Quickterm - Design Decisions

Nils 'bash0r' Jonsson

2016-05-08

Introduction

Each application is run with some basic arguments. These arguments are applied to the program (=application) with help of a command line. The command line is the most basic way to tell a program what to do. Git, for example, uses the command line even as its main user interface. Git uses a lot of integrated commands (like *git commit*) which are then routed to the specified application which implements the real command. The process of routing a command through such a modular software is error-prone and time consuming when written by hand. There are a lot of libraries[1] for routing sub-commands and parsing command line arguments. Some frameworks like Play![2] and Yesod[3] use meta-programming to achieve a routing table for HTTP-requests. A similar pattern could be used with command line arguments as the circumstances are somewhat similar. Both, Play! and Yesod provide a simple DSL for routing from URLs to resources. In Yesod it is possible to go below the DSL and wire an application with help of an EDSL. Both frameworks achieve typesafety with help of meta-programming and the Host languages type inferencer.

I forked the the original *Quickterm*-library from Samuel Schlesinger's repository[4] which is provided under the Gnu GPL v3 license[5]. My rewrite of *Quickterm*-library[6] made the library a collection of embedded domain specific languages (=EDSL) for wiring modular command line software. The processes of routing

[1] To name just a few:

- https://commandline.codeplex.com/
- http://commons.apache.org/proper/commons-cli/
- http://www.boost.org/doc/libs/1_60_0/doc/html/program_options.html
- https://docs.python.org/2/library/argparse.html

[2] https://www.playframework.com/
[3] http://www.yesodweb.com/
[4] https://github.com/SamuelSchlesinger/Quickterm
[5] http://www.gnu.org/licenses/gpl-3.0.en.html
[6] https://github.com/aka-bash0r/Quickterm

commands, marshalling arguments to Haskell types, and generating help messages are all implemented in separate languages and runtimes. Because of that the *Quickterm*-library is itself very modular and extensible. Hooking into the deepest details of *Quickterm* is always possible besides the existence of a comfortable, easy-to-use EDSL. For most applications the EDSL interface should be sufficient which reduces the effort of wiring command line arguments to the application interface enormously. It achieves typesafety by using Haskell's type inferencer.

Design Decisions

Analysing The Input

The basic command line gives the user the ability to run programs. This can be achieved by typing the name of the application. Appending additional text will apply the text to the application as command line arguments. Most shells[7] and consoles[8] preprocess the text before applying the command line arguments to the application.

```
my-program $PATH cmd line arguments
```

The input has some striking similarities over all shells and consoles which makes it possible to form a universal interface. The application receives a list of strings where the concrete type depends on their specific language and runtime. Due to that, shells and consoles form some decent and sometimes very powerful text-preprocessor[9] and lexer[10]. The application receives only a stream of tokens and forms as that a parser for the command line.

Analysing Quickterm

The representation of *Quickterm*-expressions started out as a tree structure[11]. It could be either a $Choice$ $(= (Name, [Quickterm], Usage))$ branch or a $Command$ $(= (Name, TerminalAction, Usage))$ branch. As can be seen below, $TerminalAction$ is a function which returns in fact a lazy, stateful computation - to be specific $IO()$. So the $Command$ represents a computation and $Choice$ is a list of computations.

[7] https://www.gnu.org/software/bash/ http://www.zsh.org/ https://fishshell.com/
[8] https://msdn.microsoft.com/en-us/powershell/mt173057.aspx
[9] https://gcc.gnu.org/onlinedocs/cpp/
[10] https://en.wikipedia.org/wiki/Lexical_analysis
[11] https://github.com/SamuelSchlesinger/Quickterm/blob/7007c40176c28e9ec43ecc39a6b779b0a7569bcc/src/Quickterm.hs#L62

```
type Args = [String]
type Options = Map[String,[String]]
type TerminalAction = Args -> Options -> IO ()
type Name = String
type Usage = String

data Quickterm
  = Choice Name [Quickterm] Usage
  | Command Name TerminalAction Usage
```

The original evaluator of *Quickterm*-expressions was *quickrun*[12]. It needed a lot of helper functions (*findBranch*[13], *whoopsy*[14], *findnearestBranch*[15], and *organizeInput*[16]) to do its work. The result of *quickrun* is a deterministic calculated $IO()$ - in fact another lazy computed computation.

```
findbranch :: String -> [Quickterm] -> Maybe Quickterm
findbranch = ...

whoopsy :: String -> String -> [Quickterm] -> IO ()
whoopsy = ...

findnearestbranch :: String -> [Quickterm]
                            -> Maybe Quickterm
findnearestbranch = ...

organizeInput :: [String] -> (Args, Options)
organizeInput = ...

-- I desugared the function and renamed all variables
-- to make the code more compact.
quickrun :: [String] -> Quickterm -> IO ()
quickrun x q = quickrun' as os q where
  (as,os) = organizeInput x
  quickrun' x y = case (x,y) of
    (_   ,os,Choice  _ _  u) ->
      putStr u
    (as  ,os,Command _ f  _) ->
```

[12] https://github.com/SamuelSchlesinger/Quickterm/blob/7007c40176c28e9ec43ecc39a6b779b0a7569bcc/src/Quickterm.hs#L85

[13] https://github.com/SamuelSchlesinger/Quickterm/blob/7007c40176c28e9ec43ecc39a6b779b0a7569bcc/src/Quickterm.hs#L98

[14] https://github.com/SamuelSchlesinger/Quickterm/blob/7007c40176c28e9ec43ecc39a6b779b0a7569bcc/src/Quickterm.hs#L105

[15] https://github.com/SamuelSchlesinger/Quickterm/blob/7007c40176c28e9ec43ecc39a6b779b0a7569bcc/src/Quickterm.hs#L116

[16] https://github.com/SamuelSchlesinger/Quickterm/blob/7007c40176c28e9ec43ecc39a6b779b0a7569bcc/src/Quickterm.hs#L131

```
    action as os
  (a:as,os,Choice  n bs u) ->
    case findbranch a bs of
      Just b -> quickrun' as os b
      _      -> whoopsy n u bs
```

When looking at the input *git commit –amend* you can see that the first argument in the list is the name of the program which is of no interest to us. We care about the arguments that are postfix to the name of the program. This implies that the first *Choice* does not need a name. Because of this premise we can perform a simple transformation generalizing the tree structure putting the *Name* in relation with *Quickterm* in the *Choice*[17] branch only. Additionally I replaced *Name* with *Predicate*[18] which is far more powerful and I removed the *Options* type from the argument list of *TerminalAction*. This parameter is not needed anymore as I will show later in this document. As a final change I made the function non-deterministic by wrapping the result type in a list.

```
type Predicate = String -> Int
type TerminalAction = [String] -> IO ()
type TermAction = TerminalAction

data Quickterm
  = Choice [(Predicate,Quickterm)] Usage
  | Command TermAction Usage
```

The evaluator[19] for the new data structure is far more general and a lot less complex as the previous definition. Yet it needs a wrapper function *execQuickerm* which handles the result values. In addition it uses a helper function *makeHelp* to generate help messages. The evaluator function was very underdeveloped at this state.

```
makeHelp :: Int -> Int -> Quickterm -> String
makeHelp l i (Choice ps u) =
  if   i >= 0
  then u l ++ "\n" ++ intercalate (ws i ++ "\n") (makeHelp (l+1) (i - 1) . snd <$> ps)
  else u l
makeHelp l _ (Action _ u) = u l

runQuickterm :: Quickterm -> [String] -> [(Int, [String] -> IO ())]
```

[17] https://github.com/aka-bash0r/Quickterm/blob/32a2da0d3bda6e784b5f4d6be565f1661b61044a/src/Quickterm.hs#L19

[18] https://github.com/aka-bash0r/Quickterm/blob/32a2da0d3bda6e784b5f4d6be565f1661b61044a/src/Quickterm.hs#L14

[19] https://github.com/aka-bash0r/Quickterm/blob/32a2da0d3bda6e784b5f4d6be565f1661b61044a/src/Quickterm.hs#L22

```
runQuickterm x y = case (x,y) of
 (Action t  _,[]  ) -> pure (0, t)
 (c@Choice{} ,[]  ) -> pure (0, const . putStrLn $ makeHelp 0 1 c)
 (Action t  _,_   ) -> pure (0, t)
 (Choice ps _,a:as) ->
  ps >>= \(p,q) ->
  runQuickterm q as >>= \(i, io) ->
  return (p a + i, io)

execQuickterm :: Quickterm -> [String] -> IO ()
execQuickterm q as = void . sequence $ (\q -> q as) <$> valid
  where
    rs = runQuickterm q as
    valid = snd <$> filter (\(i,_) -> i == 0) rs
```

The signature of *runQuickrun* shows, that is still produces a computation. This means that the data structure can be generalized even further. There is a state transition from input parameters ($=\alpha$) to output parameters ($=\beta$). This behaviour can be seen when implementing machines like the Turing[20] or Krivine[21] machine. *Choice* and *Command* represent a very high level instruction set for a language running in a typed lambda calculus as the evaluator returns a function in Haskell's typed lambda calculus. So I decided to generalize the machine even further.

```
runQuickterm
  :: Int      -- Punishment Flag; 0 = exact match
  -> Help     -- A program that generates help messages
              -- (e.g. a program for another interpreter)
  -> [String] -- The predicted input (differs from input
              -- arguments when the Punishment Flag is /= 0)
  -> [String] -- The rest of the input arguments
  -> [(a, Int, Help, [String], [String])]
  -- Result 'a':
```

Now *runQuickterm* shows quite clearly the state transition. The machine is non-deterministic that's why it produces list of state βs. It does not produce only one correct result (if there is any), it produces all possible state βs over the whole execution of the program and punishes incorrect states by increasing a punishment flag. A *Quickterm*-program produces therefore all possible result states which can be used to

- detect ambiguous API calls,

[20] https://en.wikipedia.org/wiki/Turing_machine
[21] http://pop-art.inrialpes.fr/~fradet/PDFs/HOSC07.pdf

- implement error correction of inputs,
- and implement help documentation generation.

State β is a tuple of $(a, Int, Help, [String], [String])$ whereas state α is a tuple of $(Int, Help, [String], [String])$. The additional value of type a is the result of a computation during state transition which is a Haskell value. A *Quickrun*-program runs now in a subamount of Haskell's the typed lambda calculus. The semantics of the language are implemented with help of the Haskell typesystem. A lambda is the most powerful representation of data[22]. For every AST there is a representation in lambdas formulating the very same meaning. The *Quickterm*-language does not need any syntax tree anymore as all syntax abstractions can be implemented with help of higher order functions in Haskell now.

The machine itself produces code for other machines running in a subamount of Haskell's typed lambda calculus. In real applications it applies that $a = IO()$ as we are routing command line programs with stateful computation for input and output. After transformation this machine still produces code for execution in the $MonadIOa$ of Haskell. The $Help$ type is $Int- > String$ and forms therefore another language in the lambda calculus of Haskell. Because of that it is possible to compute all possible machines on the fly thanks to Haskell's lazy evaluation which helps a lot to make this code run fast.

The evaluator function *quickterm* of *Quickterm*-expressions can therefore focus on filtering for the correct state β, invoking the help message generator and printing additional help information. Correct state βs have a $punishmentflag = 0$ and have processed all input arguments.

```
quickterm :: Quickterm (IO ()) -> [String] -> IO ()
quickterm qt as = f . filter (\(_, i, _, _, rs) ->
                              i == 0 && rs == []) $ ts
  where
    snd5 :: (a,b,c,d,e) -> b
    snd5 (a,b,c,d,e) = b
    f rs = case rs of
      []                 -> case sortBy (comparing snd5) ts of
        [] -> error "No match could be found."
        ts  ->
          let f i x = case x of
                []                 ->
                  return ()
                ((_,_,_,pi,_):ts') ->
                  putStrLn ("[" ++ show i ++ "] "
                            ++ getPi pi) >> f (i+1) ts'
              getPi = intercalate " " . reverse
```

[22] https://en.wikipedia.org/wiki/Church_encoding

```
          in putStrLn "Could not match arguments to a command:"
             >> putStrLn (">> " ++ intercalate " " as ++ " <<")
             >> putStrLn "Did you mean one of these?"
             >> f 1 (take 10 ts) >> putStr "[0 to quit]: "
             >> getLine >>= \l ->
               if   l =~ "(1|2|3|4|5|6|7|8|9|10)"
               then putStrLn (case ts !! read l of (_,_,h,_,_) -> h 0)
               else return ()
      (r@(a,_,_,_,_):[]) -> a
      (  (_,_,_,_,_):_ ) ->
        error "TODO: generate ambiguous call error message"
    ts = runQuickterm qt 0 (const "") [] as
```

Abstract Syntax Tree

The AST of *Quickterm* is formulated in the lambda calculus of Haskell. The *Quickterm* data type is basically a function. The newtype declaration helps to distinguish between ordinary functions with randomly that signature and *Quickterm*-expressions which makes it possible to define the applicative and monadic interfaces.

```
newtype Quickterm a = Quickterm
  { runQuickterm :: Int
                 -> Help
                 -> [String]
                 -> [String]
                 -> [(a, Int, Help, [String], [String])]
  }

instance Functor Quickterm where
  fmap f m = Quickterm $
    \i h pi as -> (\(a,i',h',pi',as') ->
      (f a,i',h',pi',as')) `fmap` (runQuickterm m i h pi as)

instance Applicative Quickterm where
  pure a = Quickterm $ \i h pi as -> pure (a,i,h,pi,as)
  f <*> m = Quickterm
    $   \i h pi as -> runQuickterm f i h pi as
    >>= \(g,i' ,h' , pi', as' ) -> runQuickterm m i' h' pi' as'
    >>= \(a,i'',h'', pi'',as'') -> return (g a,i'',h'',pi'',as'')

instance Alternative Quickterm where
  empty = Quickterm (const (const (const (const empty))))
  m <|> n = Quickterm
```

```
    $   \i h pi as ->  runQuickterm m i h pi as
    <|> runQuickterm n i h pi as

instance Monad Quickterm where
  return = pure
  m >>= f = Quickterm
    $   \i h pi as -> runQuickterm m i h pi as
    >>= \(a,i',h',pi',as') -> runQuickterm (f a) i' h' pi' as'

instance MonadPlus Quickterm where
  mzero = empty
  mplus = (<|>)
```

The applicative interface isn't pure as it relies on stateful computations on the result list. The definition of state transition implies the existence of stateful computation in the function which implements the state transition. Due to that the applicative interface can only mimic a pure interface to the outer world. The applicative interface is often shorter to write and easier to read.

The EDSL

The language of *Quickterm* is split into several small EDSLs. There are separate languages and runtimes for wiring, marshalling and help-message generation. Wiring the application to input parameters is the main task of the *Quickterm*-library. Thus the languages and runtimes integrate into language and runtime of the wiring task. The steps to achieve combination of different runtimes and languages are explained later in this document. For now, you can think of *Quickterm* for wiring, marshalling and help-message description like of HTML, JavaScript and CSS.

Wiring

The EDSL for wiring focuses on ease of use while preserving the power of the machine. Injection of custom low level commands is possible anywhere and anytime. The EDSL is implemented with the following types and functions in Haskell.

```
data Description = Description
  { nameD :: String
  , longD :: Help
  }

desc :: String -> Description
```

```
desc n = Description n (const "")

section :: Description -> [Quickterm a] -> Quickterm a
section (Description n h) qs = Quickterm $ \i h pi as ->
  let h' = \i -> h i ++ "\n" ++ indent n i
   in case as of
        []      -> qs >>= \m ->
          runQuickterm m (i + 10) h (n:pi) []
        (a:as') -> qs >>= \m ->
          let l = levenshteinDistance defaultEditCosts n a
           in runQuickterm m (i + l) h' (n:pi) as'

param :: (Show a, CanMarshall a) => Quickterm a
param = Quickterm $ \i h pi as -> case as of
  []      -> [(defaultM,i+10,h,pi,[])]
  (a:as') -> deserialize deserializer a 0 >>= \(a, _, i') ->
    return (a, i + i', h, show a:pi, as')

exact :: String -> Quickterm String
exact s = mfilter (==s) param

program :: [Quickterm a] -> Quickterm a
program qs = Quickterm $ \i h pi as -> qs >>= \m ->
  runQuickterm m i h pi as
```

In an applied environment this interface looks like the code below. The exact meaning of all of these commands is explained in detail later in this document.

```
myQtProgram :: Quickterm (IO ())
myQtProgram = program
  [ section (desc "install")
    [ cmdInstall <$> installConfig defaultInstallConfig ]
  , section (desc "sandbox")
    [ exact "init" >>= \_ -> cmdSandboxInit
    , exact "--help" >>= \_ -> cmdSandboxHelp
    , exact "--snapshot" >>= \_ -> cmdSandboxSnapshot
    ]
  , (const cmdHaddock) <$> exact "haddock"
  ]
```

Program

With *program* the user can define the root of a *Quickterm*-application. It contains the basic paths. From this point on the rooting process can start. Entries in this section will be most of the time of kind *section* or *exact*. It is

comparable with the root node of the old *Choice* branch (see chapter **Abstract Syntax Tree**).

```
myQtProgram :: Quickterm (IO ())
myQtProgram = program
  [
  ]
```

Sections

section defines a set of possible paths to go. It is a drop in replacement for the old *Choice* branch (see chapter **Abstract Syntax Tree**). But it's more powerful, though. It defines a common prefix for a set of *Quickterm*-expressions. The prefix is then used to match the exact value in the command line arguments. This kind of declaration is very static and behaves good for software that is structured like **git** or **cabal** with sub-commands. It's a straight forward declaration of code paths. See:

```
myQtProgram :: Quickterm (IO ())
myQtProgram = program
  [ section (desc "install")
    [ cmdInstall <$> installConfig defaultInstallConfig ]
    -- program.exe install

  , section (desc "sandbox")
    [ section (desc "init")
      [ pure cmdSandboxInit ]
      -- program.exe sandbox init

    , section (desc "--help")
      [ pure cmdSandboxHelp ]
      -- program.exe sandbox --help

    , section (desc "--snapshot")
      [ pure cmdSandboxSnapshot ]
      -- program.exe sandbox --snapshot
    ]
  ]
```

Pure

The *pure* function comes from the *Applicative* typeclass. It can be used to inject Haskell values into the runtime of *Quickterm*-expressions. It produces a *Quickterm*-expression containing the injected value.

```
pure 10 :: Quickterm Int
```

Param

Values can be injected straight forward. It's somewhat different for functions. Consider the function $Int -> IO()$. It constructs an $IO()$ from applying a value of type Int. The *param* function can take over marshalling of the argument to Haskell values (which means in fact $(CanMarshala) \Rightarrow String \rightarrow a$. The applicative and monadic interfaces can then be used to apply the marshalled value to the corresponding function.

```
param :: (Show a, CanMarshall a) => Quickterm a
param = Quickterm $ \i h pi as -> case as of
  []      -> [(defaultM,i+10,h,pi,[])]
  (a:as') -> deserialize deserializer a 0 >>= \(a, _, i') ->
    return (a, i + i', h, show a:pi, as')
```

The signature of *param* tells us that it produces a $(Showa, CanMarshalla) \Rightarrow Quickterma$. (For explanation of **CanMarshall a** see **EDSl/Marshalling**.) Haskell's type inferencer can then resolve the deserializer. For marshaling it removes one argument from the token stream (=command line arguments). Because of *param* the user can now

- take tokens from the token stream using a simple wrapper function ($\Rightarrow$ *param*),
- deserialize the very same token from a String-like representation to a real Haskell value without syntactic overhead,
- and apply general Haskell functions to command line arguments without much boilerplate.

The example below shows a very basic calculator demonstrating the use of *param* in combination with the applicative and monadic interface.

```
cmdAdd x y = print $ x + y
cmdSub x y = print $ x - y
cmdMul x y = print $ x * y
cmdDiv x y = print $ x / y

-- Applicative interface:
myQtProgram :: Quickterm (IO ())
myQtProgram = program
  [ section (desc "add") [ cmdAdd <$> param <*> param ]
```

```
  , section (desc "sub") [ cmdSub <$> param <*> param ]
  , section (desc "mul") [ cmdMul <$> param <*> param ]
  , section (desc "div") [ cmdDiv <$> param <*> param ]
  ]

-- Monadic interface:
myQtProgram :: Quickterm (IO ())
myQtProgram = program
  [ section (desc "add") [ param >>= \x ->
                           param >>= \y -> cmdAdd x y ]
  , section (desc "sub") [ param >>= \x ->
                           param >>= \y -> cmdSub x y ]
  , section (desc "mul") [ param >>= \x ->
                           param >>= \y -> cmdMul x y ]
  , section (desc "div") [ param >>= \x ->
                           param >>= \y -> cmdDiv x y ]
  ]
```

The applicative interface allows only pure computations and looks somewhat similar to the usual Haskell call syntax. By that it is - in my opinion - the better choice for plain marshaling and command line parameter application.

Exact

exact is defined as $String -> QuicktermString$. As *exact* is based on *param* it derives the basic semantics.

```
exact :: String -> Quickterm String
exact s = mfilter (==s) param
```

At runtime *exact* will remove one token from the input stream (e.g. it "consumes" one command line argument) and checks if the token (command line argument) is equal to the string given as parameter. A *Quickterm*-expression of type *exact* returns always a static value. It is safe to omit the value which is achieved by applying the function $const =$
$a_- > a$. These laws can be concluded.

```
  (\_ -> cmd) <$> exact s
= (const cmd) <$> exact s

  exact s >>= \_ -> cmd
= exact s >>= const cmd

where cmd: is a value of type IO ()
```

```
      s  : is a string literal containing the exact
           command line argument
```

A feature of *exact* is it's strict evaluation (could change in the future). It removes all cases from the result set where it does not match. This behaviour results from use of $mfilter$ (see declaration in section (**EDSL/Wiring**).

Recursion

Recursion is very important when it comes to considering the power of the EDSL. Command line argument token streams are an ending stream of tokens. Yet we don't know when the input ends. Writing a recursive grammar is the most natural way to overcome this restriction. It is possible to use Haskell for definition of recursion.

```
data InstallConfig
  = InstallConfig
    { bindir :: String
    , docdir :: String
    , datadir :: String
    , builddir :: String
    } deriving (Show, Eq)

defaultInstallConfig :: InstallConfig
defaultInstallConfig = InstallConfig
    { bindir = "/default/bindir"
    , docdir = "/default/docdir"
    , datadir = "/default/datadir"
    , builddir = "/default/builddir"
    }

installConfig :: InstallConfig
              -> Quickterm InstallConfig
installConfig config =
  pure config
  <|> (exact "--bindir" >> param >>= \p ->
        installConfig (config { bindir = p }))
  <|> (exact "--docdir" >> param >>= \p ->
        installConfig (config { docdir = p }))
  <|> (exact "--datadir" >> param >>= \p ->
        installConfig (config { datadir = p }))
  <|> (exact "--builddir" >> param >>= \p ->
        installConfig (config { builddir = p }))
```

The data type $InstallConfig$ contains all possible command line options for the installation process of the application. $defaultInstallConfig$ produces a default value. In $installConfig$ we define a recursive parameter parser. Each rule (except the first) defines the setting of another rule. The rules start with a flag name indicating the kindness of the following parameter which is then put into the configuration container $InstallConfig$. The tail recursive call to the Haskell-function $installConfig$ forms the recursion in the machine.

Marshalling

Marshaling begins with the type inference of Haskell's typechecker which happens on a use of *param* (see chapter **EDSL/Wiring/Param**). The *param* function then calls the *deserializer* which triggers the typeclass $CanMarshall$.

```
class CanMarshall a where
  defaultM :: a
  helpU :: a -> Int -> String
  deserializer :: Deserializer a
```

Datatypes that are used in the marshalling process require a typeclass instance of typeclass $CanMarshall$. In most cases marshalling of command line arguments is a primitive transformation from $String$ to something like Int. These cases can inject the basic mechanism in a gentle way to run in decent performance. The *Quickterm*-library provides some instances for basic types which use regular expressions ($Text.Regex.Base$ and $Text.Regex.TDFA$). More complex cases can either inject the process as well (like calling a JSON deserializer) or use the EDSL for deserialization.

```
tryConvert :: (String -> [(a,Int)]) -> Deserializer a
tryConvert f = Deserializer $ \st p -> (\(a,i) -> (a,[],p+i)) <$> f st

instance CanMarshall Int where
  defaultM = 0
  helpU _ = indent "<Integer>"
  deserializer = tryConvert $ \st ->
    if   st =~ "((0|1|2|3|4|5|6|7|8|9)+)"
    then [(read $ st,0)]
    else [(0,length st * 2)]

instance CanMarshall String where
  defaultM = "str"
  helpU _ = indent "<String>"
  deserializer = tryConvert $ \st ->
    if   st =~ "([^-]+)"
```

```
    then [(st,0)]
    else [("str",length st * 2)]
```

In both instance implementations the deserializer is injected with help of *tryConvert*. The semantics of *tryConvert* say that it processes the whole remaining input at once. The *Int* is used as punishment flag for further reasoning about the quality of the conversion. The *Deserializer* provides an applicative and monadic interface and forms itself a recursive descent lexer combinator without a beautiful EDSL for now[23]. Syntax for the outstanding EDSL can be imagined like something between *Parsec*[24] and *uu-parsinglib*[25].

Help-Description

The EDSL for help-messages is currently the most underdeveloped language in the *Quickterm*-library. It consists of a single function, *indent* to name it. The *indent* function adjusts the padding of a text block based on the current call level.

```
type Help = Int -> String

ws :: Int -> String
ws l = replicate (l*2) ' '

indent :: String -> Int -> String
indent a i = intercalate "\n" ((ws i ++) <$> splitLn a)

splitLn :: String -> [String]
splitLn = f []
  where
    f rs []        = [reverse rs]
    f rs ('\n':ss) = reverse rs : f [] ss
    f rs (s:ss)    = f (s:rs) ss
```

The machine behind the help-message generator is the type $Help = Int- > String$. So the machine transforms an input state of type *Int* to a result type *String*. Further development of the EDSL could result in something similar to Markdown[26].

[23] I just hadn't time yet to implement it.

[24] https://wiki.haskell.org/Parsec

[25] https://hackage.haskell.org/package/uu-parsinglib-2.9.1.1/docs/Text-ParserCombinators-UU-Demo-Examples.html

[26] https://en.wikipedia.org/wiki/Markdown

Conclusion

The transformation has clearly shown that the construct is a machine similar to the Krivine Machine running in a subamount of Haskell's typed lambda calculus. Additionally, the analysis of the input language shows that there are some relations between parser combinators and the implemented machine. The shell or console used by the user form a very powerful text-preprocessor and lexer whereas the application represents the parser. A program that uses the command line arguments as instructions forms therefore an interpreter.

Also, it could be shown that generalization of the system improves power of the system. Adding syntactic sugar on top with help of a DSLs or EDSLs makes the use of those libraries very easy. On further development of the library it would be possible to implement a real programming language which handles the ugly things of command line argument parsing and marshalling, error correction of inputs and help message generation.