Haskal - a Haskell shell

Mats Jansborg and Aleksandar Despotoski

May 18, 2006

Why a new type of Shell?

- Limited functionality
- No type safety

Unix shells as Monads

- Where | corresponds to `>>=`
- `cat x` to `return x`

Related Work

- H4sh
  - Haskell functions as Programs
  - http://www.cse.unsw.edu.au/ dons/h4sh.html
- Hashell
  - Uses combining of Haskell commands and shell commands
  - http://haskell.org/hashell

General Approach

- Make System functions available as Haskell functions

Example

```
cat :: Program String String
```

- Define operators to compose programs and Haskell functions

- Example

  >|< corresponds to the normal pipe in Shells

General Approach

- Importing the programs
  - Go through the path
  - Compile the file with all programs (to Object file)
- Read User Input
- Compile and link together with programs and operators

HS-Plugins

- Used for executing Haskell code dynamically
- Compiling, linking and running is done by hs-plugins
- Works fine for compiling program files, but rather slow for user input.

How to model commands?

- Using the monad analogy:

```
instance Monad command where ...

echo :: [Arg] -> String -> Command String
cat  :: [Arg] -> a -> Command a
echo [] "foo" >>= cat [] :: Command String
```

- Problem: we need to (de)serialise inputs and outputs

Add support for serialisation

-
  ```
  class Marshal a where
    marshal       :: a -> ([Word8] -> [Word8])
    unmarshal     :: [Word8] -> a
    marshalList   :: [a] -> ([Word8] -> [Word8])
    unmarshalList :: [Word8] -> [a]
  ```

Commands in Haskal

-
  ```
  newtype Command i o = Command [IO ()]

  class Cmd c where
    toCommand :: (Marshal i, Marshal o) =>
                        c i o -> Command i o
  ```

-
  ```
  instance Cmd Command where ...
  instance Cmd (->) where ...
  instance Cmd Program where ..
  ```

-
  ```
  newtype DoIO i o = DoIO (i -> IO o)
  instance Cmd DoIO where ...
  ```

Redirection

- Combining commands in parallel

```
(>|<) :: (Cmd c1, Cmd c2, Marshal t,
          Marshal i, Marshal o) =>
            c1 i t -> c2 t o -> Command i o
```

- Redirecting standard output, input and error

```
(>|), (|<), (&>|)  :: (Marshal i, Marshal o, Cmd c) =>
                      c i o -> String -> Command i o
```

Example

```
$ls >|< words >|< map length >|< sum
1228

$ls >|< map (dropWhile (/='.')) . words >| "file"
$cat |< "file"
.cabal
.hs
...
```

Name clashes

- `$ls >|< words >|< map length >|< sum`

  ```
  <haskal>:1:32:
      Ambiguous occurrence ‘sum’
      It could refer to either ‘Data.List.sum’,
      imported from Prelude at
      Implicit import declaration
      or ‘P.sum’, imported from P at
  /tmp/MeZeX10978.hs:2:0-8
  ```
- Need to use qualified names

  ```
  $ls >|< words >|< map length >|< Prelude.sum
  1228
  ```

Command line arguments

- Argument is a typeclass too

```
class Argument a where
  toArgument      :: a -> [String]
  listToArgument :: [a] -> [String]
```

-
  ```
  instance Argument String ...
  instance Arguemnt (Program i o) ...
  instance Argument Int ...
  ...
  instance Argument a => Argument [a] ...
  ```

- Add options to programs with

```
(-.) :: Argument a => Program i o -> a -> Program i o
(#)  :: Argument a => Program i o -> a -> Program i o
```

What we'd really want

- More precise types:

  `ls :: Program i [File]`

  - Problem: argument changes the type

    `ls -."l" :: Program i [FileDetails]`
  - We need dependent types for this
- Nicer syntax, writing e.g.

  `ssh -."l" #"jansborg" #"remote.mdstud.chalmers.se"`

  is really annoying for interactive use.

Other features implemented

- Job control.
- Tab completion of program names and files.
- Commands to haskal are prefixed with ':'
    - `:cd` changes directory
    - `:background`, `:foreground` and `:jobs` deals with job control.
    - `:load` imports extra modules into the command line session
    - `:rehash` regenerate the module with the program bindings
    - `:typeOf` gives the type of an expression. Works only for monomorphic types.
    - `:which` gives the path to a program.

Non-features

- Any kind of error handling mechanism.
- Exit codes or similar.
- A way of defining functions interactively or by sourcing a file.

Conclusions

- > *Although most users think of the shell as an interactive command interpreter, it is really a programming language in which each statement runs a command. Because it must satisfy both the interactive and programming aspects of command execution, it is a strange language, shaped as much by history as by design.*

  – Brian Kernighan Rob Pike 1984

  - A Haskell shell is probably only usable for scripting.
  - Rewrite using GHC API.