Auca Source Manual

Linus Arver

`0.0.1.4-0-gfa4e4dc`

`2014-09-05 11:11:06 -0700`

Contents

1 Introduction

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

2 auca.lhs

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

module Main where
```

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 `fa4e4dcda7d3cb666e8c731357be8df9c30566e8`.

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

```
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
```

argsCheck rejects any obviously illegal arguments.

```
argsCheck :: Opts -> IO Int
argsCheck Opts{..}
    | null commands && 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
```

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

```
-- 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
```

`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.

```
prog :: Opts -> [FilePath] -> IO ()
prog opts@Opts{..} filesToWatch = do
    let
        comDef = if null command_simple
            then (head commands)
            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
    let
        appState = AppState
            { timeBuffer = tb
            , comDef = comDef
            , comSimpleFilePath = head filesToWatch
            , inotify = inotify
            , opts = opts
            }
    evalStateT keyHandler appState
```

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

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

```
progOpts :: Opts
progOpts = Opts
    { commands = def &= typ "COMMAND(S)"
        &= help "command(s) to execute; up to 10 (hotkeyed to 1-0)"
    , command_simple = def &= typ "COMMAND" &= name "C"
        &= help (unwords
            [ "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 (unwords
            [ "file(s) to watch; can be repeated multiple times to define"
            , "multiple files"
            ])
    , list = def
        &= help "list of files to watch"
    , buffer_seconds = 2
        &= help "minimum interval of seconds to process file changes/keystrokes"
    }
    &= details
        [ "Notes:"
        , ""
        , "  All commands are passed to the default shell."
        ]
```

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**').

```
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]
```

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.

```
helpMsg :: Opts -> FilePath -> IO ()
helpMsg Opts{..} f = do
    mapM_ showCom $ if null commands
        then [("0", command_simple ++ " " ++ f)]
        else zip (map show [(0::Int)..9]) commands
    putStrLn "press `h' for help"
    putStrLn "press `q' to quit"
    putStrLn $ unwords
        [ "press `d' to set the default command to another one from the"
        , "command slot"
        ]
    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 commands
        then command_simple ++ " " ++ f
        else head commands
```

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
```

```
data AppState = AppState
    { timeBuffer :: TimeBuffer
    , comDef :: String
    , comSimpleFilePath :: FilePath
    , inotify :: INotify
    , opts :: Opts
    }
```

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

4.1 Event Handling

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]

```
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
```

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

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

4.2 Key Handling

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, with-

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

out 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 :: StateT AppState IO ()
keyHandler = do
    appState@AppState{..} <- get
    t1 <- lift getCurrentTime
    c <- lift getChar
    when (c == 'q') . lift $ do
        killINotify inotify
        exitSuccess
    let
        tb@TimeBuffer{..} = timeBuffer
    t2 <- lift getCurrentTime
    let
        t3 = diffUTCTime t2 t1
        stockpile = t3 + bufSecStockpile
    if (stockpile >= bufSeconds)
        then do
            let
                tb' = tb {bufSecStockpile = stockpile - bufSeconds}
            put $ appState {timeBuffer = tb'}
            keyHandler' c
            keyHandler
        else do
            let
                tb' = tb {bufSecStockpile = stockpile + t3}
            put $ appState {timeBuffer = tb'}
            keyHandler
```

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

```
keyHandler' :: Char -> StateT AppState IO ()
keyHandler' key
    | key == 'h' = do
        AppState{..} <- get
        lift $ helpMsg opts comSimpleFilePath
    | key == 'd' = do
        appState@AppState{..} <- get
        lift $ helpMsg opts comSimpleFilePath
        lift . putStrLn $ colorize Cyan "swapping default command..."
```

```haskell
        c <- lift getChar
        comHash <- getComHash
        case lookup [c] comHash of
            Just com -> do
                let
                    opts' = opts
                        { commands = swapElems (0, toInt c)
                            $ commands opts
                        }
                put $ appState
                    { comDef = com
                    , opts = opts'
                    }
                lift $ helpMsg opts' comSimpleFilePath
            _ -> do
                lift . putStrLn . colorize Red $ unwords
                    [ "key"
                    , show c
                    , "is not a valid command slot"
                    ]
    | elem key comKeys = do
        AppState{..} <- get
        comHash <- getComHash
        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"
    | otherwise = do
        AppState{..} <- get
        lift $ putStrLn []
        lift showTime
        lift . putStr $ ": " ++ colorize Cyan "manual override"
        lift . putStrLn $ "; executing command "
            ++ squote (colorize Blue comDef)
        lift . runCom $ cmd comDef
    where
    comKeys :: String
    comKeys = concatMap show [(0::Int)..9]
    getComHash = do
        AppState{..} <- get
        let
            coms = commands opts
            comSimple = command_simple opts
        return $ if null coms
            then [("0", comSimple ++ " " ++ comSimpleFilePath)]
            else zip (map show [(0::Int)..9]) coms
```

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.

```
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
    , delegate_ctlc = True
    , env = Nothing
    , std_in = CreatePipe
    , std_out = Inherit
    , std_err = Inherit
    , close_fds = True
    , create_group = False
    }
```

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

```
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"
```

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

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

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

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

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

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

swapElems swaps two elements in a list. It does nothing if any of the arguments are invalid.

```
swapElems :: (Int, Int) -> [a] -> [a]
swapElems (a, b) xs
    | null xs = xs
    | length xs == 1 = xs
    | a < 0 = xs
    | b < 0 = xs
    | a == b = xs
    | a > (length xs - 1) = xs
    | b > (length xs - 1) = xs
    | b < a = swapElems (b, a) xs
    | otherwise = preA
```

```
        ++ [xs!!b]
        ++ betweenAB
        ++ [xs!!a]
        ++ postB
    where
    preA = take a xs
    betweenAB = drop (a + 1) $ take b xs
    postB = drop (b + 1) xs
```

```
toInt :: Char -> Int
toInt c = case c of
    '0' -> 0
    '1' -> 1
    '2' -> 2
    '3' -> 3
    '4' -> 4
    '5' -> 5
    '6' -> 6
    '7' -> 7
    '8' -> 8
    '9' -> 9
    _ -> 0
```

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 = "0.0.1.4"
_PROGRAM_INFO = _PROGRAM_NAME ++ " version " ++ _PROGRAM_VERSION
_PROGRAM_DESC = "execute arbitrary command(s) based on file changes"
_COPYRIGHT = "(C) Linus Arver 2011-2014"
```