HsDep: Dependency Graph Generator for Haskell

Wolfram Kahl

Software Quality Research Laboratory, McMaster University

2004-06-21

This is a wrapper around the dependency generation facilities of GHC that produce a dot graph for module dependencies.

Usage:

`HsDep [--excludes="modules ..."] graphname GHCoptions ... files ...`

This calculates a dependency graph among the Haskell modules contained in *files* (assuming usual Haskell file naming conventions), except that dependencies to any of the modules listed in the "`--excludes`" argument are omitted (quotes are necessary only for protecting space-separated multiple *modules* against interference from the shell). The dot representation of the resulting dependency graph is saved as *graphname*`.dot`, and dot is invoked to convert it into the PostScript file *graphname*`.ps` All the *GHCoptions* ... and *files* ... are passed unchanged to the dependency generation invokation of GHC.

The remainder of this document is the literate Haskell implementation code.

```
import System   -- Haskell 98 imports
import IO
import List (partition, isPrefixOf)
import Dot      -- companion module
```

Since full-fledged command-line parsing would require careful distinction of HsDep options from GHC options, we only extract the `--excludes` option in a simple way.

```
exclopt = "--excludes="
```

After that, we assume that the first remaining argument is the dot graph name.

The implementation is presented top-down:

```
main = do
  (excl, name : args) ← fmap (partition (exclopt`isPrefixOf`)) getArgs
  let dotfile = name ++ ".dot"
      psfile = name ++ ".ps"
      depfile = name ++ ".depend"
      excludes = concatMap (words ∘ drop (length exclopt)) excl
  putStrLn ∘ unwords $ excludes
```

Since it seems to be impossible to have "ghc -M" output the generated dependencies to *stdout*, we need to save them to a file; we use *graphname*`.depend` for this purpose and do not to remove it after processing it, in case it may be needed also for other purpses.

Then we write the dependencies into that temporary file, and parse the file contents into dependency pairs:

```
  system $ unwords (mkdepCommand depfile excludes : args)
  deps ← fmap parseDepFile $ readFile depfile
```

The dependencies are then output as dot graph, and `dot` is invoked to convert to PostScript.

```
  writeFile dotfile ∘ show ∘ dotOfDeps name $ deps
  system $ unwords ["dot -Tps ", dotfile, ">", psfile]
```

This finishes the definition of *main*.

For writing dependencies into a given file path, currently the following invocation is necessary:

```
mkdepCommand depfile excludes =
  unwords ∘ ("ghc -M":) ∘ map ("-optdep"++) ∘
    ("-f":) ∘ (depfile:) ∘ map ("--exclude-module="++) $ excludes
```

We represent a module dependency as a pair of module names, encoded as strings:

```
type ModDep = (String, String)
```

For parsing dependency files, we filter out comments first, expect all remaining lines to be file dependencies, which are converted into module dependencies by *depFromLine*. We are only interested in the irreflexive part of this relation, and therefore filter out dependencies arising from "`modname.o : modname.lhs`" lines.

```
parseDepFile :: String → [ModDep]
parseDepFile = filter (uncurry (≢)) ∘ map depFromLine ∘ filter noComment ∘ lines
```

Comments begin with the hash character, we also consider empty lines as comments.

```
noComment []       = False
noComment (c : cs) = c ≢ '#'
```

For obtaining module dependencies from file dependencies, we need to drop the suffixes (here `.lhs`, `.hs`, `.hi`, `.o`) from both elements. Furthermore, GHC in many cases lists dependencies from the current directory prefixed with "`./`", as well as dependancies from relative directories prefixed with "`.../directory/to/file/`", so we use *dropPrefix* to eliminate those, too:

```
depFromLine l = case words l of
  [ofile, ":", depfile] → ((dropSuffix ∘ dropPrefix) ofile, dropPrefix $ dropSuffix depfile)
  _ → error ("unexpected dependency line ‘‘" ++ l ++ "’’")
```

This implementation of *depFromLine* is based on the assumption that "ghc -M" only generates single-dependency lines and surrounds the colon with spaces; if other output is encountered, this will be flagged as a run-time error.

The two auxiliary functions used here are straightforward:

```
dropSuffix = reverse ∘ tail ∘ dropWhile ('.' ≢) ∘ reverse
dropPrefix = reverse ∘ takeWhile ('/' ≢) ∘ reverse
```

For the generation of dot graphs, we just turn each dependency into an edge and insert some useful default settings:

```
dotOfDeps :: String → [ModDep] → DotGraph
dotOfDeps name = DotGraph name ∘ (settings++) ∘ map (λ(x, y) → Edge x y [])

settings =
  NodeSettings
    [("shape", "plaintext"), ("height", "0"), ("width", "0")
    , ("fontsize", "20")
    ]
  : map Setting [("nodesep", "0.1")
    , ("nslimit", "100"), ("mclimit", "100")]   -- make dot work harder
```

The first three attributes produce outline-free nodes with as little free space around them as possible. The choice of font size, with otherwise standard settings, makes arrows reasonably thin and short (relative to the nodes).

Since the generated dot file can be edited and dot run again, and dot settings can also be supplied on the dot command-line, the lack of possibility to influence the settings chosen here should not be a big problem.