DE GRUYTER
OPEN

JACSM 2016, Vol. 8, No. 1, pp. 69 - 79

10.1515/jacsm-2016-0005

MONADIC PRINTING REVISITED

Konrad Grzanek

IT Institute, University of Social Sciences
9 Sienkiewicza St., 90-113 Łódź, Poland
kgrzanek@spoleczna.pl

Abstract

Expressive and clear implementation of monadic printing requires an amount of work to define and design proper abstractions to rely upon when performing the actual programming works. Our previous realization of tree printing library left us with a sense of lack with respect to these considerations. This is why we decided to re-design and re-implement the library with core algorithms based upon new, effective and expressive text printing and concatenation routines. This paper presents the results of our work.

Keywords: Functional programming, monads, Haskell, polymorphism

1 Introduction

Textual presentation of data structures is invariably one of the most effective ways to visualize them, especially 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).

Our previous work on this subject aimed towards creating a library for visualizing trees and DAGs. Our few years old paper [5] presented a library for Haskell [1, 2], the purely functional and statically typed programming language. The library described there possessed the following properties:

- The ability to generate representations of arbitrary DAGs.
- Writing to any monad including IO. This also means it was capable of writing to normal Haskel Strings (lists of Char) via Identity monad.
- Extensive use of Haskell type-system to verify correctness of the usage scenarios.

Unfortunately, the design and implementation of this library was not perfect. It missed expressiveness and the clarity of algorithm formulation. These issues led to extensive re-design of the library. The updated architecture of the library consists of:
– Printing abstraction,
– String/Text concatenation routines,
– Re-designed tree printing implemented on top of the two previous ones.

This paper is an attempt to present all the details of the refreshed library.

2 Printing Abstraction and Its Implementations

Generic printing mechanisms are defined in *Kask.Print* module [6]. All its contents are defined in the presence of the following import clauses:

```
import qualified Control.Monad.State.Strict as S
import Data.Monoid ((<>))
import qualified Data.Text as T
import qualified Data.Text.IO as TIO
import qualified Data.Text.Lazy as TL
import qualified Data.Text.Lazy.Builder as TLB
import qualified Data.Text.Lazy.IO as TLIO
import Prelude hiding (print)
```

The most essential abstraction is a *type-class* called *Printable*. It is parameterized by two *type-arguments* out of which the first one, *m*, is a *monad* [4].

We have two procedures defined here, namely *print* and *printLn*. They both return a *unit-type* in the monad *m*. The *printLn* works exactly like *print*, but it adds a newline character to the end of the printed entity of type *p*:

```
class Monad m ⇒ Printable m p where
  print :: p → m ()
  printLn :: p → m ()
```

2.1 IO Monad

The *Printable* type-class is implemented within the *IO monad* for a collection of textual data-types, like *String*, *ShowS*, and *Text*, either lazily and eagerly evaluated. See the listing below:

```
instance Printable IO String where
  print = putStr
  printLn = putStrLn
    {-# INLINE print #-}
    {-# INLINE printLn #-}
instance Printable IO ShowS where
```

```
    print = print ∘ evalShowS
    printLn = putStrLn ∘ evalShowS
      {-# INLINE print #-}
      {-# INLINE printLn #-}
  instance Printable IO T.Text where
    print = TIO.putStr
    printLn = TIO.putStrLn
      {-# INLINE print #-}
      {-# INLINE printLn #-}
  instance Printable IO TL.Text where
    print = TLIO.putStr
    printLn = TLIO.putStrLn
      {-# INLINE print #-}
      {-# INLINE printLn #-}
```

We also provide an *IO-monadic* implementation for an effective textual builder defined in *Data.Text.Lazy.Builder*, like:

```
  instance Printable IO TLB.Builder where
    print = print ∘ TLB.toLazyText
    printLn = printLn ∘ TLB.toLazyText
      {-# INLINE print #-}
      {-# INLINE printLn #-}
```

2.2 Text in the State Monad

Another interesting monad to mention here is the *state monad*, as defined in *Control.Monad.State.Stric*. We define a special type *TextBuilder* to wrap the textual state management within an useful text-coercible abstraction:

```
  type TextBuilder = S.State T.Text
  toText :: TextBuilder () → T.Text
  toText tb = snd (S.runState tb "")
    {-# INLINE toText #-}
```

The *TextBuilder* monad has the following *Printable* implementations for *String* and *ShowS*:

```
  instance Printable TextBuilder String where
    print = print ∘ T.pack
    printLn = printLn ∘ T.pack
      {-# INLINE print #-}
      {-# INLINE printLn #-}
  instance Printable TextBuilder ShowS where
```

```
  print = print ∘ evalShowS
  printLn = printLn ∘ evalShowS
    {-# INLINE print #-}
    {-# INLINE printLn #-}
```

as well as for eagerly, and lazily evaluated *Text*:

```
instance Printable TextBuilder T.Text where
  print txt = do
    buf ← S.get
    S.put (T.append buf txt)
   {-# INLINE print #-}
  printLn txt = do
    buf ← S.get
    S.put (T.append (T.append buf txt) "\n")
   {-# INLINE printLn #-}
instance Printable TextBuilder TL.Text where
  print = print ∘ TL.toStrict
  printLn = printLn ∘ TL.toStrict
    {-# INLINE print #-}
    {-# INLINE printLn #-}
```

We also provide implementation for *Data.Text.Lazy.Builder* like in the case of *IO* monad:

```
instance Printable TextBuilder TLB.Builder where
  print = print ∘ TLB.toLazyText
  printLn = printLn ∘ TLB.toLazyText
    {-# INLINE print #-}
    {-# INLINE printLn #-}
```

2.3 Lazy Text Builder in the State Monad

Eagerly evaluated state monad may be also used as a basis for a lazily evaluated string builder, as defined below, together with two state evaluators:

```
type LazyTextBuilder = S.State TLB.Builder
toLazyTextBuilder :: LazyTextBuilder () → TLB.Builder
toLazyTextBuilder tb = snd $ S.runState tb $ TLB.fromString ""
  {-# INLINE toLazyTextBuilder #-}
toLazyText :: LazyTextBuilder () → TL.Text
toLazyText = TLB.toLazyText ∘ toLazyTextBuilder
  {-# INLINE toLazyText #-}
```

Like in the case of the previous monadic implementations, firstly we define the implementations for *String* and *ShowS*:

```
instance Printable LazyTextBuilder String where
  print = print ∘ T.pack
  printLn = printLn ∘ T.pack
    {-# INLINE print #-}
    {-# INLINE printLn #-}
instance Printable LazyTextBuilder ShowS where
  print = print ∘ evalShowS
  printLn = printLn ∘ evalShowS
    {-# INLINE print #-}
    {-# INLINE printLn #-}
```

as well as for strictly and lazily evaluated *Text* instances:

```
instance Printable LazyTextBuilder T.Text where
  print = print ∘ TLB.fromText
  printLn = printLn ∘ TLB.fromText
    {-# INLINE print #-}
    {-# INLINE printLn #-}
instance Printable LazyTextBuilder TL.Text where
  print = print ∘ TLB.fromLazyText
  printLn = printLn ∘ TLB.fromLazyText
    {-# INLINE print #-}
    {-# INLINE printLn #-}
```

To make this realization conceptually coherent with the previous ones, we also provide an implementation for *TLB.Builder* (as it was presented in the previous sub-sections):

```
instance Printable LazyTextBuilder TLB.Builder where
  print b = do
    builder ← S.get
    S.put (builder <> b)
   {-# INLINE print #-}
  printLn b = do
    builder ← S.get
    S.put (builder <> b <> TLB.fromLazyText "\n")
   {-# INLINE printLn #-}
```

2.4 ShowS in the State Monad

For *ShowS* type we define a separate State Monad instance, together with the following evaluators:

```
type StringBuilder = S.State ShowS
evalShowS :: ShowS → String
```

```
evalShowS s = s ""
  {-# INLINE evalShowS #-}
toShowS :: StringBuilder () → ShowS
toShowS tb = snd (S.runState tb (showString ""))
  {-# INLINE toShowS #-}
toString :: StringBuilder () → String
toString = evalShowS ∘ toShowS
  {-# INLINE toString #-}
```

The *String* and *ShowS* instances of the *Printable* type-class raise up in a natural way:

```
instance Printable StringBuilder String where
   print = print ∘ showString
   printLn = printLn ∘ showString
     {-# INLINE print #-}
     {-# INLINE printLn #-}
instance Printable StringBuilder ShowS where
   print s = do
      buf ← S.get
      S.put (buf ∘ s)
     {-# INLINE print #-}
   printLn s = do
      buf ← S.get
      S.put (buf ∘ s ∘ showString "\n")
     {-# INLINE printLn #-}
```

along with *Text* instances, like in the following listing:

```
instance Printable StringBuilder T.Text where
   print = print ∘ T.unpack
   printLn = printLn ∘ T.unpack
     {-# INLINE print #-}
     {-# INLINE printLn #-}
instance Printable StringBuilder TL.Text where
   print = print ∘ TL.toStrict
   printLn = printLn ∘ TL.toStrict
     {-# INLINE print #-}
     {-# INLINE printLn #-}
instance Printable StringBuilder TLB.Builder where
   print = print ∘ TLB.toLazyText
   printLn = printLn ∘ TLB.toLazyText
     {-# INLINE print #-}
     {-# INLINE printLn #-}
```

3 Compatible Abstraction for Concatenation

Early in the design phase it became apparent that we might use the *Printable* for string concatenation. After all the concatenation may be viewed here as printing into the concatenating (string/text builder) object. To make things clear we provide the following *StrCat* type-class, that is another useful abstraction in our library:

```
class StrCat c where
  strCat :: (Foldable t) ⇒ t c → c
```

Concatenation is being treated as a *fold* (e.g. see [3]) operation, that's why we define the *strCat* mechanism as taking place inside a *Foldable*.

Functional merging of *StrCat* and *Printable* takes place via the following *strCatWith* procedure:

```
strCatWith :: (Printable m c, Foldable t) ⇒ (m () → c) → t c → c
strCatWith f = f ∘ mapM_ print
 {-# INLINE strCatWith #-}
```

This immediately allows us to provide *StrCat* implementations for *String* and *ShowS*:

```
instance StrCat String where
  strCat = strCatWith toString
   {-# INLINE strCat #-}
instance StrCat ShowS where
  strCat = strCatWith toShowS
   {-# INLINE strCat #-}
```

The same approach applies to *Text* and *TLB.Builder*:

```
instance StrCat T.Text where
  strCat = strCatWith toText
   {-# INLINE strCat #-}
instance StrCat TL.Text where
  strCat = strCatWith toLazyText
   {-# INLINE strCat #-}
instance StrCat TLB.Builder where
  strCat = strCatWith toLazyTextBuilder
   {-# INLINE strCat #-}
```

4 Re-designed Tree Printing

All abstractions and their implementations described so far allow us to provide an updated realization of tree printing, previously defined and presented in [5]. The new realization can be viewed as a whole in *Kask.Data.Tree.Print* module [7]. In the presence of the following import clauses:

```
import Control.Monad (unless, forM_)
import Data.Foldable (toList)
import qualified Data.Text as T
import qualified Data.Text.Lazy as TL
import qualified Data.Text.Lazy.Builder as TLB
import qualified Kask.Constr as C
import Kask.Data.List (markLast)
import qualified Kask.Print as P
import Prelude hiding (Show, show)
```

we have the basic type definitions like below:

```
type Adjs a t = Foldable t ⇒ a → t a
type Show a s = Symbolic s ⇒ a → s
type Depth = C.Constr (C.BoundsConstr C.Positive) Int
```

One additional visible change with respect to mechanisms defined in [5] relates to *Depth* - a new data type that describes the maximum depth of tree-printing. Currently it is a positive integer, with the contract enforced by using *Constr* and *BoundsConstr*, an effective compile-time contract definition routines, also provided by the *kask* repository.

4.1 Tree Printing API

Essentially it consists of a single procedure *printTree* with the following signature and implementation:

```
printTree :: (P.Printable m s, Symbolic s, Foldable t) ⇒
  a → Adjs a t → Show a s → Maybe Depth → m ()
printTree node adjacent show maxDepth =
  doPrintTree node adjacent show (case maxDepth of
    Just d → C.unconstr d − 1
    Nothing → maxBound)
    0        -- initial level is 0-th
    [True]   -- node has no siblings
    True     -- .. and it is the first one
```

4.2 Simplified and More Expressive Tree Printing Implementation

The procedure takes the following form:

```
doPrintTree :: (P.Printable m s, Symbolic s, Foldable t) ⇒
    a → Adjs a t → Show a s → Int → Int → [Bool] → Bool → m ()
```

```
doPrintTree node adjacent show maxDepth level
  lastChildMarks isFirst = do
  let s      = show node
        pfx = if isFirst then empty else eol
        repr = if level ≡ 0
          then P.strCat [pfx, s]
          else P.strCat [pfx, genIndent lastChildMarks, s]
  P.print repr
  unless (level ≡ maxDepth) $ do
    let children = toList $ adjacent node
    forM_ (zip children (markLast children)) $ λ(child, isLast) →
      doPrintTree child adjacent show maxDepth (level + 1)
        (isLast : lastChildMarks) False
```

All the printing, concatenation and string-building abstractions allowed us to achieve two goals:

1. Make the implementation clear and obvious.
2. Make the API expressive.

The clarification seems apparent here, and the expressiveness enhancement takes place thanks to powerful compile time abstractions provided in the signature: *Printable*, *Foldable*, *Adjs*, *Show*.

4.3 Further Implementation Details

String concatenation abstraction is also used to implement properly the indentation used to layout the printed tree:

```
genIndent :: Symbolic s ⇒ [Bool] → s
genIndent [] = empty   -- should not happen anyway
genIndent (isLast : lastChildMarks) = P.strCat [prefix, suffix]
  where
    indentSymbol True = emptyIndent
    indentSymbol False = indent
    suffix  = if isLast then forLastChild else forChild
    prefix = P.strCat $ fmap indentSymbol $ reverse $ init lastChildMarks
```

Additionally we use a *Symbolic* type class that holds the information about all textual elements forming the tree printing layout. The abstraction is defined as:

```
class P.StrCat s ⇒ Symbolic s where
  indent       :: s
  emptyIndent :: s
  forChild     :: s
  forLastChild :: s
  eol          :: s
  empty        :: s
```

with the following realization for *String* and *ShowS*:

```
instance Symbolic String where
  indent      = "   "
  emptyIndent = "    "
  forChild    = " "
  forLastChild = " "
  eol         = "\n"
  empty       = ""
instance Symbolic ShowS where
  indent      = showString (indent      :: String)
  emptyIndent = showString (emptyIndent :: String)
  forChild    = showString (forChild    :: String)
  forLastChild = showString (forLastChild :: String)
  eol         = showString (eol         :: String)
  empty       = showString (empty       :: String)
```

Finally, we also provide an implementation for *Text*:

```
instance Symbolic T.Text where
  indent      = T.pack (indent      :: String)
  emptyIndent = T.pack (emptyIndent :: String)
  forChild    = T.pack (forChild    :: String)
  forLastChild = T.pack (forLastChild :: String)
  eol         = T.pack (eol         :: String)
  empty       = T.pack (empty       :: String)
instance Symbolic TL.Text where
  indent      = TL.pack (indent      :: String)
  emptyIndent = TL.pack (emptyIndent :: String)
  forChild    = TL.pack (forChild    :: String)
  forLastChild = TL.pack (forLastChild :: String)
  eol         = TL.pack (eol         :: String)
  empty       = TL.pack (empty       :: String)
```

and for *TLB.Builder*:

```
instance Symbolic TLB.Builder where
  indent      = TLB.fromText (indent      :: T.Text)
  emptyIndent = TLB.fromText (emptyIndent :: T.Text)
  forChild    = TLB.fromText (forChild    :: T.Text)
  forLastChild = TLB.fromText (forLastChild :: T.Text)
  eol         = TLB.fromText (eol         :: T.Text)
  empty       = TLB.fromText (empty       :: T.Text)
```

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 Beginners 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. Grzanek K., 2014, Monadic Tree Print, JACSM 2014, Vol. 6, No. 2, pp. 147-157
6. GitHub, 2016, Kask.Print module: Kask repository, https://github.com/kongra/kask/blob/master/src/Kask/Print.hs
7. GitHub, 2016, Kask.Data.Tree.Print module: https://github.com/kongra/kask/blob/master/src/Kask/Data/Tree/Print.hs