flag:

```
userId :: Flag Int
userId = make ("user_id", "help user id", required intFlag)

userName :: Flag (Maybe String)
userName = make ("user_name", "help user name", optional stringFlag)
```

6.2 March 15, 2014

6.2.1 Flag creation API - flag constraints

Flag constraints needed to be included in the API and some different ideas were considered. Some of these constraints were default values, require a flag conditionally, etc. Initially this approach was taken for the API:

```
userId :: Flag (Maybe Int)
userId = make ("user_id",
               "the user id of the app. required if printUser is True",
               optional intFlag)

printUser :: Flag (Maybe Bool)
printUser = make ("print_user", "prints the user", optional boolFlag)

helpFlag :: Flag (Maybe Bool)
helpFlag = make ("help", "show this help", optional boolFlag)

constraints ::[ FlagConstraint]
constraints = [
  userId `requiredIf` (\ flagResults -> isTrue (get flagResults printUser))
  helpFlag `defaultsTo` (\ flagResults -> Just False)
  userId `alias` "uid"
  ]

{- FlagData needs to contain the constraints -}
flagData :: FlagData
flagData = mkFlagData constraints [
  flagToData userId,
  flagToData helpFlag
  flagToData printUser
  ]
```

In this style, Flag Constraints is a separate data structure that holds all the constraints for all the flags. These constraints needed to be carried around in the flag data so that the library could enforce them.

This design was rejected for several issues. First, it presented problems with types. Because it's a separate data structure, its type needs to be generic, but since each flag is of a different type (Flag Int, Flag String, Flag Bool) then their constraints needed to be transformed to a generic model. So for instance the *defaultsTo* constraint needed to return a value that either a String or a value that could be converted to a string (Show a). But this was inconvenient. The desire for type checking was wanted so it was better to convert the return value to a string, but then again if a custom flag type is used then a way to convert this flag to string

needs to be defined, which is more work to the user. Furthermore, calling show on a String itself will add quotation marks to the resulting value which we don't want.

Another issue with this design is added more parameters to the basic API functions the user needs to call, for instance since constraints are an argument of mkFlagData then the user needs to pass [] if there are no constraints on the flags.

6.2.2 Flag creation API - flag constraints new design

A new design was put in place that solves the problems with the previous one and added some benefits in terms of flexibility. Some sort of Combinator was created to hold all the constraints for a single flag, and these constraints need to be defined in the flag itself. Here is an example of the new API:

```
userIdFlag :: Flag Int
userIdFlag = make ("user_id", "the user id of the app", [parser intParser])

database :: Flag (Maybe String)
database = make ("database",
                 "database connection string. required if user_id == -1",
    maybeParser stringParser :
     requiredIf (\ fr -> get fr userIdFlag == -1))

tellJoke :: Flag Bool
tellJoke = make ("tell_joke", " tells a joke", boolFlag)

-- Source: HsOptions.hs
boolFlag :: [FlagConf Bool]
boolFlag = [parser boolParser,
            defaultIs False,
            emptyValueIs True]
-- End Source: HsOptions.hs
```

As you can observe here, the third parameter of make was changed to a collection of constraints. The parser is considered one of the constraints. Since we are in the scope of make method then we know the type of the flag, so we can use this to return a type checked value for the constraints such as defaultsTo.

Also, since it's a collection of constraints we can use this same API to create default collections for specific flag types, as you can observe with the boolFlag (provided on the library). A boolFlag is just a flag that parses boolean, that has a default value of False if flag is not provided and that has a value of True if the flag was provided with no value. This allows the user to define a boolean flag easily with the functionality of this being false by default and being true if the flag was mentioned on the arguments.

You can also observe this collection of constraints in the requiredIf method call, currently for a flag to be conditionally required this flag must also be optional.

6.3 March 17, 2014

6.3.1 Program main method and help

Initially the help was pushed to the user's level, he had to define a boolean flag for the help and check if this flag was true himself. Methods were provided to print the help text, the user could choose to use these methods or to create his own.

One problem appeared, for instance in the scenario where we had two flags: Help and UserId(required), if the user were to call the application with –help but not with `--userId val` then instead of the help text being displayed an error of *user id flag is required* was displayed. This was because the help flag was not treated as an special flag but as any other flag, and the flag processor reported errors so the user had to handle those instead of printing the help.

To solve this the decision to push the help flag manipulation inside the library was made. Now the library had to do some IO to print the help text in case the help flag was true. A main method was created to process the flags and branch depending on the current status.

There are three scenarios. The help flag is true, so we need to print the help text. Second, the help flag is false and there are parsing errors, then we must notify the user the errors. Last, the help flag is false and no error, then the application must continue. A default main method was created that takes in 3 callbacks:

```
main_errors :: [FlagError] -> IO ()
main_success :: ProcessResults -> IO ()
defaultDisplayHelp :: String -> [(String, String)] -> IO ()

main :: IO ()
main = processMain description
                   flagData
                   main_success
                   main_errors
                   defaultDisplayHelp
```

As you can see, the real main is replace with a middle man (processMain) that will branch depending on the status. It takes the description of the program (used for the help text), the flagData, a success function callback, a failure function callback and a display help function callback. After parsing the flag it will call one of these three functions depending on the scenario mentioned above.

6.4 March 27, 2014

6.4.1 Cleanup of internal constraints representations

At first several constraint representation existed for similar constraints. This is the case for *optional flags* and *conditionally required flags*. So for instance this is the internal data structure and creator methods for these features:

```
data FlagConf a =
    FlagConf_IsOptional
  | FlagConf_RequiredIf (FlagResults -> Bool)
  | ...

isOptional :: FlagConf (Maybe a)
isOptional = FlagConf_IsOptional

requiredIf :: (FlagResults -> Bool) -> [FlagConf (Maybe a)]
requiredIf predicate = [isOptional, FlagConf_RequiredIf predicate]
```

But one implies the other, if a flag is required if some predicate holds, then this flag is optional. At first this was solved by composing the two using the constraint combinator. But then they were merged into a single one (isOptional was merged into requiredIf) like this:

```
data FlagConf a =
  FlagConf_RequiredIf (FlagResults -> Bool)
  | ...

isOptional :: FlagConf (Maybe a)
isOptional = requiredIf (const False)

requiredIf :: (FlagResults -> Bool) -> FlagConf (Maybe a)
requiredIf = FlagConf_RequiredIf
```

Basically if a flag is optional then it is marked as requiredIf with a lambda expression that always returns false. This way there is code reuse.

The same was done for defaultIf and defaultIs. The latter just sets a default value to a flag. The first, defaultIf, sets a default to a flag if a predicate returns true. So defaultIs just calls defaultIf with a predicate that always returns True:

```
defaultIs :: a -> FlagConf a
defaultIs a = FlagConf_DefaultIf a (const True)

defaultIf :: a -> (FlagResults -> Bool) -> FlagConf a
defaultIf = FlagConf_DefaultIf
```

As we see this produces the same result as with isOptional, duplicate structures are eliminated and the same code will handle both scenarios.

7 Resources

The following resources contain the most updated information of the library to the date, ranging from source code documentation to an user's guide.

7.1 Source code url

The library was made open source and the source code is available via Github at this address:

- `https://github.com/josercruz01/hsoptions`

7.2 Package url

The library was packaged and published into Hackage. This made the library available to all *cabal* users. It is a resource that shows all the versions of the application and all the documentation history for each version.

- `https://github.com/josercruz01/hsoptions`

7.3 Getting started guide url

User's guide showing all the features of the library and examples.

- `https://github.com/josercruz01/hsoptions#table-of-contents`

7.4 Source code documentation url

- `http://hackage.haskell.org/package/hsoptions`

8 Future work

- Dependency tree between flags

  Since there are features that denotes dependencies between flags, such as *dependent default* value of a flag, a way to express this dependency tree between the flags will be helpful.

  If flag A's default value depends on flag B's default value, which in terms depends on flag C's value, then when getting the default value for flag A we need to make sure that flag B's value was already processed, otherwise we will find a problem with the flag A's value.

- Analyze common usages and create general functions.

  After a fair amount of use of the application, common patterns can be noted and custom functions should be created to facilitate these patterns. An example of this is the *boolFlag* method. Which is a combination of three flag configurations (*boolParser, emptyValueIs True, defaultValueIs False*. Another examples that might be implemented is a validation method that denotes exclusiveness between two or more flags such as `flagsAreExclusives [debugFlag, profileFlag, runFlag]`. This is currently implemented by using the global validation function *validate*, but the exclusiveness check is pushed to the user as he is the one that has to specify the predicate lambda expression of the validate function.

- Debug mode.

  Since flag values can be set in multiple scopes, such as command-line, configuration file or a configuration file included by another configuration file, it can be seen that in a complex scenario one could set or overwrite the value of a flag and have a hard time finding where this flag value is coming from.

  A debug mode will allow the *HsOptions* library to display a tree of all the flags, next to all the history of values of the flag and where each of this value came from. Something similar to this:

```
userName
--> "john" : command-line
--> "mike" : command-line
--> "josh" : conf-file= file1.conf
--> "custom-mike" : conf-file= file2.conf
```

  In this scenario we can figure out that in the command-line input stream the userName was set to *john* then overwritten to *mike*, then the configuration file *file1.conf* was included which in turn overwrote the value to *josh* and that finally a second configuration file was included that set the value to *custom-mike*0.

  This debug functionality is expected to save a lot of time in debugging.

- Heterogeneous collection for cache — Speed improvements

  At the moment every time you call the *get* method to get a value for a flag the entire logic of checking if the flag was provided, validation and parsing are executed. This is fine in the perspective of functional programming, as no state is being modified this is proven to always return the same value. But the problem is that this flow is executed over and over again.

  The basic problem behind this is that Haskell does not provided an easy way to hold a collection of different objects in the same array. Since all flag will have different types then the easiest solution is to store the string value sent by the user and then parse this value.

  Haskell Heterogeneous collections seems to be a possible solution to this problem. All flag values should be pre-cached in this collection and every time the *get* method is called this value will not have to be computed again.

- Mark flag as valueless

  At the moment, for a parsed flag, the parser will always assign the next token in the input stream as the value of this flag, unless the next token is another flag. Sometimes it makes sense to mark a flag as valueless so that the parser does not parse the next argument on the input stream as the value. For example it would be desired that this stream runhaskell Prog.hs $--$*debug file1.txt file2 . txt* would interpret the flags *debug* = *True* and args *["file1.txt", "file2.txt"]*, but in the current implementation a *"file1.txt is not a valid value for debug flag"* error is reported to the user.

9 Conclusion

The main focus of this project was to build a medium-to-large sized library in a functional programming language. Haskell was chosen as it is a modern functional programming language that is used widely and provides a lot of neat features.

The chosen project was a command-line processing library for Haskell. This project was built successfully from zero and all requirements were achieved. One major step of this project was the design stage. A solid design, for the features API and the project's infrastructure, had to be established so that it could support all required functionality in a concise manner. A log of all design decisions and changes was created.

The project was published publicly to the Haskell community so that other developers can take advantage and make use of it. It is expected this will contribute to the project as feedback will be gathered from real life application usage.

References

[1] G. R. Mihaly Barasz, "hflags: Command line flag parser, very similar to google's gflags." `http://hackage.haskell.org/package/hflags`, 2013. [Online; accessed 02-May-2014].

[2] J. Millikin, "options: A powerful and easy-to-use command-line option parser." `http://hackage.haskell.org/package/options`, 2013. [Online; accessed 02-May-2014].