DE GRUYTER OPEN

JACSM 2014, Vol. 6, No. 2, pp. 147 - 157
10.1515/jacsm-2015-0005

MONADIC TREE PRINT

Konrad Grzanek

IT Institute, Academy of Science, Łódź, Poland
kgrzanek@spoleczna.pl, kongra@gmail.com

Abstract

Directed acyclic graphs and trees in particular belong to the most extensively used data structures. Visualizing them properly is a key to a success when developing complex algorithms that make use of them. Textual visualizations a la UNIX tree command is essential when the urge is to deal with large trees. Our aim was to design a library that would exploit this approach and to make an implementation of it for a purely functional programming language. The library uses monads to print directly into an output stream or to generate immutable Strings. This paper gives a detailed overview of the solution.

Key words: Functional programming, monads, Haskell

1 Introduction

Textual presentation of data structures is invariably one of the most effective ways to visualize them. This statement becomes apparent when it comes to presentation of large data structures. The ability to display textual content and working on the presentation results with automated text-processing tools sometimes makes this way of visualizing much more appealing to the end-user than displaying using GUI views. The data structure that is especially susceptible to this approach is tree, or – even more generally – *DAG* (*Directed Acyclic Graph*).

This state of affairs is reflected in such programs as UNIX *tree* (see [7]) command. Figure 1 shows a directory structure displayed in a command line after using this tool:

Figure 1. A directory structure visualized using *tree* command in Ubuntu 12.04.

This paper aims to present our realization[1] of tree-like visualization library for Haskell ([1], [2]), a purely functional ([3]) and statically typed programming language. The library possesses the following properties:

- Generates representations of arbitrary *DAGs*.
- Writes to any *monad* (for a detailed explanation of what monads are, please see [4]), including *IO*. This also means it writes to normal Haskel *Strings* (lists of *Char*) via *Identity* monad.
- Extensively uses Haskell type-system to verify correctness of the usage scenario.

2 Abstraction

A somewhat central point in the design of the library is the *ShowM* type-class[2]. It expresses a relation between the objects (nodes) *s* of the visualized data structure, the monad *m* in which the print process takes place and the options *o* for the process.

```
class (Monad m) ⇒ ShowM m o s where
  showM :: o → s → m ShowS
```

1 To access GitHub repository for the project, please visit [12].

2 We assume the following Haskell extensions in all the listings: *Trustworthy*, *RankNTypes*, *FlexibleContexts*, *MultiParamTypeClasses*.

One important thing is worth mentioning here. All abstractions and implementations described here use a *ShowS* type (as documented in [8]) rather than raw *String* ([*Char*]). The type is declared as follows:

type *ShowS* = *String* → *String*

According to the documentation the *ShowS* functions return a function that prepends the output String to an existing String. This allows constant-time concatenation of results using function composition ([11]). We must remember that the default lists concatenation operator defined as follows (in [10]):

(++) :: [*a*] → [*a*] → [*a*]
(++) [ ] *ys* = *ys*
(++) (*x*:*xs*) *ys* = *x* : *xs* ++ *ys*

doesn't offer a constant time behavior here. For more information, please read the article on difference lists as functions [9], which states: "*Whether this kind of difference list is more efficient than another list representations depends on usage patterns. If an algorithm builds a list by concatenating smaller lists, which are themselves built by concatenating still smaller lists, then use of difference lists can improve performance by effectively "flattening" the list building computations.*". In our case we have exactly the kinds of operations that may be sped up using ShowS.

Another element of the abstraction is the Printer, a function that converts ShowS into a target monadic inter-process representation *a*:

type *Printer m a* = *Monad m* ⇒ *ShowS* → *m a*

The inter-process representations of misc. textual tree elements are combined into larger ones by a Merger, that is effectively a monadic *catamorphism* on lists:

type *Merger m a* = *Monad m* ⇒ [*a*] → *m a*

Finally, there is also an implementation *ADT* (*Algebraic Data Type*) that holds printer and the merger. Impl is de facto the way of expressing what kind of result we expect from the process:

data *Impl m a* = *Impl* !(*Printer m a*) !(*Merger m a*)

3 Abstract Configuration

The configuration of the process must give, in the first place, the answer to the question, what the nature of the visualized tree really is. We introduce *Adjs* (fr. *adjacents* – adjacent element) type that describes ways of generating sub-elements of a tree node:

```
type Adjs m o s = Monad m ⇒ o → s → m [s]
```

There is also a compound configuration ADT:

```
data Conf m a o s =
    Conf
    {
     impl     :: !(Impl m a)
    , adjs    :: !(Adjs m o s)
    , maxDepth :: !(Maybe Int)
    , opts    :: !o
    }
```

that gathers a concrete implementation, the description of a tree (*adjs*), options (opts) as well as a depth constraint – *maxDepth*. The latter may be undefined (*Nothing* constructor of *Maybe* monad).

4 Implementations

There are two implementations in the library: the one that prints to the IO monad and the other, that generates ShowS.

IO implementation (named *io*) prints by evaluating a ShowS and composing the evaluator with *putStr* :: *String* → *IO* (). Its merger does nothing, because the generated pieces of textual representation are being put immediately into the output stream through *putStr* and so there is no accumulation of partial results that would have to be merged.

```
io :: Impl IO ()
io = Impl (putStr ∘ evalShowS) (const $ return ())
    where evalShowS s = s ""
```

String-generating implementation is actually abstract and it is capable to be used in any monad. Its printer only returns a ShowS in a monad *m*, and the merger returns the result of a composition of ShowS:

```
str :: Monad m ⇒ Impl m ShowS
str = Impl return (return ∘ compose)
```

The most obvious concrete use of this abstract implementation is the str implementation, that simply exploits the *Identity* monad to generate ShowS in the pure (side-effect free) way:

```
istr :: Impl Identity ShowS
istr = str
```

5 Realization

This section gives a detailed description of how all the elements described earlier come together to form an effective visualization. Listings to be presented later depend on the following import clauses:

```
import Control .Monad (forM)
import Control .Monad .Identity (Identity)
import Data.Maybe (fromMaybe)
import Kask .Data.Function (compose, rcompose)
import Kask .Data.List (markLast)
```

Functions *compose* and *rcompose* create compositions of functions passed in sequential orders. They are defined as follows in a separate utilities module *Kask.Data.Function* in *kask-base* project[3]:

```
-- | Composes functions passed in a list. Uses foldl.
-- compose [f1, f2, ..., fn] = f1 . f2 . ... . fn
compose :: [a → a] → a → a
compose = foldl (.) id
{-# INLINE compose #-}

-- | Composes functions passed in a list in a reversed order.
-- Uses foldl. compose [f1, f2, ..., fn] = fn . ... . f2 . f1
rcompose :: [a → a] → a → a
rcompose = foldl (flip (.)) id
{-# INLINE rcompose #-}
```

The actual implementation of the monadic printing process was done in a *printImpl* procedure, as given below:

3 Implemented by author.

```
printImpl :: (ShowM m o s) ⇒
          Printer m a
        → Merger m a
        → Adjs m o s
        → o
        → Int
        → s → Int → [Bool] → m a
printImpl printer merger adjs' opts' mdepth s level lastChildInfos = do
```

First, the ShowS form of the current node s is being generated using *showM* procedure:

```
s' ← showM opts' s
```

Then, a target representation is prepared. During this stage all necessary indentation elements are prepended:

```
let repr = if level == 0
             then compose [s', eol]
             else  compose [genIndent lastChildInfos, s', eol]
```

Printer takes the ShowS representation and converts it into an internal, in-process monadic form:

```
r  ← printer repr
```

After that a decision of making the recursive step is being made based on the maximal depth parameter and the result of adjacent nodes generation:

```
rs ← if level == mdepth
     then return []  -- Do not recurse lower than maxDepth'
     else do
       let nextLevel = level + 1
       children ← adjs' opts' s
       forM (zip children (markLast children)) $ \(child, isLast) →
         printImpl printer merger adjs' opts' mdepth
          child nextLevel (isLast:lastChildInfos)
```

Finally, the merger is used to reduce the partial results (adjacent elements' representations) into the target:

```
merger (r:rs)
```

Special state variable *lastChildInfos* is a sequence that allows tracking whether or not a node is a last child of its parent. The information is essential for generating the textual "ruler" lines in the visualization properly. The function markLast used in the algorithm above is defined in *Kask.Data.List* (again in *kask-base*) as follows:

```
-- | Takes a list [e0, e1, ..., en] and returns [False, False, ..., True] or
-- [False, False, ...] if the argument is infinite.
markLast :: [a] → [Bool]
markLast [ ]  = [ ]
markLast (_ : xs)
  | null xs      = [True]
  | otherwise = False : markLast xs
```

Lastly, the procedure named *genIndent* is responsible for generating a proper indentation for a particular node:

```
genIndent :: [Bool] → ShowS
genIndent [ ] = error "Empty genIndent argument !!!"
genIndent (isLast : lastChildInfos) = compose [prefix , suffix ]
  where
    suffix  = if isLast then forLastChild else forChild
    prefix  = rcompose (map indentSymbol (init lastChildInfos))
    indentSymbol True  = emptyIndent
    indentSymbol False = indent
{-# INLINE genIndent #-}
```

It uses the following symbols, all of type ShowS defined as follows:

```
indent      = showString "│   "
emptyIndent = showString "    "
forChild    = showString "├── "
forLastChild = showString "└── "
eol         = showString "\n"
```

6 Interface

This is the most important part of the library from the perspective of its user. Our solution offers its content via the following interface:

```
module Data.Tree.Print
    (
    -- * Abstraction
      ShowM
    , showM
    , Printer
    , Merger
    , Adjs
    , Impl (..)

    -- * Implementations
    , io
    , str
    , istr

    -- * Use
    , Conf (..)
    , printTree
    )
```

As it can be observed, we do not restrict any of its usage scenarios to the one defined and described above. In particular, a potential user may feel free to define his own Impls (implementations), with custom Printers and Mergers.

Printing process starts by calling the following procedure that actually delegates all interesting work to *printImpl*, described above.

```
printTree :: (ShowM m o s) ⇒ Conf m a o s → s → ma
printTree
    Conf { impl     = Impl printer merger
         , adjs     = adjs
         , maxDepth = maxDepth
         , opts     = opts } s =
printImpl printer merger adjs opts (mdepth − 1) s 0 [True ]
   where
    -- When no max depth specified, we use maxBound :: Int
    mdepth = fromMaybe maxBound maxDepth
    mdepth = if mdepth < 1 then 1 else mdepth
{-# INLINE printTree #-}
```

7 Use cases

Testing for performance and for behavior under a heavy stress is crucial when talking about properties of an effective library. We decided to lay our tests on natural numbers generation. Our adjacency function looks as follows:

```
intree :: Monad m ⇒ () → N → m [N]
intree _ i = return $ take m $ iterate (1+) (10 * I)
```

We generate 8 natural numbers on each level, and for any of them consecutive 8 children exist, with 8 children each, etc. down to the level 8 in depth. We make an attempt to visualize this hierarchy of numbers. The tree contains *sum* $ *map* (*8*↑) [*0* .. *7*], that is 2,396,745 nodes. Generating a representation of this structure takes under 1 minute on a commodity laptop machine with AMD E-450 CPU and 4GBs of RAM on board. As can be seen in the following Figure 2, the actual memory usage is abysmal[4]:

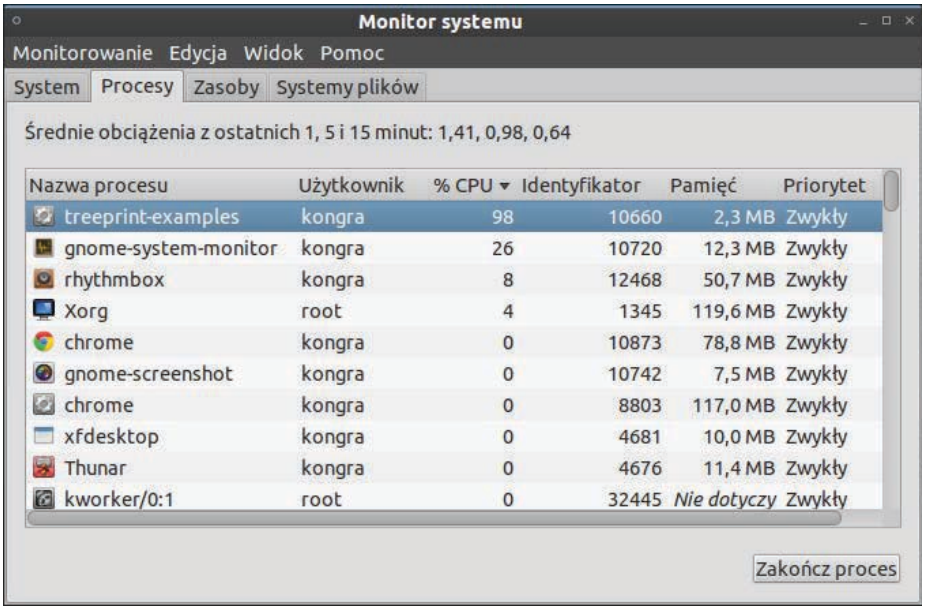

Figure 2. Running textual tree visualization process.

4 Using *io* implementation. In the case of *istr* (*str*) the memory consumption might be significant.

The generated file is over 130 MB in size, but it's the content that matters (Figure 3):

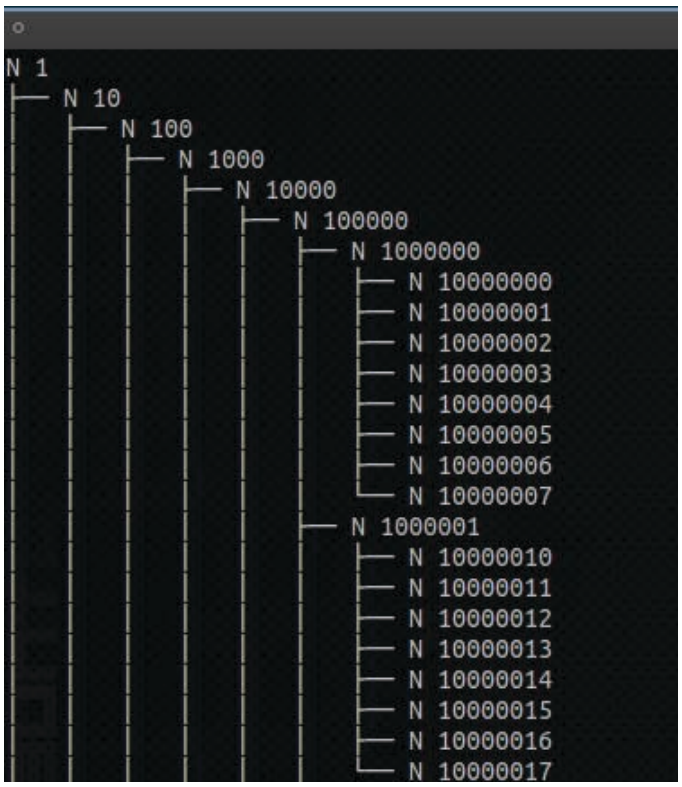

Figure 3. The beginning of generated tree representation.

The monadic tree-print library was developed to help visualize a structure of Rete [5], [6] graph in a Haskell implementation of this great knowledge storing and inferring algorithm. Works on this project are in progress now, and so the presentation of its details are beyond the scope of this paper.

References

1. Peyton Jones S., 1987, *The Implementation of Functional Programming Languages,* Prentice-Hall International Series in Computer Science. Prentice Hall International (UK) Ltd
2. Lipovaca M., 2011, *Learn You a Haskell for Great Good!: A Beginner's Guide,* No Starch Press; 1st edition (April 21, 2011)
3. Bird R., Wadler R., 1988, *Introduction to Functional Programming. Series in Computer Science* (Editor: C.A.R. Hoare), Prentice Hall International (UK) Ltd
4. Awodey S., 2010, *Category Theory, Second Edition,* Oxford University Press

5. Forgy Ch., 1979, *On the efficient implementation of production systems,* Department of Computer Science, Carnegie-Mellon University
6. Doorenbos R. B., 1995, *Production Matching for Large Learning Systems,* PhD Thesis, Computer Science Department, Carnegie Mellon University Pittsburgh, PA
7. *tree (1) - Linux man page*, 2015, http://linux.die.net/man/1/tree
8. Hackage, 2015, ShowS documentation, http://hackage.haskell.org/package/base-4.7.0.2/docs/Prelude.html#t:ShowS
9. Haskell Wiki, 2015, *Difference lists*, https://wiki.haskell.org/Difference_list
10. Hackage, 2015, *(++) operator source code*, http://hackage.haskell.org/package/base-4.7.0.2/docs/src/GHC-Base.html#%2B%2B
11. Stackoverflow, 2013, *What is the showS trick in Haskell?*, http://stackoverflow.com/questions/9197913/what-is-the-shows-trick-in-haskell
12. GitHub, 2015, *tree-print repository*, https://github.com/kongra/treeprint