Beyond Pretty-Printing: Galley Concepts in Document Formatting Combinators

Wolfram Kahl

Institut für Softwaretechnologie, Fakultät für Informatik
Universität der Bundeswehr München, D-85577 Neubiberg, Germany
E-Mail: kahl@informatik.unibw-muenchen.de

Abstract. Galleys have been introduced by Jeff Kingston as one of the key concepts underlying his advanced document formatting system Lout. Although Lout is built on a lazy functional programming language, galley concepts are implemented as part of that language and defined only informally.

In this paper we present a first formalisation of document formatting combinators using galley concepts in the purely functional programming language Haskell.

1 Introduction and Related Work

Pretty printing is an established topic in the functional programming community; it is targeted towards appropriate layout of data structures, with one of the main applications being user-friendly display of internal data structures of compilers, for example for the creation of error messages. Hughes has turned the development of pretty-printing combinators into a fine art in [2,3]. His combinators have subsequently been improved upon by Peyton Jones [10]. Wadler [11] has presented a simpler design; yet another set of "optimal pretty printers" has come forth in [1].

However, document formatting with problems such as the placement of footnotes is simply outside the scope of pretty printing as discussed in the papers mentioned so far.

The best-known document formatting system in the scientific community is probably TEX [9]. TEX is built on a rather large number of primitives that incorporate many fine points of the art of typesetting. TEX offers access to these primitives through an imperative programming language with dynamic binding and rather weak structuring capabilities — in short, TEX considered as a programming language is about as impure as possible.

Another document formatting system recently gaining acceptance and popularity is Jeff Kingston's Lout [6,7,8]. Lout had twenty-three primitives in the version presented in [6] — this number has grown in the meantime, but is still small. The programming language available for programming document format definitions is a small lazy functional programming language with an innovative definition and name visibility mechanism, flexible composition of aligned objects, a versatile cross reference concept and, most important for this paper, the *galley* concept.

The galley abstraction allows to direct part of the document to places somewhere else in the document, for example footnotes are implemented as galleys directed to targets at the bottom of the page. The generality of the galley abstraction as conceived by

G. Gupta (Ed.): PADL'99, LNCS 1551, pp. 76-90, 1998.

Kingston allows essentially all problems of text flow, tables of contents, and even indices and lists of references to be defined via galleys, so that Lout is a document formatting system that does not even need a built-in concept of pages — pages are defined in the programming language as objects of a certain size and containing certain targets.

However, the galley concept itself is defined outside the functional programming language of Lout, described only informally in [6] (and even more informally in the "Expert's Guide" [7]), and implemented in C.

In this paper we give a completely formal definition of galley movement in the pure functional programming language Haskell. By disregarding all aspects unrelated to galley flushing we are able to present a small set of basic document combinators that satisfactorily capture the essential behaviour of galleys in Lout and still can be implemented with relative ease.

2 Overview of Document Formatting Combinators

We start with giving a brief overview of how to reflect the central features of Lout's galleys in document formatting combinators.

The exposition of this paper is centred around *objects*, the datatype for which we shall present in detail in the next section:

```
> data Object
```

A very important object is the `null` object which never leaves a trace[1]:

```
> null :: Object
```

For being able to build up more interesting documents, we have to decide on their basic constituents. Adhering to Lout nomenclature we call these *components*. For this paper, we consider lists of lines as components — this is more general than just lines and allows us to experiment with unbreakable components of different heights:

```
> type Component = [String]
```

We want to be able to turn components into objects, and to prefix objects with components:

```
> singleton :: Component -> Object
> prefix    :: Component -> Object -> Object
```

Obviously we can define `singleton` via `prefix`:

```
> singleton c = prefix c null
```

A more general means to compose objects is *concatenation:*

```
> (#) :: Object -> Object -> Object
```

In the expert's guide to Lout [7], Kingston defines: "*A galley is an object plus a cross reference which points to where the object is to appear.*" However, cross references occurring in galleys are limited to those referring to a name together with a direction[2]:

```
> type Galley = (String, Direction, Object)
> data Direction = Preceding | Following
```

Such a cross reference is taken to mean the list of all targets of the same name, starting with the next occurence in the respective direction and always continuing forward.

---

[1] Since we have a name clash here, we have to remember to "`> import Prelude hiding (null)`"!

[2] In Lout, a third mode is possible which is only used for bibliographic reference list entries.

A galley considered as an object sends itself as the first outgoing galley and otherwise behaves like `null` — in Lout, such a galley is built by defining a symbol with the galley object as the body of the definition and a special "@Target" parameter (or the "into" abbreviations).

```
> galley :: Galley -> Object
```

In Lout, a target that is receptive to galleys is built using the "@Galley" symbol, which therefore does not represent a galley, but a possible *target* for galleys. Our combinator for introducing targets is the following:

```
> target :: String -> Object
```

These combinators are not yet sufficient for the definition of pages. Obviously a page should contain targets for text and footnotes, but the essential feature of a page is that it restricts the size of what can be received into these targets. Size restriction is achieved by using the following combinator:

```
> high :: Int -> Object -> Object
```

The footnote section, when it appears, will start with some space and a horizontal line followed by a list of footnotes — we can achieve this by "`prefix ["", "-----"] footList`".

However, when there is no footnote, then the whole footnote section should disappear — Lout has the concept of *receptive symbols* for this: A receptive symbol is any symbol defined to directly or indirectly contain a target, and receptive symbols are replaced by `null` if no contained target ever receives a galley.

Our framework lacks the concepts of symbols and definitions, therefore we make the effect of receptive symbols explicit as a combinator that lets its argument object appear only if it receives a galley:

```
> delay :: Object -> Object
```

Lout relies on recursive definitions for implementing e.g. lists of pages or footnotes; however Lout uses a special semantics of recursion that includes recursive objects being delayed and not expanding unless space is sufficient. Since we cannot change the semantics of recursion in Haskell, we have to provide our own combinator for building recursive objects and abstain from using Haskell recursion in object definitions:

```
> recurse :: (Object -> Object) -> Object
```

Finally we need a function that actually formats a document; its result is a list of components, which usually shall stand for pages:

```
> force :: Object -> [Component]
```

3 A Simple Document Format

These combinators are sufficient to build document formats analogous to those used to explain galley concepts in [6,7].

A list of pages to hold the whole document is a simple recursion over of a page concatenated with the recursive call, and every page is a restricted-height object containing a target for text concatenated with a footnote section:

```
> pageList = recurse (page #)
> page = high 12 (target "TextPlace" # footSect)
```

The footnote section is delayed, and contains a recursive list of footnote targets (in Haskell, the infix operator "$" is low-priority, right-associative function application):

```
> footSect = delay $ prefix ["", "-----"] footList
> footList = recurse (target "FootPlace" #)
```

Now we have two kinds of targets; in the simple documents we are considering, correspondingly two kinds of galleys occur, the main text of the document and footnotes:

```
> text     t = galley ("TextPlace",Preceding,t)
> footNote f = galley ("FootPlace",Following,f)
```

A sample document is constructed with ease:

```
> purcell = prefix ["PURCELL(1)"] (footNote fn) # body
> fn = singleton ["(1) Blom, Eric. Some", "Great Composers.", "Oxford, 1944."]
> body = foldr prefix null [ ... ]
```

So now we can format the running example of [6,7] by evaluating the expression: `force (pageList # text purcell)`, and displaying the result in a pleasing manner:

```
-------------------------------------------------------------------------------
|PURCELL(1)               |that is regarded          |of the world's musical    |
|                         |elsewhere as perfectly    |classics, as here, we find|
|In the world of music    |normal and natural; but   |that we cannot omit this  |
|England is supposed to be|if foreign students of    |English master.           |
|a mere province.  If she |musical history have to   |                          |
|produces an indifferent  |acknowledge a British     |                          |
|composer or performer,   |musical genius, he is     |                          |
|                         |considered a freak.       |                          |
|-----                    |                          |                          |
|(1) Blom, Eric. Some     |    Such a freak is       |                          |
|Great Composers.         |Henry Purcell.  Yet if we |                          |
|Oxford, 1944.            |make a choice of fifteen  |                          |
-------------------------------------------------------------------------------
```

4 Implementation

Objects are modeled as entities that essentially are prepared to react to two kinds of stimuli:

– an *evaluation request* tries to extract a component into available space,
– a *reception request* asks the object to receive a galley directed at some target contained in the object — we consider it to be an error to send a galley to an object that does not contain a target for it.

The reactions to both of these have to take into account the available space, which is expressed as a `Constraint`, and a boolean *forcing* parameter which influences the eagerness of the object's behaviour and depends on the object's position inside concatenation chains.

Furthermore we need to know which receiving targets an object contains, both open and within delayed or recursive subobjects, and its minimum height as a whole. Therefore we define:

```
> data Object = Object
>  {eval           :: Bool -> Constraint -> EvalResult
>  ,receive        :: Bool -> Constraint -> Galley -> RcvResult
>  ,openTargets    :: [String]
>  ,delayedTargets :: [String]
>  ,height      :: Int
>  }
```

```
> type RcvResult = (Bool, [Galley], Object)
> targets o = openTargets o ++ delayedTargets o
```

The result of evaluating can be:

– `Disappearing`: The object disappears without trace, which is what `null` always does.
– `Suspended`: The object cannot yet say what it is going to be, as for example targets that have not yet been attached to, or recursive or delayed objects. If `forcing` is set to `True` in the call to `eval`, an object must not suspend — it will then usually choose to disappear.
– `NoSpace`: The object wants to yield a first component which is too large for the current constraints.
– `Sending`: A list of galleys is dispatched.
– `Yielding`: A non-empty component fitting into the current space constraints is yielded.

Except `Disappearing`, all results carry a continuation object since the evaluation attempt always might change the object:

```
> data EvalResult = Disappearing
>                 | Suspended {                  obj :: Object}
>                 | NoSpace   {                  obj :: Object}
>                 | Sending   {gall :: [Galley], obj :: Object}
>                 | Yielding  {comp :: Component, obj :: Object}
```

The only measure to be constrained in our current setting is the height of objects; and there is also the possibility that an object is evaluated without any constraints, so we define:

```
> type Constraint = Maybe Int
```

For modelling more of Lout, the constraint datatype should not only contain information about the allowed height and width, but also other settings such as font and spacing information.

With this we can already define the global document formatting function which forces evaluation and does not impose any constraints — we decide to simple discard any stray galleys:

```
> force :: Object -> [Component]
> force o = case eval o True Nothing of
>   Disappearing  -> []
>   Yielding c o' -> c : force o'
>   Sending gs o' -> force o'
```

The possible results of `receive` have the following semantics:

– "`(True,gs,o)`" means that the galley was received, triggered further galleys `gs`, and that the resulting object is "`o`".
– "`(False,gs,o)`" means that the galley was *not* received, usually because of insufficient space, but that the object has triggered the galleys `gs` and may have undergone changes (such as deciding never to receive anymore); therefore one now has to refer to "`o`" instead. Although the argument galley has not been received, it may have been modified or discarded, so if it is to be sent on, it will always be found in `gs`.

The `null` object is always ready to disappear immediately without leaving any trace (`rcverror` documents an error that should not occur since objects should only be sent galleys they can receive):

```
> null :: Object
> null = Object
>  {eval = (\ forcing co -> Disappearing)
>  ,receive = (\ forcing co g -> rcverror g "null")
>  ,openTargets = []
>  ,delayedTargets = []
>  ,height = 0
>  }
```

A singleton object yields its component if there is space for it and otherwise signals `NoSpace`. For testing whether some component or object fits into a constraint, respectively for decreasing a constraint by the amount indicated by some component or object, we use an appropriately defined class `Constrainer` and the following functions:

```
> (&?) :: Constrainer c => Constraint -> c -> Bool
> (&-) :: Constrainer c => Constraint -> c -> Constraint
```

We give an explicit definition of `singleton`:

```
> singleton :: Component -> Object
> singleton c = o where
>  o = Object
>   {eval = (\ forcing co -> if co &? c then Yielding c null else NoSpace o)
>   ,receive = (\ forcing co g -> rcverror g "singleton")
>   ,openTargets = [], delayedTargets = []
>   ,height = length c
>   }
```

The same space considerations apply to `prefix`, which otherwise inherits the behaviour of the prefixed object:

```
> prefix :: Component -> Object -> Object
> prefix c o = o' where
>  o' = Object
>   {eval =  (\ forcing co -> if co &? c then Yielding c o else NoSpace o')
>   ,receive = (\ forcing co g ->
>                 thrdUpd3 (prefix c) $ receive o forcing (co &- c) g)
>   ,openTargets = openTargets o
>   ,delayedTargets = delayedTargets o
>   ,height = length c + height o
>   }
```

For updating the object within the `RcvResult` we used the prelude-quality `thrdUpd3`:

```
> thrdUpd3 :: (c -> d) -> (a, b, c) -> (a, b, d)
> thrdUpd3 f (a, b, c) = (a, b, f c)
```

It is easy to see that the equation `singleton c = prefix c null` does indeed hold.

A galley considered as an object sends itself as the first outgoing galley and otherwise behaves like `null`:

```
> galley :: Galley -> Object
> galley g = Object
>   {eval    = (\ forcing co -> Sending [g] null)
>   ,receive = (\ forcing co g' -> rcverror g' "galley")
>   ,openTargets = [],  delayedTargets = []
>   ,height = 0
>   }
```

Before we start to present the definition of object concatenation, we first introduce an auxiliary combinator that shall be needed there. Taking a component `c` and an object `o`, `suffix` delivers an object that yields `c` only after `o` disappears:

```
> suffix :: Component -> Object -> Object
> suffix c o = o' where
>  o' = Object
>   {eval = (\ forcing co -> case eval o forcing (co &- c) of
>                   Disappearing -> eval (singleton c) forcing co
>                   r -> r {obj = suffix c (obj r)})
>   ,receive = (\ forcing co g ->
>                   thrdUpd3 (suffix c) $ receive o forcing (co &- c) g)
>   ,openTargets = openTargets o
>   ,delayedTargets = delayedTargets o
>   ,height = length c + height o
>   }
```

Concatenation of two objects into a single object has two main tasks:

– *Communication*: Galleys sent off by one object, according to their direction either have to be sent to the other object or to the context.
– *Space negotiation*: Two objects placed together within one constraint may only grow as long as they do not infringe into each others space.

Since the final result of evaluating an object must always either be that the object disappears or that it yields a component *from the object's beginning*, a natural asymmetry is imposed on both seemingly symmetrical tasks. We resolve this asymmetry using the `forcing` parameters to `eval` and `receive` as indicated by the following:

– While galleys are sent between the several objects in a concatenation tree, galleys arriving at the left-most leaf are not expanded, while all other galleys are expanded as far as possible (are "forced") on arrival. This ensures for example that footnotes are allowed to grow after being sent off before the subsequent main text (which is sent to the competing `"TextPlace"` target) is allowed to grow into its space.
– While objects are evaluated, `Suspended` objects are left waiting everywhere except in the right-most leaf. There, evaluation is "forced", and targets or delayed objects are made to disappear — if no galley has reached them yet, there will also be no such galley later on. Otherwise we would never be able to determine the end of a document in our sample format, since there is always a `pageList` waiting at the right end.

For communication of galleys we first present a few auxiliary definitions. We use a communication state that contains two objects and a flag signalling whether some galley already has been received:

```
> type OCState = (Bool, Object, Object)
```

This is then used as the state for a simple state monad:

```
> type STfun s x = s -> (s,x)
> type STOc = STfun OCState [Galley]
```

We provide a function to combine results of `STOc` transformers into `RcvResult`s:

```
> ocMkResult :: (OCState,[Galley]) -> RcvResult
> ocMkResult ((rcv, o1, o2), gs) = (rcv, gs, o1 # o2)
```

Communication between two adjacent objects has to respect the direction of the galleys sent off by either. Furthermore we take care that only the very first object of a longer concatenation receives in a delayed manner, while all later objects receive with `forcing` set. The definitions of `ocGalleyFrom1` and `ocGalleyFrom2` are therefore symmetric except for the `forcing` parameter passed to the `receive` calls: The second object of a concatenation always receives with `forcing = True`, while the first inherits from above:

```
> ocGalleyFrom1, ocGalleyFrom2 :: Bool -> Constraint -> Galley -> STOc
> ocGalleyFrom1 forcing co g @ (name,Preceding,_) s = (s, [g])
> ocGalleyFrom1 forcing co g @ (name,Following,_) s @ (rcv, o1, o2) =
>   if name `notElem` targets o2 then (s,[g])
>   else let (rcv', gs2, o2') = receive o2 True (co &- o1) g
>        in ocGalleysFrom2 forcing co gs2 (rcv || rcv', o1, o2')
> ocGalleyFrom2 forcing co g @ (name,Following,_) s = (s, [g])
> ocGalleyFrom2 forcing co g @ (name,Preceding,_) s @ (rcv, o1, o2) =
>  if name `notElem` targets o1 then (s,[g])
>  else let (rcv', gs1, o1') = receive o1 forcing (co &- o2) g
>       in ocGalleysFrom1 forcing co gs1 (rcv || rcv', o1', o2)
```

Iterated communication is achieved via a simple fold function in the state monad:

```
> stfold :: (a -> STfun b [c]) -> [a] -> STfun b [c]
> stfold f []     s = (s,[])
> stfold f (g:gs) s = let (s' , gs' ) = f g s
>                         (s'', gs'') = stfold f gs s'
>                     in (s'', gs' ++ gs'')
> ocGalleysFrom1, ocGalleysFrom2 :: Bool -> Constraint -> [Galley] -> STOc
> ocGalleysFrom1 forcing co = stfold (ocGalleyFrom1 forcing co)
> ocGalleysFrom2 forcing co = stfold (ocGalleyFrom2 forcing co)
```

In the definition of `receive`, the concatenation looks at the open and delayed receptors of both components and decides accordingly to which the galley is going to be sent.

```
> (#) :: Object -> Object -> Object
> o1 # o2 = o where
>  o = Object
>   {eval = ocEval o1 o2
>   ,receive = (\ forcing co g @ (name,d,o') -> let
>     send1 = let (r,gs,o1') = receive o1 forcing (co &- o2) g
>             in ocMkResult $ ocGalleysFrom1 forcing co gs (r, o1', o2)
>     send2 = let (r,gs,o2') = receive o2 True (co &- o1) g
>             in ocMkResult $ ocGalleysFrom2 forcing co gs (r, o1, o2')
>     sendO1 x = if name `elem` openTargets o1 then send1 else x
>     sendO2 x = if name `elem` openTargets o2 then send2 else x
>     sendD1 x = if name `elem` delayedTargets o1 then send1 else x
>     sendD2 x = if name `elem` delayedTargets o2 then send2 else x
>     in case d of
>       Following -> sendO1 $ sendO2 $ sendD1 $ sendD2 $ rcverror g "(#)"
>       Preceding -> sendO2 $ sendO1 $ sendD2 $ sendD1 $ rcverror g "(#)")
>   ,openTargets = openTargets o1 ++ openTargets o2
>   ,delayedTargets = delayedTargets o1 ++ delayedTargets o2
>   ,height = height o1 + height o2
>   }
```

When evaluating the concatenation of two objects, we first evaluate the first (never forcing) and then perform any communication that might result from the galleys sent by that evaluation. Only the case that the first object suspends is not straight-forward. In that case, we always evaluate the second object, too, in the hope that the resulting communication relieves the suspension. Therefore we resume evaluation of the combined object if the concatenated object is forced or if either communication succeeded or if the evaluation of the second object yielded a component — in order to enable more of the second object to be evaluated if necessary we have to use `suffix` to stick that component onto the end of the first object in the recombination.

```
> ocEval :: Object -> Object -> Bool -> Constraint -> EvalResult
> ocEval o1 o2 forcing co = case eval o1 False (co &- o2) of
>   Disappearing -> eval o2 forcing co
>   NoSpace o1′ -> NoSpace (o1′ # o2)
>   Yielding c o1′ -> Yielding c (o1′ # o2)
>   Sending gs o1′ ->
>     case ocMkResult $ ocGalleysFrom1 False co gs (False, o1′, o2) of
>      (rcv,[],o′) -> eval o′ forcing co
>      (rcv,gs,o′) -> Sending gs o′
>   Suspended o1′ -> case eval o2 forcing (co &- o1′) of
>    Disappearing -> if forcing then eval o1′ forcing co else Suspended o1′
>    Suspended o2′-> Suspended (o1′ # o2′)
>    NoSpace o2′ -> if forcing then NoSpace o2′ else Suspended (o1′ # o2′)
>    Yielding c o2′ -> eval (suffix c o1′ # o2′) forcing co
>    Sending gs o2′ ->
>     case ocMkResult $ ocGalleysFrom2 False co gs (False, o1′, o2′) of
>       (True, [], o′) -> eval o′ forcing co
>       (False, [], o′) -> error ("empty Sending???")
>       (_, gs, o′) -> Sending gs o′
```

The function `high` constrains the height of its object argument to the amount indicated by the integer argument. If the object argument does not fit in the indicated height, the remainder object is discarded and a placeholder of the appropriate height is substituted. Otherwise, the result is filled to the specified height.

Filling up to the specified height involves recursively evaluating the remaining objects after `Yielding` and concatenating the components into a single component — this task is delegated to the auxiliary function `prefixConc`. Besides we also use the following small functions:

```
> strut h = replicate h ""
> fill h c = take h (c ++ repeat "")
```

If the resulting object itself is placed in a too tightly constrained environment, then it does not fit and remains as the continuation object of `NoSpace`.

```
> high :: Int -> Object -> Object
> high h o = o′ where
>  eval′ forcing = case eval o forcing (Just h) of
>    NoSpace o1 -> Yielding (fill h ["@High: Object too large"]) null
>    Disappearing -> Yielding (strut h) null
>    Suspended o1 -> Suspended (high h o1)
>    Sending gs o1 -> Sending gs (high h o1)
>    Yielding c o1 -> let h′ = h - length c in
>      if h′ < 0 then error "@High: yielded component too high!"
>      else case eval (high h′ o1) forcing Nothing of
>        Yielding c′ o2 -> Yielding (c ++ c′) o2
>        Sending gs o2 -> Sending gs (prefixConc c o2)
>        Suspended o2 -> Suspended (prefixConc c o2)
>        NoSpace o2 -> error "@High: NoSpace in recursive call!"
>        Disappearing -> Yielding (fill h c) null
>  o′= Object
>   {eval = (\ forcing co -> case co of
>               Nothing -> eval′ forcing
>               Just h′ -> if h′ < h then NoSpace o′ else eval′ forcing)
>   ,receive = (\ forcing co g ->
>                thrdUpd3 (high h) $ receive o forcing (Just h) g)
```

```
>   ,openTargets = openTargets o
>   ,delayedTargets = delayedTargets o
>   ,height = h
>   }
```

The auxiliary function `prefixConc` used above is the tool that allows `high` to assemble components of exactly the right height from objects that yield small irregular components by modifying its argument object to *concatenate* the argument component before its first yielded component:

```
> prefixConc :: Component -> Object -> Object
> prefixConc c o = o' where
>  o' = Object
>   {eval = (\ forcing co -> case eval o forcing (co &- c) of
>     Disappearing -> Yielding c null
>     Yielding c' o2 -> Yielding (c ++ c') o2
>     r -> r {obj = prefixConc c (obj r)})
>   ,receive = (\ forcing co g -> if co &? c
>                 then thrdUpd3 (prefixConc c) $ receive o forcing (co &- c) g
>                 else (False, [forward g], o'))
>   ,openTargets = openTargets o, delayedTargets = delayedTargets o
>   ,height = length c + height o
>   }
```

If a galley cannot be received because of lack of space, it has to be sent on looking for its next target; but it always has to be turned forward for this purpose (imagine overflowing `text` in the example presented in Sect. 3; although it finds its first page initially as the `Preceding` target, the next target to be unveiled by `pagelist` is the `Following`):

```
> forward :: Galley -> Galley
> forward (name,d,o) = (name,Following,o)
```

A galley that has reached its target transforms this target into a concatenation consisting of the object carried by the galley and the target itself which has to be ready to receive more galleys directed at it. However, if the galley object does not fit into the space at the target — a fact that cannot be excluded at reception time — it has to be sent on to the next target with the same name, and the original target disappears.

For this purpose we provide the following auxiliary combinator that shall be used in the definition of `target` and that *attaches* a received object to a target name.

An important case to consider is that the attached object might be a concatenation, since concatenation takes into account the heights of *both* its components before deciding to yield a component. However, if the attached object is too large for the space where it currently resides, it will send itself to the next target — therefore we have to evaluate the attached object with no size constraint so that it may yield its first component without internal size considerations (`isEmpty` determines whether a constraint is `Just 0`):

```
> attach :: String -> Object -> Object
> attach name = attach' where
>  attach' o = o' where
>   o'= Object
>    {eval = (\ forcing co -> case eval o forcing Nothing of
>        Disappearing -> Disappearing
>        NoSpace o1 -> error "attach: NoSpace without constraints!"
>        Suspended o1 -> if isEmpty co
>                        then Sending [(name,Following,attach' o1)] null
>                        else Suspended (attach' o1)
```

```
>        Sending gs o1 -> Sending gs (attach' o1)
>        Yielding c o1 -> if co &? c then Yielding c (attach' o1)
>                  else Sending [(name,Following,attach' (prefix c o1))] null)
>    ,receive = (\ forcing co g -> thrdUpd3 attach' $ receive o forcing co g)
>    ,openTargets = openTargets o
>    ,delayedTargets = delayedTargets o
>    ,height = 0
>    }
```

The `target` function is used to build the basic receptive objects. It therefore disappears when evaluation is forced on it, and suspends otherwise.

When receiving, we have to distinguish whether reception is forced or not — remember that all but the first object of a concatenation have to perform forced reception. The motivation behind this is that, for example, the size of a footnote needs to be known before the text can be allowed to fill the remainder of the page. Therefore the footnote has to expand as far as possible once its galley has reached its target. Again, this expansion involves evaluation without constraints since the part that does not fit into the current target will be sent on to the next target.

```
> target :: String -> Object
> target name = o where
>  o = Object
>   {eval = (\ forcing co -> if forcing then Disappearing
>    else case co of Just 0 -> Disappearing
>                    _ -> Suspended o)
>   ,receive = (\ forcing co g @ (name',d',o') -> case co of
>    _ -> if name /= name' then rcverror g "target"
>         else if not forcing then (True, [], (attach name o' # o))
>         else case eval o' False Nothing of
>           Disappearing -> (True, [], o)
>           Suspended o'' -> (True, [], (attach name o'' # o))
>           NoSpace o'' -> error "target: NoSpace without constraint!"
>           Sending gs1 o'' -> (True, gs1, (attach name o'' # o))
>           Yielding c o'' -> if co &? c then
>              let g' = (name',Following,o'')
>                  (rcv, gs1, o1) = receive (prefix c o) forcing co g'
>              in (True, gs1, o1)
>            else (False, [(name,Following,prefix c o'')], null))
>   ,openTargets = [name]
>   ,delayedTargets = []
>   ,height = 0
>   }
```

The last line of the definition of `receive` reflects the fact that when there is not enough space for the first component of the galley arriving at its target, then the galley is sent on to the next target in forward direction, and the original target disappears.

Objects built with `recurse` and `delay` are both delayed objects[1] and expand only if some contained target receives a galley. Therefore both disappear when forced and suspend on normal evaluation.

As a small concession to the forcing galleys of Lout (which are not directly

[1] In this paper we only consider *receptive* recursive objects. The behaviour of other recursive objects in Lout apparently cannot be integrated easily into the forcing strategy of the current paper.

connected with our forcing parameters) we allow them to disappear, too, when it is clear that there is not enough space for expansion. Since eventually these objects would be forced away in the end, this is not strictly necessary, but in the effect it allows pages to be flushed earlier.

When a recursive object is asked to receive a galley, it first checks whether a single instance of the body of the recursion, with `null` substituted for the recursive call, can receive the galley under the present size constraints. Only if this is possible, the recursion is unfolded one level and the galley sent to the result — note that this tentative evaluation is possible without problems since it cannot provoke any side effects. If reception is impossible, then the galley is sent on and the recursive object disappears — otherwise footnotes might appear out of order.

```
> recurse :: (Object -> Object) -> Object
> recurse ff = o
>  where
>   ffo = ff o
>   ff0 = ff null
>   targs = targets ff0
>   o = Object
>    {eval = (\ forcing co -> if forcing || isEmpty co || not (co &? ffo)
>                             then Disappearing else Suspended o)
>    ,receive = (\ forcing co g @ (name,d,o') -> case co of
>         Just 0 -> (False, [forward g], null)
>         _ -> if name `elem` targs
>              then case receive ff0 forcing co g of
>                   (False, gs, o1) -> (False, [forward g], null)
>                   r -> receive ffo forcing co g
>              else rcverror g "recurse")
>    ,openTargets = []
>    ,delayedTargets = targs
>    ,height = 0
>    }
```

Since it only receives after expansion, a recursive object contains no open targets, and all targets of the body are delayed targets of the recursion.

For objects built with `delay`, the same considerations apply:

```
> delay :: Object -> Object
> delay o = o' where
>  o' = Object
>   {eval = (\ forcing co -> if forcing || isEmpty co || not (co &? o)
>                         then Disappearing else Suspended o')
>   ,receive = (\ forcing co g @ (name,d,o') -> case co of
>         Just 0 -> (False, [forward g], null)
>         _ -> if name `elem` targs
>              then case receive o forcing co g of
>                (False, gs, o1) -> (False, [forward g], null)
>                r -> r
>              else rcverror g "delay")
>   ,openTargets = []
>   ,delayedTargets = targs
>   ,height = 0
>   }
>  targs = targets o
```

5 Extensions

If in our example document format there is a footnote on a page with only little other material, then this footnote will be appended immediately below that material instead of being pushed to the bottom of the page. Since we do not have anything corresponding to the gap modes of lout, which influence the space behaviour of concatenation and can be used for such purposes, we might instead want to define

```
> vfill = recurse (prefix [""])
```

and insert this on top of `footSect` for pushing the footnote sections to the bottom of the page.

However, as mentioned before, this un-receptive recursion does not work well with our present evaluation strategy. Nevertheless we can introduce a combinator roughly corresponding to the Lout symbol @VExpand that makes an object take up all space available for it (as long as that space is finite):

```
> vExpand :: Object -> Object
> vExpand o = o' where
>  o' = Object
>   {eval = (\ forcing co -> case co of
>     Nothing ->     eval o forcing co
>     Just 0 ->      eval o forcing co
>     Just h -> case eval o forcing co of
>       Disappearing -> Yielding (strut h) null
>       NoSpace o1 -> NoSpace o1
>       Sending gs o1 -> Sending gs (vExpand o1)
>       Suspended o1 -> Suspended (vExpand o1)
>       Yielding c o1 -> Yielding c (if length c < h then vExpand o1 else o1))
>   ,receive = (\ frc co g -> thrdUpd3 vExpand (receive o frc co g))
>   ,openTargets = openTargets o
>   ,delayedTargets = delayedTargets o
>   ,height = height o
>   }
```

If we now modify the definition of `page` accordingly, then footnotes are always pushed to the bottom of the page:

```
> page = high 12 (vExpand (target "TextPlace") # footSect)
```

Another extension that we would like to present will allow us to number our pages.

We give a variant of the `recurse` combinator that uses functions from arbitrary domains to `Objects` instead of just `Objects`. The definition is unchanged except for the framework needed to set up the two expanded objects:

```
> recurseF :: ((a -> Object) -> (a -> Object)) -> (a -> Object)
> recurseF ff = f
>  where
>   f' = ff f
>   f a = o where
>    ffo = f' a
>    ff0 = ff (const null) a
>    targs = targets ff0
>    o = Object -- { as before }
```

The modifications necessary to obtain numbered pages are now very easy: the numbered page itself is obtained via a Haskell function taking the page number and delivering a numbered page (including the `vExpand` from above), and the list of numbered pages

is a recursion over a function that maps a page-list generator `mkpl` to another page-list generator that generates a numbered page concatenated with a recursive call using an incremented page number:

```
> npage :: Int -> Object
> npage n = high 14 $ prefix ["          - " ++ show n ++ "-",""]
>                            (vExpand (target "TextPlace") # footSect)

> npageList :: Object
> npageList = let f mkpl n = npage n # mkpl (n+1)
>             in recurseF f 1
```

6 Concluding Remarks

Motivated by the galley concept implemented in Lout, we have presented Haskell definitions of eight elementary and two advanced document formatting combinators. These combinators allow to put together document formats that include page numbering, footnotes and tables of contents in a simple, declarative and intuitive way. We have not strived to mimick exactly the behaviour of Lout, but have instead tried to construct a self-contained, intuitively motivated and explainable implementation of the document formatting combinators. The definitions of those combinators, although they are not trivial and may not yet be the most elegant, still fitted into the space constraints set for this paper and show the elegance of Kingston's galley abstraction and how this may be reflected in a purely functional formalisation.

Especially the extension to recurse over object-yielding functions also shows how document formatting can profit from being embedded into a full-fledged functional programming language, just as with many other domain-specific languages.

Although much still needs to be done to be able to reflect also the other abstractions underlying Lout in a Haskell setting, these combinators perhaps may serve to illuminate a little bit the path towards Jeff Kingston's "vapourware successor to Lout":

– A Haskell implementation of Lout or its successor would allow compiled document formats, so that only the document itself needed to be interpreted.
– Furthermore, document format authors could be given the choice whether to program in Lout or directly in Haskell, where the full power of Haskell would be at their fingertips.
– The concise functional specification could also serve as a more rigorous yet accessible documentation — the Lout mailing list abounds of postings by users baffled by the intricacies of the galley flushing algorithm.

An interesting application of the galley concept could also be found in literate programming: There variants of the same document parts are directed at different targets, namely program code files and documentation, very much like section headings are directed to their place above the section and to the table of contents.

Such an extension will be much easier to explore from within a Haskell setting than from the C implementation — one point to take care of is that we relied on the absence of side-effects in the definition of `recurse`, so that output to files either should not happen during `receives`, or the defintion of `recurse` (and the forcing strategy) would have to be adapted.

Since the document formatting combinators presented in this paper are purely functional, and since their interaction is rather complicated, it is not easy to find the reasons for unexpected behaviour during development, especially since the relation between cause and effect may be very indirect and obscure. Since the author does not have access to any usable Haskell debugger or tracer, the Haskell definitions of this paper have been translated into term graph rewriting rules for the graphically interactive strongly typed second-order term graph transformation system HOPS [4,5]. Being able to *watch* things go wrong was invaluable for tracking down subtle design errors.

This paper has been typeset using the Lout document formatting system. Lout definitions and an auxiliary Haskell program provided by the author turn this paper into a literate program; the Haskell code generated from this paper with the use of Lout and FunnelWeb [12] is available on the WWW at

URL: http://diogenes.informatik.unibw-muenchen.de/kahl/Haskell/VGalleys.html.

References

1. Pablo R. Azero, S. Doaitse Swierstra. Optimal Pretty-Printing Combinators, 1998. URL http://www.cs.ruu.nl/groups/ST/Software/PP/.
2. John Hughes. Pretty-printing: An Exercise in Functional Programming. In R. S. Bird, C. C. Morgan, J. C. P. Woodcock (eds.), *Mathematics of Program Construction*, pages 11–13. LNCS 669. Springer-Verlag, 1992.
3. John Hughes. The Design of a Pretty-printing Library. In J. Jeuring, E. Meijer (eds.), *Advanced Functional Programming*, pages 53–96. LNCS. Springer-Verlag, 1995.
4. Wolfram Kahl. The **H**igher **O**bject **P**rogramming **S**ystem — User Manual for HOPS, Fakultät für Informatik, Universität der Bundeswehr München, February 1998. URL http://diogenes.informatik.unibw-muenchen.de:8080/kahl/HOPS/.
5. Wolfram Kahl. Internally Typed Second-Order Term Graphs. In J. Hromkovic, O. Sýkora (eds.), *Graph Theoretic Concepts in Computer Science, WG '98*, pages 149–163. LNCS 1517. Springer-Verlag, 1998.
6. Jeffrey H. Kingston. The design and implementation of the Lout document formatting language. *Software — Practice and Experience* **23**, 1001–1041 (1993).
7. Jeffrey H. Kingston. *An Expert's Guide to the Lout Document Formatting System (Version 3).* Basser Department of Computer Science, University of Sydney, 1995.
8. Jeffrey H. Kingston. *A User's Guide to the Lout Document Formatting System (Version 3.12).* Basser Department of Computer Science, University of Sydney. ISBN 0 86758 951 5, 1998. URL ftp://ftp.cs.su.oz.au/jeff/lout.
9. Donald E. Knuth. *The TEXBook*. Addison-Wesley, 1984.
10. Simon Peyton Jones. A Pretty Printer Library in Haskell, Version 3.0, 1997. URL http://www.dcs.gla.ac.uk/~simonpj/pretty.html.
11. Philip Wadler. A Prettier Printer, 1998. URL http://cm.bell-labs.com/cm/cs/who/wadler/papers/prettier/. Draft paper
12. Ross N. Williams. FunnelWeb User's Manual, May 1992. URL http://www.ross.net/funnelweb/introduction.html. Part of the FunnelWeb distribution