Auca Source Manual

Linus Arver[*†]

0.1.2-0-g7424b49[§]

2014-02-17 12:08:50 -0800

Contents

1 Introduction **1**
1.1 How to Read This Manual 1

2 auca.lhs **2**

3 AUCA/Option.lhs **3**

4 AUCA/Core.lhs **5**
4.1 Event Handling 5
4.2 Key Handling 6

5 AUCA/Util.lhs **8**

6 AUCA/Meta.lhs **9**

1 Introduction

`auca` is a program that automatically executes an arbitrary command based on the modification of a file or set of files.

1.1 How to Read This Manual

The general format is to show the raw source code first, followed by commentary on what the just-shown block of code does. The idea is to try to read the source code first, and then have it explained in detail later. Whenever the commentary says "this block of code" or "here", it is referring to the block of code directly above it.

[*]Email: `X@Y.Z`, where Z is `edu`, Y is `ucla`, and X is `linus`.
[†]Website: `http://listx.github.io`.
[§]This document is generated from the sources from the latest commit. The full hash of this commit is `7424b4991436e44edafc144e90efc42681d6a8f1`.

2 auca.lhs

```
{-# LANGUAGE PackageImports #-}
{-# LANGUAGE RecordWildCards #-}

module Main where

import "monads-tf" Control.Monad.State
import Data.List (nub)
import System.IO
import System.Directory
import System.Environment
import System.Exit
import System.INotify

import AUCA.Core
import AUCA.Option
import AUCA.Util

main :: IO ()
main = do
    hSetBuffering stdout NoBuffering
    hSetBuffering stderr NoBuffering
    args' <- getArgs
    opts@Opts{..} <- (if null args' then withArgs ["--help"] else id) $ getOpts
    errNo <- argsCheck opts
    when (errNo > 0) $ exitWith $ ExitFailure errNo
    files <- if null list
        then return []
        else return . nub . filter (not . null) . lines =<< readFile list
    fs <- mapM doesFileExist file -- e.g., --file x --file y --file z
    -- e.g., --list x (and files defined in file x)
    flist <- mapM doesFileExist files
    errNo' <- filesCheck fs flist
    when (errNo' > 0) $ exitWith $ ExitFailure errNo
    let filesMaster = nub $ file ++ files
    helpMsg opts (head filesMaster)
    prog opts filesMaster
```

`main` checks for various errors before passing control over to `prog`.

```
argsCheck :: Opts -> IO Int
argsCheck Opts{..}
    | null command && null command_simple
        = errMsgNum "--command or --command-simple must be defined" 1
    | null file && null list = errMsgNum "either --file or --list must be defined" 1
    | otherwise = return 0
```

`argsCheck` rejects any obviously illegal arguments.

```
-- Verify that the --file and --list arguments actually make sense.
filesCheck :: [Bool] -> [Bool] -> IO Int
filesCheck fs flist
    | any (==False) fs = errMsgNum "an argument to --file does not exist" 1
    | any (==False) flist = errMsgNum "a file defined in --list does not exist" 1
    | otherwise = return 0
```

`filesCheck` makes sure that all files defined by the user actually exist in the filesystem.

```
prog :: Opts -> [FilePath] -> IO ()
prog opts@Opts{..} filesToWatch = do
    let
        comDef = if null command_simple
            then (head command)
            else command_simple ++ " " ++ (head filesToWatch)
        tb = TimeBuffer
            { bufSeconds = fromIntegral buffer_seconds
            , bufSecStockpile = 0
            }
    inotify <- initINotify
    putStrLn "\nFiles to watch:\n"
    mapM_ putStrLn filesToWatch
    mapM_ (\f -> addWD inotify f (eventHandler comDef f inotify)) filesToWatch
    hSetBuffering stdin NoBuffering
    hSetEcho stdin False -- disable terminal echo
    evalStateT
        (keyHandler opts comDef (head filesToWatch) inotify)
        tb
```

`prog` initializes the `inotify` API provided by the Linux kernel. We simply tell the API to check for any file modifications on the list of files in `filesToWatch`, with the `addWD` helper function defined in `AUCA.Core`. We then move on and enter into `keyHandler`, a simple loop that checks for manual key presses by the user. The calls to disable buffering on STDIN allow `keyHandler` to detect individual key presses at a time.

3 AUCA/Option.lhs

```
{-# LANGUAGE DeriveDataTypeable #-}
{-# LANGUAGE RecordWildCards #-}

module AUCA.Option where

import System.Console.CmdArgs.Implicit

import AUCA.Meta
import AUCA.Util

data Opts = Opts
    { command :: [String]
```

```
    , command_simple :: String
    , file :: [FilePath]
    , list :: FilePath
    , buffer_seconds :: Int
    } deriving (Data, Typeable, Show, Eq)

progOpts :: Opts
progOpts = Opts
    { command = def &= typ "COMMAND"
        &= help "command(s) to execute; up to 10 (hotkeyed to 1-0)"
    , command_simple = def &= typ "COMMAND" &= name "C"
        &= help "command to execute; it takes the first file, and calls command after\
            \ it; e.g., `-C lilypond -f foo.ly' will translate to `lilypond foo.ly'\
            \ as the default command"
    , file = def
        &= help "file(s) to watch; can be repeated multiple times to define multiple\
            \ files"
    , list = def
        &= help "list of files to watch"
    , buffer_seconds = 1
        &= help "minimum interval of seconds to process file changes/keystrokes"
    }
    &= details
        [ "Notes:"
        , ""
        , "  All commands are passed to the default shell."
        ]
```

`progOpts` is the data structure that actually defines all options and also describes their help messages.

```
getOpts :: IO Opts
getOpts = cmdArgs $ progOpts
    &= summary (_PROGRAM_INFO ++ ", " ++ _COPYRIGHT)
    &= program _PROGRAM_NAME
    &= help _PROGRAM_DESC
    &= helpArg [explicit, name "help", name "h"]
    &= versionArg [explicit, name "version", name "v", summary _PROGRAM_INFO]
```

`getOpts` is the custom IO action that gets the options from the environment. It also explicitly sets the '`-h`' and '`-v`' flags, to override the ones given by `CmdArgs` (which define '`-?`' as `--help` and '`-v`' as '`--verbose`').

```
helpMsg :: Opts -> FilePath -> IO ()
helpMsg Opts{..} f = do
    mapM_ showCom $ if null command
        then [("1", command_simple ++ " " ++ f)]
        else zip (map show [(1::Int)..10]) command
    putStrLn "press `h' for help"
    putStrLn "press `q' to quit"
```

```
    putStrLn $ "press any other key to execute the default command " ++
        squote (colorize Blue comDef)
    where
    showCom :: (String, String) -> IO ()
    showCom (a, b) = putStrLn $ "key "
        ++ squote (colorize Yellow a)
        ++ " set to "
        ++ squote (colorize Blue b)
    comDef = if null command
        then command_simple ++ " " ++ f
        else head command
```

helpMsg is the function that gets called if the user requests for help interactively by pressing the '**h**' key. It is also displayed on startup.

4 AUCA/Core.lhs

There are two main functions here — **eventHandler** and **keyHandler**. **eventHandler** hooks into the **inotify** API for executing arbitrary commands, and **keyHandler** handles all interactive key presses by the user.

```
{-# LANGUAGE PackageImports #-}
{-# LANGUAGE RecordWildCards #-}

module AUCA.Core where

import Control.Monad
import "monads-tf" Control.Monad.State
import Data.Time.Clock
import System.Exit
import System.INotify
import System.Process

import AUCA.Option
import AUCA.Util
```

4.1 Event Handling

```
eventHandler :: String -> FilePath -> INotify -> Event -> IO ()
eventHandler comDef fp inotify ev = case ev of
    Attributes{..} -> runCom'
    Modified{..} -> runCom'
    Ignored -> runCom'
    DeletedSelf -> do
        _ <- addWD inotify fp (eventHandler comDef fp inotify)
        return ()
    _ -> showInfo
    where
```

```
    showInfo = putStrLn ("File: " ++ fp ++ " Event: " ++ show ev)
    runCom' = do
        putStrLn []
        showTime
        putStr $ ": " ++ colorize Magenta "change detected on file " ++ squote fp
        putStrLn $ "; executing command " ++ squote (colorize Blue comDef)
        runCom $ cmd comDef
```

We only execute the given command when the detected event is a *modification* event of a `file`. We ignore all other types of events, but print out info messages to tell the user what happened. If a file becomes ignored or deleted for some reason, we re-watch it.[1]

```
addWD :: INotify -> FilePath -> (Event -> IO ()) -> IO WatchDescriptor
addWD inotify fp evHandler = addWatch inotify evs fp evHandler
    where
    evs = [Attrib, Modify, DeleteSelf]
```

`addWD` is a simple wrapper function around the more general `addWatch` function provided by `System.INotify`.

4.2 Key Handling

```
data TimeBuffer = TimeBuffer
    { bufSeconds :: NominalDiffTime
    , bufSecStockpile :: NominalDiffTime
    }
```

The keypresses are interpreted through a buffer system. Essentially, this system works to prevent spamming the `keyHandler` loop. I.e., if a user presses and *holds down* a key, without a buffering system, the loop would execute the total number of keypresses that the windowing system would allow. Even with a modest delay between keypresses, allowing such a torrent of repeated keypresses to go through unabated would be undesirable. Thus, `keyHandler` measures the amount of time taken to process a keypress, and adds it to the buffer, called `bufSecStockpile`. If this stockpile adds up to the treshhold defined by `bufSeconds`, we execute the latest keypress; otherwise, we add the amount taken by the single keypress and add it to the stockpile.

Note that if the user waits a long time, that's fine as the `getChar` function will take that much longer to finish extracting the keypress.

```
keyHandler :: Opts -> String -> FilePath -> INotify -> StateT TimeBuffer IO ()
keyHandler o@Opts{..} comDef f inotify = do
    t1 <- lift getCurrentTime
    c <- lift getChar
    when (c == 'q') . lift $ do
        killINotify inotify
        exitSuccess
    tb@TimeBuffer{..} <- get
    t2 <- lift getCurrentTime
```

[1] Vim tends to delete and re-create files when saving a modification.

```
    let
        t3 = diffUTCTime t2 t1
        stockpile = t3 + bufSecStockpile
    if (stockpile >= bufSeconds)
        then do
            put $ tb { bufSecStockpile = stockpile - bufSeconds }
            keyHandler' c
        else do
            put $ tb { bufSecStockpile = stockpile + t3 }
            keyHandler o comDef f inotify
    where
    keyHandler' 'h' = do
        lift $ helpMsg o f
        keyHandler o comDef f inotify
    keyHandler' 'q' = do
        lift $ putStrLn []
        lift $ killINotify inotify
    keyHandler' key = do
        if elem key comKeys
            then case lookup [key] comHash of
                Just com -> do
                    lift $ putStrLn []
                    lift $ showTime
                    lift . putStr $ ": "
                        ++ colorize Cyan "manual override"
                        ++ " (slot "
                        ++ colorize Yellow [key]
                        ++ ")"
                    lift . putStrLn $ "; executing command "
                        ++ squote (colorize Blue com)
                    lift . runCom $ cmd com
                _ -> do
                    lift $ putStrLn []
                    lift . putStrLn $ "command slot for key "
                        ++ squote (colorize Yellow [key]) ++ " is empty"
            else do
                lift $ putStrLn []
                lift showTime
                lift . putStr $ ": " ++ colorize Cyan "manual override"
                lift . putStrLn $ "; executing command "
                    ++ squote (colorize Blue comDef)
                lift . runCom $ cmd comDef
        keyHandler o comDef f inotify
    comHash :: [(String, String)]
    comHash = if null command
        then [("1", command_simple ++ " " ++ f)]
        else zip (map show [(1::Int)..10]) command
```

```
    comKeys :: String
    comKeys = concatMap show [(0::Int)..9]
```

The `comHash` and `comKeys` structures define the hotkeys available to the user if multiple commands were defined.

```
runCom :: CreateProcess -> IO ()
runCom com = do
    (_, _, _, p) <- createProcess com
    exitStatus <- waitForProcess p
    showTime
    putStrLn $ ": " ++ if (exitStatus == ExitSuccess)
        then colorize Green "command executed successfully"
        else colorize Red "command failed"

cmd :: String -> CreateProcess
cmd com = CreateProcess
    { cmdspec = ShellCommand $
        (com ++ " 2>&1 | sed \"s/^/  " ++ colorize Cyan ">" ++ " /\"")
    , cwd = Nothing
    , env = Nothing
    , std_in = CreatePipe
    , std_out = Inherit
    , std_err = Inherit
    , close_fds = True
    , create_group = False
    }
```

`runCom` and `cmd` are the actual workhorses that spawn the external command defined by the user. The output of the external command is colorized using the `sed` stream editor.

5 AUCA/Util.lhs

```
module AUCA.Util where

import Data.Time.LocalTime
import System.IO

data Color
    = Red
    | Green
    | Yellow
    | Blue
    | Magenta
    | Cyan
    deriving (Show, Eq)

colorize :: Color -> String -> String
colorize c s = c' ++ s ++ e
```

```
    where
    c' = "\x1b[" ++ case c of
        Red -> "1;31m"
        Green -> "1;32m"
        Yellow -> "1;33m"
        Blue -> "1;34m"
        Magenta -> "1;35m"
        Cyan -> "1;36m"
    e = "\x1b[0m"
```

`colorize` adds special ANSI escape sequences to colorize text for output in a terminal.

```
errMsg :: String -> IO ()
errMsg msg = hPutStrLn stderr $ "error: " ++ msg

errMsgNum :: String -> Int -> IO Int
errMsgNum str num = errMsg str >> return num
```

`errMsg` and `errMsgNum` are helper functions to ease reporting simple errors.

```
squote :: String -> String
squote s = "`" ++ s ++ "'"

showTime :: IO ()
showTime = getZonedTime >>= putStr . show
```

`squote` quotes a string with single quotes. `showTime` displays the current local zoned time.

6 AUCA/Meta.lhs

This module mainly defines the metadata that comes with `auca`. Of particular note here is the version number definition.

```
module AUCA.Meta where

_PROGRAM_NAME
    , _PROGRAM_VERSION
    , _PROGRAM_INFO
    , _PROGRAM_DESC
    , _COPYRIGHT :: String
_PROGRAM_NAME = "auca"
_PROGRAM_VERSION = "0.1.2"
_PROGRAM_INFO = _PROGRAM_NAME ++ " version " ++ _PROGRAM_VERSION
_PROGRAM_DESC = "execute arbitrary command(s) based on file changes"
_COPYRIGHT = "(C) Linus Arver 2011-2014"
```