The Design of a Pretty-printing Library

John Hughes

Chalmers Tekniska Högskola, Göteborg, Sweden.

1 Introduction

On what does the power of functional programming depend? Why are functional programs so often a fraction of the size of equivalent programs in other languages? Why are they so easy to write? I claim: because functional languages support software reuse extremely well.

Programs are constructed by putting program components together. When we discuss reuse, we should ask

- What kind of components can be given a name and reused, rather than reconstructed at each use?
- How flexibly can each component be used?

Every programming language worthy of the name allows sections of a program with identical control flow to be shared, by defining and reusing a procedure. But 'programming idioms' — for example looping over an array — often cannot be defined as procedures because the repeated part (the loop construct) contains a varying part (the loop body) which is different at every instance. In a functional language there is no problem: we can define a *higher-order function*, in which the varying part is passed as a function-valued parameter. This ability to name and reuse programming idioms is at the heart of functional languages' power.

Other features contribute to making reused components more flexible. *Polymorphic typing* enables us to use the *same* programming idiom to manipulate data of *different* types. *Lazy evaluation* abstracts away from execution time, and enables us to reuse the same function with many different behaviours. For example, a lazy list can behave like an array (a sequence of elements stored at the same time), or like an imperative variable (a sequence of values stored at different times), or like something in between (say a buffer in which a bounded number of elements are stored at any one time). Regardless of behaviour the same functions can be used to manipulate the list.

Software reuse is plainly visible in functional programs: for example, the Haskell standard prelude contains many higher-order functions such as *map*, *foldr* etc., which are used intensively in many programs. These standard functions capture very general programming idioms that are useful in almost any context. But it is just as important to define and use *application specific* idioms.

The functional programmer, then, should approach a new application by seeking to identify the programming idioms common in that application area, and to define them as (probably higher order) functions. Each particular application program should then be built by so far as possible combining these functions, rather than writing 'new code'. (Perhaps for this reason, such functions are often called *combinators*). The benefits of such an approach are very rapid programming, once the library of idioms is defined, and very often that application programs are correct first time, since they are built by assembling correct components.

One example of an application area whose idioms have been thoroughly studied is parsing: libraries of parsing combinators are described in this volume. Another good example on a much larger scale is Carlsson and Hallgren's *fudget* library, also described here, which enables graphical user interfaces to be constructed very easily.

The question we address in this chapter is: how should libraries of combinators be designed? How do we know which operations to provide? Monads, also explained in this volume, are certainly helpful — but how do we know which monad to use? Must we rely completely on intuition?

Our goal is to show how we can use formal specification of the combinators, and a study of their algebraic properties, to guide both the design and the implementation of a combinator library. Our case study is a library for pretty-printing, which has gone through many iterations and been much improved by a more formal approach. But we hope the methods we present are of wider applicability, and we will also present some smaller examples to justify this claim.

2 A Preview of the Pretty-printing Library

2.1 Why Pretty-printing?

Almost every program which manipulates symbolic data needs to display this data to the user at some point — whether it is a compiler displaying internal structures for debugging, a proof-editor displaying proofs, or a program transformer writing its output. The problem of displaying symbolic, and especially tree structured data, is thus a recurring one.

At the same time, structured data is hard to read unless layout is used to make the structure visible. Take a simple example: a binary tree of type[1].

$$\mathbf{data}\ \mathit{Tree} = \mathit{Node}\ \mathit{String}\ \mathit{Tree}\ \mathit{Tree} \mid \mathit{Leaf}$$

The tree *Node* "foo" (*Node* "baz" *Leaf Leaf*) (*Node* "foobaz" *Leaf Leaf*) is much easier to read if it is presented as

```
Node "foo"  (Node "baz"  Leaf Leaf)
            (Node "foobaz"  Leaf Leaf)
```

A pretty-printer's job is to lay out structured data appropriately.

Pretty-printing is complicated because the layout of a node cannot just be inferred from its form. In the example above, *Node*s are laid out in two different ways: some horizontally and some vertically. Moreover the correct indentation of the final *Node* depends on the length of the string in the parent node. A pretty-printer must keep track of much contextual information.

Because of this pretty-printers are hard to write, and there is plenty of scope for mistakes. Many programmers simply do not bother — they put up with badly formatted output instead. There is much to be gained by capturing the hard part of pretty-printing in a library.

[1] All examples in this chapter use Haskell syntax

Remark Note that we are considering the problem of displaying internal data-structures in a readable form, not the harder problem of improving the layout of an existing text, such as a program. In the latter case we would have to consider questions such as: should we try to preserve anything of the original layout? How should we handle comments? Such problems are outside the scope of this chapter.

2.2 A Sketch of the Design

What kind of objects should pretty-printing combinators manipulate? I chose to work with 'pretty documents', of type *Doc*, which we can think of as documents which 'know how to' lay themselves out prettily. A pretty-printer for a particular datatype is a function mapping any value to a suitable *Doc*. The library provides operations for constructing *Doc*s in various ways, and for converting a *Doc* to text at the top level.

We will need to convert literal strings to *Doc*s, and it seems reasonable to provide operations that combine *Doc*s horizontally and vertically. That suggests we provide operations

$$\begin{array}{ll}
\mathit{text} :: \mathit{String} \rightarrow \mathit{Doc} & \\
(<>) :: \mathit{Doc} \rightarrow \mathit{Doc} \rightarrow \mathit{Doc} & \text{[horizontal composition]} \\
(\$\$) :: \mathit{Doc} \rightarrow \mathit{Doc} \rightarrow \mathit{Doc} & \text{[vertical composition]}
\end{array}$$

The composition operators (<>) and ($$) relieve the user of the need to think about the correct indentation: for example, the pretty tree layout above can be constructed as

$$\begin{array}{l}
\mathit{text} \text{ ``}\mathit{Node}\ \text{``}\mathsf{foo}\text{'' ''} <> (\mathit{text}\ \text{``}\mathit{Node}\ \text{``}\mathsf{baz}\text{''}\ \mathit{Leaf\ Leaf}\text{''}\ \$\$ \\
\qquad\qquad\qquad\qquad\quad \mathit{text}\ \text{``}\mathit{Node}\ \text{``}\mathsf{foobaz}\text{''}\ \mathit{Leaf\ Leaf}\text{''})
\end{array}$$

and the last *Node* is automatically indented the right amount.

However, these operations only enable us to construct *Doc*s with a fixed layout. We also need to construct *Doc*s that choose between alternative layouts depending on the context. We will therefore define

$$\mathit{sep} :: [\mathit{Doc}] \rightarrow \mathit{Doc}$$

which combines a list of *Doc*s horizontally or vertically, depending on the context.

With these operations we can write a pretty-printer for the tree type above:

$$\begin{array}{l}
\mathit{pp} :: \mathit{Tree} \rightarrow \mathit{Doc} \\
\mathit{pp}\ \mathit{Leaf} = \mathit{text}\ \text{``}\mathsf{Leaf}\text{''} \\
\mathit{pp}\ (\mathit{Node}\ s\ l\ r) = \mathit{text}\ (\text{``}\mathsf{Node}\ \text{''} +\!\!+ s) <> \mathit{sep}\ [\mathit{pp'}\ l, \mathit{pp'}\ r] \\
\mathit{pp'}\ \mathit{Leaf} = \mathit{pp}\ \mathit{Leaf} \\
\mathit{pp'}\ t = \mathit{text}\ \text{``(''} <> \mathit{pp}\ t <> \text{``)''}
\end{array}$$

The context-dependent choice of layout is entirely hidden in the implementation of the *Doc* type — the only complication is deciding when to insert brackets.

The library provides one further operation,

$$\mathit{nest} :: \mathit{Int} \rightarrow \mathit{Doc} \rightarrow \mathit{Doc}$$

which indents a document a given number of spaces. For example,

$$\textit{text}\ \text{“while x>0 do”}\ \$\$\ \textit{nest}\ 2\ (\textit{text}\ \text{“x := x-2”})$$

produces the layout

```
while x>0 do
  x := x-2
```

The difference between using *nest* and inserting spaces is that *nest* indents only where it is appropriate — so for example,

$$\textit{sep}\ [\textit{text}\ \text{“while x>0 do”},\ \textit{nest}\ 2\ (\textit{text}\ \text{“x := x-2”})]$$

will appear as above laid out vertically, but without indentation as

```
while x>0 do x := x-2
```

if laid out horizontally.

This choice of combinators was made quite early on in the development of the library, and the first implementation was written from a description more or less like the one just given. But the description is far from satisfactory: although the intention of the design is fairly clear, the precise behaviour of the combinators is certainly not. Not surprisingly, this led to a number of difficulties and strange behaviours.

Later on we will give a precise specification of the combinators' behaviour, and use this to derive several alternative implementations. But before we continue with this larger case study, we'll present some simpler examples to illustrate the methods we will be using.

3 Deriving Functional Programs from Specifications

How can we conveniently use a specification to help develop a functional program? Let us suppose that the specification consists of a *signature*, containing possibly new types such as *Doc* and the names and types of the functions being specified, and properties that the new functions must satisfy. Our task is to invent representations of the new types and definitions of the functions so that the properties are satisfied. We will call functions from the new types to old types *observations*. Observations are important: if there are none then we cannot distinguish between values of the new types, and so we can represent them all by (). We will assume that the specification determines the value of every possible observation — if not, we must strengthen the specification until it does.

The implementations which we are trying to derive consist of equations of a restricted form. We will derive implementations by proving their constituent equations from the specification. By itself this is no guarantee that the implemented functions satisfy the specification (because we might not have proved *enough* equations). But if we also check that the derived definitions are *terminating* and *exhaustive*, then this property is guaranteed.

To see why, consider the case of a single function f. We start from a specification $P(f)$ and derive implementation equations $Q(f)$, both considered as predicates on f.

By construction $P(f) \Rightarrow Q(f)$. But in general, the implementation equations $Q(f)$ might be satisfied by many different functions, of which the *least* is the one that the equations define. Call this least function f_{imp}. Now, if the derived definitions are exhaustive and terminating, then for any argument x, f_{imp}/x is a defined value y and $Q(f) \Rightarrow f\ x = y$. In other words $Q(f) \Rightarrow f = f_{imp}$ — the implementation equations have a unique solution. Now if the specification is satisfied by any f at all, we know that

$$P(f) \Rightarrow Q(f) \Rightarrow f = f_{imp}$$

and therefore $P(f_{imp})$ holds — the implementation satisfies the specification.

Since we will use the specification to derive equations, it will be most convenient if the specification also consists of equations — or laws — that the new functions are to satisfy.

But before we can start deriving implementations of functions we must choose a representation for each new type. We will present two different ways of choosing such a representation. The first is based on representing values by *terms* in the algebra we are working with. The second is based on representing values by *functions* from the context in which the value is placed to the value of the corresponding observation.

4 Designing a Sequence Type

We begin by considering a very simple and familiar example: the design of a representation for sequences. Of course we know how to represent sequences — as lists. The point here is not to discover a new representation, but to see how we could have arrived at the well-known representation of lists starting from an algebraic specification.

We take the following signature as our starting point,

$$\begin{array}{l}
\mathit{nil} :: \mathit{Seq}\ a \\
\mathit{unit} :: a \to \mathit{Seq}\ a \\
\mathit{cat} :: \mathit{Seq}\ a \to \mathit{Seq}\ a \to \mathit{Seq}\ a \\
\mathit{list} :: \mathit{Seq}\ a \to [a]
\end{array}$$

where *nil*, *unit*, and *cat* give us ways to build sequences, and *list* is an observation. The correspondence with the usual list operations is

$$\begin{aligned}
\mathit{nil} &= [] \\
\mathit{unit}\ x &= [x] \\
\mathit{cat} &= (+\!\!+)
\end{aligned}$$

These operations are to satisfy the following laws[2]:

$$\begin{aligned}
xs\ \text{`}\mathit{cat}\text{`}\ (ys\ \text{`}\mathit{cat}\text{`}\ zs) &= (xs\ \text{`}\mathit{cat}\text{`}\ ys)\ \text{`}\mathit{cat}\text{`}\ zs \\
\mathit{nil}\ \text{`}\mathit{cat}\text{`}\ xs &= xs \\
xs\ \text{`}\mathit{cat}\text{`}\ \mathit{nil} &= xs \\
\mathit{list}\ \mathit{nil} &= [] \\
\mathit{list}\ (\mathit{unit}\ x\ \text{`}\mathit{cat}\text{`}\ xs) &= x : \mathit{list}\ xs
\end{aligned}$$

[2] Haskell allows a binary function to be used as an infix operator if the name is enclosed in backquotes. Thus $a\ \text{`}op\text{`}b$ is the same as $op\ a\ b$

4.1 Term Representation

The most direct way to represent values of sequence type is just as terms of the albegra, for example using

$$\mathbf{data}\ \mathit{Seq}\ a = \mathit{Nil} \mid \mathit{Unit}\ a \mid \mathit{Seq}\ a\ \text{`}\mathit{Cat}\text{`}\ \mathit{Seq}\ a$$

But this trivial representation does not exploit the algebraic laws that we know to hold, and moreover the *list* observation will be a little tricky to define (ideally we would like to implement observations by very simple, non-recursive functions: the real work should be done in the implementations of the *Seq* operators themselves). Instead, we may choose a restricted subset of terms — call them simplified forms[3] — into which every term can be put using the algebraic laws. Then we can represent sequences using a datatype that represents the syntax of simplified forms.

In this case, there is an obvious candidate for simplified forms: terms of the form *nil* and *unit x 'cat' xs*, where *xs* is also in simplified form. Simplified forms can be represented using the type

$$\mathbf{data}\ \mathit{Seq}\ a = \mathit{Nil} \mid a\ \text{`}\mathit{UnitCat}\text{`}\ \mathit{Seq}\ a$$

with the interpretation[4]

$$\begin{aligned} \mathit{Nil} &= \mathit{nil} \\ x\ \text{`}\mathit{UnitCat}\text{`}\ xs &= \mathit{unit}\ x\ \text{`}\mathit{cat}\text{`}\ xs \end{aligned}$$

We choose this representation because a definition of *list* is now very simple to derive:

$$\begin{aligned} \mathit{list}\ \mathit{Nil} &= \mathit{list}\ \mathit{nil} \\ &= [] \\ \mathit{list}\ (x\ \text{`}\mathit{UnitCat}\text{`}\ xs) &= \mathit{list}\ (\mathit{unit}\ x\ \text{`}\mathit{cat}\text{`}\ xs) \\ &= x : \mathit{list}\ xs \end{aligned}$$

We can also derive implementations of the three operators of the algebra by simply applying the algebraic laws:

$$\mathit{nil} = \mathit{Nil} \qquad [\text{defn. } \mathit{Nil}]$$

$$\begin{aligned} \mathit{unit}\ x &= \mathit{unit}\ x\ \text{`}\mathit{cat}\text{`}\ \mathit{nil} \\ &= x\ \text{`}\mathit{UnitCat}\text{`}\ \mathit{Nil} \qquad [\text{defn. } \mathit{UnitCat}] \end{aligned}$$

$$\begin{aligned} \mathit{Nil}\ \text{`}\mathit{cat}\text{`}\ ys &= \mathit{nil}\ \text{`}\mathit{cat}\text{`}\ ys \\ &= ys \end{aligned}$$

$$\begin{aligned} (x\ \text{`}\mathit{UnitCat}\text{`}\ xs)\ \text{`}\mathit{cat}\text{`}\ ys &= (\mathit{unit}\ x\ \text{`}\mathit{cat}\text{`}\ xs)\ \text{`}\mathit{cat}\text{`}\ ys \\ &= \mathit{unit}\ x\ \text{`}\mathit{cat}\text{`}\ (xs\ \text{`}\mathit{cat}\text{`}\ ys) \qquad [\text{associativity}] \\ &= x\ \text{`}\mathit{UnitCat}\text{`}\ (xs\ \text{`}\mathit{cat}\text{`}\ ys) \qquad [\text{defn. } \mathit{UnitCat}] \end{aligned}$$

```
data Seq a = Nil | a ‘UnitCat‘ Seq a

nil = Nil

Nil ‘cat‘ ys = ys
(x ‘UnitCat‘ xs) ‘cat‘ ys = x ‘UnitCat‘ (xs ‘cat‘ ys)

list Nil = []
list (x ‘UnitCat‘ xs) = x : list xs
```

Fig. 1. Term representation of sequences.

Collecting the results we obtain the definitions in figure 1. Termination of each function is obvious.

How do we know that every *Seq* term can be expressed as a simplified form? The definitions we have derived are a proof! Since each function maps simplified arguments to simplified results (and always terminates), we can construct a simplified form equal to any term just by evaluating it with these definitions. In more complicated algebras this observation is valuable: when we're choosing a simplified form we need not worry whether all terms can be put into it — we simply try to derive terminating definitions for the operations, and if we succeed, the result follows.

So far we've just derived the usual implementation of lists — *Nil* and *UnitCat* correspond to [] and (:). But notice that it isn't without its problems: the implementation of *cat* is linear in its first argument, and we run into the well known problem that an expression such as

$$(\ldots(\mathit{unit}\ x_1\ \text{‘}\mathit{cat}\text{‘}\ \mathit{unit}\ x_2)\ldots\ \text{‘}\mathit{cat}\text{‘}\ \mathit{unit}\ x_{n-1})\ \text{‘}\mathit{cat}\text{‘}\ \mathit{unit}\ x_n$$

takes quadratic time to evaluate. Using the associative law n times we can obtain the equivalent expression

$$\mathit{unit}\ x_1\ \text{‘}\mathit{cat}\text{‘}(\mathit{unit}\ x_2\ \text{‘}\mathit{cat}\text{‘}\ \ldots(\mathit{unit}\ x_{n-1}\ \text{‘}\mathit{cat}\text{‘}\ \mathit{unit}\ x_n))$$

which runs in linear time. We might hope to exploit the associative law in an improved implementation that achieves the better complexity in the first case also. We could try to derive an implementation of *cat* that recognises *cat* in its left argument, and applies the associative law before continuing. But alas, if we are to recognise applications of *cat* then they must be simplified forms, which means that the *cat* operation can do nothing; we are forced back to the trivial representation we started with. In the next section we look at a different approach which can exploit associativity in this case.

[3] We avoid the term ‘canonical form’ because in general there's no reason why a term need have a *unique* simplified form.

[4] Here we really mean the *semantics* of *Nil* and *UnitCat*, and by equality we mean equality in the algebra we are implementing — not necessarily Haskell's equality. Perhaps it would be more conventional to write $[\![\mathit{Nil}]\!]$ and $[\![\mathit{UnitCat}]\!]$ here, but we prefer to identify syntax and semantics in the interests of lightening the notation.

4.2 Context Passing Representation

If we can't apply the associative law by making the outer *cat* recognise that its left argument is a *cat*, perhaps we can make the inner *cat* recognise that it is called in a *cat* context. This idea motivates a representation of sequences as functions from their context to the observation being made.

A *context* is just an expression with a hole, written $[\bullet]$. For example, $[\bullet]$ *'cat' ys* is a context. If $C[\bullet]$ is a context and e is an expression, we write $C[e]$ for the result of replacing the hole with e. In this case $([\bullet]\ \textit{`cat`}\ \textit{ys})[\textit{xs}]$ is *xs 'cat' ys*.

We can describe the contexts we are interested in by a grammar. For example, the following grammar describes all possible contexts of type list for expressions of type *Seq* .

$$\begin{aligned} C[\bullet] ::= &\ \textit{list}\ [\bullet] \\ \mid &\ C[[\bullet]\ \textit{`cat`}\ E] \\ \mid &\ C[E\ \textit{`cat`}\ [\bullet]] \end{aligned}$$

where E is an expression of *Seq* type. And just as with terms, we can represent contexts by a corresponding Haskell datatype:

$$\textbf{data}\ \textit{Cxt}\ a = \textit{List} \mid \textit{CatLeft}\ (\textit{Seq}\ a)\ (\textit{Cxt}\ a) \mid \textit{CatRight}\ (\textit{Seq}\ a)\ (\textit{Cxt}\ a)$$

where

$$\begin{aligned} \textit{List} &= \textit{list}\ [\bullet] \\ \textit{CatLeft}\ E\ C &= C[[\bullet]\ \textit{`cat`}\ E] \\ \textit{CatRight}\ E\ C &= C[E\ \textit{`cat`}\ [\bullet]] \end{aligned}$$

Notice that the representation of, say, a *CatLeft* context *contains* the representation of the enclosing context; contexts resemble therefore a stack. Notice also that the context type must be parameterised on a because it refers to *Seq a*.

In fact, just as we used the laws to work with a restricted set of terms, we shall use the laws to work with a restricted set of contexts. For our purposes in this example, we will only need to consider contexts of the form

$$C[\bullet] ::= \textit{list}\ [\bullet] \mid \textit{list}\ ([\bullet]\ \textit{`cat`}\ E)$$

represented by the following datatype:

$$\textbf{data}\ \textit{Cxt}\ a = \textit{List} \mid \textit{ListCat}\ (\textit{Seq}\ a)$$

Now we can represent sequence values by functions from contexts to lists: the value e is represented by the function $\lambda C[\bullet].C[e]$. (So contexts are like continuations whose internal structure can be inspected). For example,

$$\textit{nil} = \lambda C[\bullet].C[\textit{nil}]$$

where again we make no notational distinction between the *nil* on the left, which is a representation, and the *nil* on the right, which is a semantic object. When we apply this representation to a context, we derive for example

$$\begin{aligned} nil\ (ListCat\ zs) &= nil\ (list\ ([\bullet]\ \text{`}cat\text{`}\ zs)) && [\text{defn. } ListCat] \\ &= list\ (nil\ \text{`}cat\text{`}\ zs) && [\text{defn. } nil] \end{aligned}$$

In future we will switch backwards and forwards between the first and last form in one step, and without comment. We can derive definitions of the operators using the laws of the algebra as before:

$$\begin{aligned} nil\ List &= list\ nil \\ &= [\,] \end{aligned}$$

$$\begin{aligned} nil\ (ListCat\ zs) &= list\ (nil\ \text{`}cat\text{`}\ zs) \\ &= list\ zs \\ &= zs\ List \end{aligned}$$

$$\begin{aligned} unit\ x\ \ List &= list\ (unit\ x) \\ &= list\ (unit\ x\ \text{`}cat\text{`}\ nil) \\ &= x : list\ nil \\ &= [x] \end{aligned}$$

$$\begin{aligned} unit\ x\ \ (ListCat\ zs) &= list\ (unit\ x\ \text{`}cat\text{`}\ zs) \\ &= x : list\ zs \\ &= x : zs\ List \end{aligned}$$

$$\begin{aligned} (xs\ \text{`}cat\text{`}\ ys)\ List &= list\ (xs\ \text{`}cat\text{`}\ ys) \\ &= xs\ (ListCat\ ys) \end{aligned}$$

$$\begin{aligned} (xs\ \text{`}cat\text{`}\ ys)\ (ListCat\ zs) &= list\ ((xs\ \text{`}cat\text{`}\ ys)\ \text{`}cat\text{`}\ zs) \\ &= list\ (xs\ \text{`}cat\text{`}\ (ys\ \text{`}cat\text{`}\ zs)) && [\text{assoc!}] \\ &= xs\ (ListCat\ (ys\ \text{`}cat\text{`}\ zs)) \end{aligned}$$

Notice that the derived definition of *cat* recognises an enclosing *cat* and applies the associative law — just the optimisation we wanted to capture. Gathering the results together, we obtain the implementation shown in figure 2.

We can show that these definitions terminate, and moreover derive their complexity, by considering a suitable cost measure on terms. We construct the cost measure so that every reduction strictly reduces cost.

Start by observing that terms not containing *cat* or *ListCat* are reduced to a normal form in one step. We'll give such terms a cost of zero. Now notice that the second equations defining *nil* and *unit* eliminate a *ListCat*. If *ListCat* is assigned a cost of one, then these reductions reduce cost. Looking at the definition of *cat*, we see that the first equation converts a *cat* to a *ListCat*. If we assign *cat* a cost of two, then this reduction also reduces cost. The tricky case is the second equation for *cat*, since it neither reduces the number of occurrences of *cat* nor of *ListCat*.

We can obtain a cost reduction in this case also by assigning *different costs* to the occurrences of *cat* on the left and right hand side. We assign *cat* a cost of two in a 'cheap' context, and a cost of three in other contexts. Cheap contexts are defined by the following grammar:

$$\begin{aligned} Cheap[\bullet] ::= {} & [\bullet]\ List \\ \mid {} & ListCat\ [\bullet] \\ \mid {} & Cheap[E\ \text{`}cat\text{`}\ [\bullet]] \end{aligned}$$

```
data Cxt a = List | ListCat (Seq a)
type Seq a = Cxt a → [a]

nil List = []
nil (ListCat zs) = zs List

unit x List = [x]
unit x (ListCat zs) = x : zs List

(xs‘cat‘ys) List = xs (ListCat ys)
(xs‘cat‘ys) (ListCat zs) = xs (ListCat (ys‘cat‘zs))

list xs = xs List
```

Fig. 2. The context passing implementation of sequences.

Now it is easy to verify that the *cat* on the right in the last equation is in a cheap context, while that on the left is not. We also have to check that in every equation, bound variables appear in a cheap context on the left hand side iff they appear in a cheap context on the right hand side — otherwise our implicit assumption that a bound variable contributes the same cost at each occurrence would be false. Having done so, we know that the number of reductions needed to evaluate a term is bounded by its cost. And this is linear in the size of the term.

We have therefore cured the quadratic behaviour that motivated us to consider a context-passing implementation.

4.3 Changing the Representation of Contexts

If we examine the definitions in figure 2, we can see that the *zs* component of *ListCat zs* is only used by applying it to *List*. That is, we are not interested in the value of *zs* itself, only in the value of *list zs*. This suggests that we try changing the representation of contexts to store the latter rather than the former.

The new context datatype will therefore be

$$\mathbf{data}\ \mathit{Cxt}\ a = \mathit{List} \mid \mathit{ListCat}\ [a]$$

with the interpretation

$$\begin{aligned} \mathit{List} &= \mathit{list}\ [\bullet] \\ \mathit{ListCat}\ (\mathit{list}\ \mathit{zs}) &= \mathit{list}\ ([\bullet]\ \text{‘}\mathit{cat}\text{‘}\ \mathit{zs}) \end{aligned}$$

Now if we let $\hat{zs} = \mathit{list}\ \mathit{zs}$, we can derive

$$\begin{aligned} \mathit{nil}\ (\mathit{ListCat}\ \hat{zs}) &= \mathit{list}\ (\mathit{nil}\ \text{‘}\mathit{cat}\text{‘}\ \mathit{zs}) \\ &= \mathit{list}\ \mathit{zs} \\ &= \hat{zs} \end{aligned}$$

$$\begin{aligned}
\mathit{unit}\ x\ (\mathit{ListCat}\ \hat{zs}) &= \mathit{list}\ (\mathit{unit}\ x\ \text{‘}\mathit{cat}\text{‘}\ zs) \\
&= x : \mathit{list}\ zs \\
&= x : \hat{zs}
\end{aligned}$$

$$\begin{aligned}
(xs\ \text{‘}\mathit{cat}\text{‘}\ ys)\ (\mathit{ListCat}\ \hat{zs}) &= \mathit{list}\ ((xs\ \text{‘}\mathit{cat}\text{‘}\ ys)\ \text{‘}\mathit{cat}\text{‘}\ zs) \\
&= \mathit{list}\ (xs\ \text{‘}\mathit{cat}\text{‘}\ (ys\ \text{‘}\mathit{cat}\text{‘}\ zs)) \\
&= xs(\mathit{ListCat}\ (\mathit{list}\ (ys\ \text{‘}\mathit{cat}\text{‘}\ zs)) \\
&= xs(\mathit{ListCat}\ (ys(\mathit{ListCat}\ (\mathit{list}\ zs)))) \\
&= xs(\mathit{ListCat}\ (ys(\mathit{ListCat}\ \hat{zs})))
\end{aligned}$$

Notice how each time we introduce a *ListCat*, the accompanying application of *list* enables a further simplification.

In each case we have succeeded in maneouvering the right hand side into a form in which zs does not appear — only $\hat{zs}$. We can therefore take the derived equations as definitions, with a formal parameter $\hat{zs}$. Provided, of course, that contexts of the form $\mathit{ListCat}\ \hat{zs}$ always satisfy the invariant $\exists zs.\hat{zs} = \mathit{list}\ zs$, which is easily verified.

In this case we can go a little further still. Noting that

$$\begin{aligned}
\mathit{list}\ xs &= \mathit{list}\ (xs\ \text{‘}\mathit{cat}\text{‘}\ \mathit{nil}) \\
&= xs(\mathit{ListCat}\ (\mathit{list}\ \mathit{nil})) \\
&= xs(\mathit{ListCat}\ [])
\end{aligned}$$

we can redefine *list* and do without *List* contexts altogether. Now since only one form of context remains we can drop the *ListCat* constructor also, and represent contexts just by lists. The resulting definitions appear in figure 3.

$$\begin{aligned}
&\textbf{type}\ \mathit{Cxt}\ a = [a] \\
&\textbf{type}\ \mathit{Seq}\ a = \mathit{Cxt}\ a \to [a] \\
&\mathit{nil}\ \hat{zs} = \hat{zs} \\
&\mathit{unit}\ x\ \hat{zs} = x : \hat{zs} \\
&(xs\ \text{‘}\mathit{cat}\text{‘}\ ys)\ \hat{zs} = xs\ (ys\ \hat{zs}) \\
&\mathit{list}\ xs = xs\ []
\end{aligned}$$

Fig. 3. Optimised context passing representation of sequences

Exercise 1. Could we have used a similar trick to eliminate *List* contexts and the *ListCat* constructor in the previous section?

5 Implementing Monads

The ideas in the previous section are applicable when we want to implement a datatype specified by a signature and some equations that the operations in the signature should satisfy. One very interesting class of datatypes specified in this way are *monads*. At its simplest, a monad is a parameterised type M and a pair of operations

$$\begin{aligned} unit &:: a \rightarrow M\ a \\ bind &:: M\ a \rightarrow (a \rightarrow M\ b) \rightarrow M\ b \end{aligned}$$

satisfying the laws

$$\begin{aligned} unit\ x\ \text{`}bind\text{`}\ f &= f\ x \\ m\ \text{`}bind\text{`}\ unit &= m \\ m\ \text{`}bind\text{`}\ \lambda x \rightarrow (f\ x\ \text{`}bind\text{`}\ g) &= (m\ \text{`}bind\text{`}\ \lambda x \rightarrow fx)\ \text{`}bind\text{`}\ g \end{aligned}$$

See the chapter by Wadler in this volume for an exposition of the uses of monads in functional programming.

With no further operations a monad is rather uninteresting. In reality, we always extend the signature with some additional operations. In particular, there must be some way to observe a monad value — otherwise we could implement the monad by

$$\begin{array}{l} \textbf{type}\ M\ a = () \\ unit\ x = () \\ m\ \text{`}bind\text{`}\ f = () \end{array}$$

which satisfies the monad laws.

We will consider the simplest interesting monad: that with one additional operation

$$value :: M\ a \rightarrow a$$

satisfying the law

$$value\ (unit\ x) = x$$

We'll look at implementations based on simplified terms and on context passing.

5.1 The Term Representation of a Simple Monad

Suppose we try to represent monad values directly by terms:

$$\textbf{data}\ M\ a = Unit\ a \mid M\ b\ \text{`}Bind\text{`}\ (b \rightarrow M\ a)$$

Notice that the type variable b does not occur on the left hand side of this definition! It is an existentially quantified type variable: one may construct an $M\ a$ by applying *Bind* at *any* type b[5]. With this representation *value* can be defined by

$$\begin{array}{l} value :: M\ a \rightarrow a \\ value\ (Unit\ x) = x \\ value\ (m\ \text{`}Bind\text{`}\ f) = value\ (f\ (value\ m)) \end{array}$$

[5] Such existential type definitions were proposed by Läufer[2] and are not part of standard Haskell, but are accepted by `hbc`.

which uses polymorphic recursion: the inner recursive call of *value* is at a different type from the enclosing one[6].

However, we can avoid these complications by using a representation based on simplified terms instead. In fact, we can simplify every term to the form *unit* x. Dropping the *Bind* constructor from the monad type, we obtain

$$\begin{aligned} &unit\ x = Unit\ x \\ &(Unit\ x)\ \text{`}bind\text{`}\ f = f\ x \\ &value\ (Unit\ x) = x \end{aligned}$$

where the only property of *unit* and *bind* we need to derive these definitions is the first monad law. And now, since *Unit* is the only constructor in the monad type we can drop it too, represent $M\ a$ just by a, and obtain the standard identity monad.

5.2 The Context-passing Representation of a Simple Monad

Suppose we instead derive a context-passing implementation. We are interested in contexts which make an observation by applying *value*, and using the monad laws we will be able to put every such context into the form $value\ ([\bullet]\ \text{`}bind\text{`}\ k)$, because

$$\begin{aligned} value\ [\bullet] &= value\ ([\bullet]\ \text{`}bind\text{`}\ unit) \\ value\ (([\bullet]\ \text{`}bind\text{`}\ f)\ \text{`}bind\text{`}\ k) &= value\ ([\bullet]\ \text{`}bind\text{`}\ \lambda x \to (f\ x\ \text{`}bind\text{`}\ k)) \end{aligned}$$

Notice here that if the hole is of type $M\ a$, the final value computed may be of some other type — call it *ans*. We must therefore represent contexts by a type parameterised on both a and *ans*. Consequently we are also obliged to represent monad values by a type parameterised on both a and *ans*. For example, we can define

$$\begin{aligned} &\textbf{data}\ Cxt\ a\ ans = ValueBind\ (a \to M\ ans\ ans) \\ &\textbf{type}\ M\ a\ ans = Cxt\ a\ ans \to ans \end{aligned}$$

where

$$ValueBind\ k = value\ ([\bullet]\ \text{`}bind\text{`}\ k)$$

However, it isn't hard to guess that uses of k will all take the form $value\ (k\ x)$ for some x. We therefore optimise the representation of contexts to

$$\textbf{data}\ Cxt\ a\ ans = ValueBind\ (a \to ans)$$

where

$$ValueBind\ (\lambda x.value\ (k\ x)) = value\ ([\bullet]\ \text{`}bind\text{`}\ k)$$

(If our guess proves to be wrong no harm will be done, we will simply be unable to derive definitions for the monad operations).

Now letting $\hat{k} = \lambda x.value\ (k\ x)$, we can derive

[6] Again this is not standard Haskell, but is accepted by hbc provided the type of *value* is explicitly given.

$$\begin{aligned}
\mathit{unit}\ x\ (\mathit{ValueBind}\ \hat{k}) &= \mathit{value}\ (\mathit{unit}\ x\ \text{`}\mathit{bind}\text{`}\ k) \\
&= \mathit{value}\ (k\ x) && \text{[1st monad law]} \\
&= \hat{k}\ x
\end{aligned}$$

$$\begin{aligned}
&(m\ \text{`}\mathit{bind}\text{`}\ f)\ (\mathit{ValueBind}\ \hat{k}) \\
&= \mathit{value}\ ((m\ \text{`}\mathit{bind}\text{`}\ f)\ \text{`}\mathit{bind}\text{`}\ k) \\
&= \mathit{value}\ (m\ \text{`}\mathit{bind}\text{`}\ \lambda x \rightarrow (f\ x\ \text{`}\mathit{bind}\text{`}\ k)) && \text{[3rd monad law]} \\
&= m\ (\mathit{ValueBind}\ (\lambda x \rightarrow \mathit{value}\ (f\ x\ \text{`}\mathit{bind}\text{`}\ k))) && \text{[prop. } \mathit{ValueBind}\text{]} \\
&= m\ (\mathit{ValueBind}\ (\lambda x \rightarrow f\ x\ (\mathit{ValueBind}\ \hat{k}))) && \text{[again]}
\end{aligned}$$

$$\begin{aligned}
\mathit{value}\ m &= \mathit{value}\ (m\ \text{`}\mathit{bind}\text{`}\ \mathit{unit}) && \text{[2nd monad law]} \\
&= m\ (\mathit{ValueBind}\ (\lambda x \rightarrow \mathit{value}\ (\mathit{unit}\ x))) && \text{[prop. } \mathit{ValueBind}\text{]} \\
&= m\ (\mathit{ValueBind}\ (\lambda x \rightarrow x)) && \text{[prop. } \mathit{value}\text{]}
\end{aligned}$$

And now dropping the superfluous constructor *ValueBind*, we obtain the definitions in figure 4 — the standard monad of continuations!

$$\begin{aligned}
&\textbf{type}\ M\ a\ \mathit{ans} = (\mathit{Cxt}\ a\ \mathit{ans}) \rightarrow \mathit{ans} \\
&\textbf{type}\ \mathit{Cxt}\ a\ \mathit{ans} = a \rightarrow \mathit{ans} \\
&\mathit{unit}\ x\ \hat{k} = \hat{k}\ x \\
&(m\ \text{`}\mathit{bind}\text{`}\ f)\ \hat{k} = m\ (\lambda x \rightarrow f\ x\ \hat{k}) \\
&\mathit{value}\ m = m\ (\lambda x \rightarrow x)
\end{aligned}$$

Fig. 4. The Optimised Context-passing Monad.

6 Monads for Backtracking

We've seen how we can derive both the identity monad and the monad of continuations from the 'vanilla' monad specification. But in reality we wish to add further operations to the signature — that is the *raison d'être* of monads. As an example, we'll consider operations for backtracking:

$$\begin{aligned}
\mathit{fail} &:: M\ a \\
\mathit{orelse} &:: M\ a \rightarrow M\ a \rightarrow M\ a
\end{aligned}$$

The new operations form a monoid,

$$\begin{aligned}
\mathit{fail}\ \text{`}\mathit{orelse}\text{`}\ x &= x \\
x\ \text{`}\mathit{orelse}\text{`}\ \mathit{fail} &= x \\
(x\ \text{`}\mathit{orelse}\text{`}\ y)\ \text{`}\mathit{orelse}\text{`}\ z &= x\ \text{`}\mathit{orelse}\text{`}\ (y\ \text{`}\mathit{orelse}\text{`}\ z)
\end{aligned}$$

and we must also specify their interaction with the monad operations[7]:

$$\begin{aligned} \mathit{fail}\ \text{`}\mathit{bind}\text{`}\ f &= \mathit{fail} \\ (x\ \text{`}\mathit{orelse}\text{`}\ y)\ \text{`}\mathit{bind}\text{`}\ f &= (x\ \text{`}\mathit{bind}\text{`}\ f)\ \text{`}\mathit{orelse}\text{`}(y\ \text{`}\mathit{bind}\text{`}\ f) \end{aligned}$$

Finally, it is no longer appropriate to give *value* the type

$$\mathit{value} :: M\ a \rightarrow a$$

because there is no sensible behaviour for *value fail*. Instead, we give it the type

$$\mathit{value} :: M\ a \rightarrow \mathit{Maybe}\ a$$

where

$$\textbf{data}\ \mathit{Maybe}\ a = \mathit{Yes}\ a \mid \mathit{No}$$

satisfying the laws

$$\begin{aligned} \mathit{value}\ \mathit{fail} &= \mathit{No} \\ \mathit{value}\ (\mathit{unit}\ x\ \text{`}\mathit{orelse}\text{`}\ m) &= \mathit{Yes}\ x \end{aligned}$$

So we can observe whether a backtracking computation succeeds or fails, and if it succeeds we observe the first answer. Let us apply the same methods to derive implementations of this monad.

6.1 The Term Representation of the Backtracking Monad

Rather than start from scratch to develop a term representation for backtracking, observe that if we replace M by *Seq*, *fail* by *nil*, and *orelse* by *cat*, then these operations together with *unit* satisfy exactly the same axioms as in section 4. That suggests that we try to use the same kind of simplified terms as in section 4.1, namely *fail* and *unit x ‘orelse‘ m*. So let us define

$$\textbf{data}\ M\ a = \mathit{Fail} \mid a\ \text{`}\mathit{UnitOrElse}\text{`}\ M\ a$$

reuse the previously derived definitions for *unit*, *fail* and *orelse*, and see if we can derive implementations of the remaining operators.

In the case of *bind*, we derive

$$\begin{aligned} \mathit{Fail}\ \text{`}\mathit{bind}\text{`}\ f &= \mathit{fail}\ \text{`}\mathit{bind}\text{`}\ f \\ &= \mathit{fail} \\ &= \mathit{Fail} \end{aligned}$$

$$\begin{aligned} (x\ \text{`}\mathit{UnitOrElse}\text{`}\ m)\ \text{`}\mathit{bind}\text{`}\ f &= (\mathit{unit}\ x\ \text{`}\mathit{orelse}\text{`}\ m)\ \text{`}\mathit{bind}\text{`}\ f \\ &= (\mathit{unit}\ x\ \text{`}\mathit{bind}\text{`}\ f)\ \text{`}\mathit{orelse}\text{`}(m\ \text{`}\mathit{bind}\text{`}\ f) \\ &= f\ x\ \text{`}\mathit{orelse}\text{`}(m\ \text{`}\mathit{bind}\text{`}\ f) \end{aligned}$$

(which is a terminating definition because the recursive call of *bind* has a smaller first argument), and in the case of *value*, we find directly that

$$\begin{aligned} &\mathit{value}\ \mathit{Fail} = \mathit{No} \\ &\mathit{value}\ (x\ \text{`}\mathit{UnitOrElse}\text{`}\ m) = \mathit{Yes}\ x \end{aligned}$$

So as we expected, we can implement the backtracking monad using lists.

[7] It is the second equation here which distinguishes backtracking from exception handling.

6.2 Context-passing Implementation of Backtracking

When we develop a context-passing implementation of backtracking we have to consider more complex forms of context than in section 5.2, since of course the new operations *fail* and *orelse* may occur in the context too. But just as we used the monad laws then to express all contexts with a single bind, so here we can use the monoidal properties of *fail* and *orelse* to express all contexts with a single *orelse*. Furthermore, we need not consider contexts with *orelse* nested inside *bind*, because

$$([\bullet]\ \text{`}orelse\text{`}b)\ \text{`}bind\text{`}k = ([\bullet]\ \text{`}bind\text{`}k)\ \text{`}orelse\text{`}(b\ \text{`}bind\text{`}k)$$

It is therefore sufficient to consider contexts of the form

$$value\ (([\bullet]\ \text{`}bind\text{`}k)\ \text{`}orelse\text{`}b)$$

(Remember that this choice isn't critical. If we make a mistake at this point, we will discover it when we are unable to complete the derivations of the operators.)

Moreover, we may reasonably guess (or discover by doing the derivations) that uses of k will be in the context $value\ (k\ x\ \text{`}orelse\text{`}b)$ for some x and b, and uses of b will be in the context $value\ b$. We will therefore represent contexts by the type

$$\mathbf{data}\ Cxt\ a\ ans = VBO\ (a \to Maybe\ ans \to Maybe\ ans)\ (Maybe\ ans)$$

where

$$\begin{aligned}&(\forall x, b'.\hat{k}\ x\ (value\ b') = value\ (k\ x\ \text{`}orelse\text{`}b'))\\ &\Rightarrow VBO\ \hat{k}\ (value\ b) = value\ (([\bullet]\ \text{`}bind\text{`}k)\ \text{`}orelse\text{`}b)\end{aligned}$$

The antecedent says that uses of k of the form we expect can be represented by applying $\hat{k}$. Since we plan to store only the *value* of b and b' it is natural to require that $\hat{k}$ need only the value. The conclusion says that in that case, the contexts we are interested in can be represented using *VBO*.

Now assuming that $\hat{k}$ has the property in the antecedent and that $\hat{b} = value\ b$, we can derive

$$\begin{aligned}unit\ x\ (VBO\ \hat{k}\ \hat{b}) &= value\ ((unit\ x\ \text{`}bind\text{`}k)\ \text{`}orelse\text{`}b)\\ &= value\ (k\ x\ \text{`}orelse\text{`}b) && \text{[1st monad law]}\\ &= \hat{k}\ x\ (value\ b) && \text{[prop. } \hat{k}\text{]}\\ &= \hat{k}\ x\ \hat{b} && \text{[prop. } \hat{b}\text{]}\end{aligned}$$

$$\begin{aligned}fail\ (VBO\ \hat{k}\ \hat{b}) &= value\ ((fail\ \text{`}bind\text{`}k)\ \text{`}orelse\text{`}b)\\ &= value\ (fail\ \text{`}orelse\text{`}b)\\ &= value\ b\\ &= \hat{b}\end{aligned}$$

The derivation of *bind* is a little more complicated because of the more complex property that $\hat{k}$ satisfies. We begin in the usual way,

$$\begin{aligned}&(m\ \text{`}bind\text{`}f)\ (VBO\ \hat{k}\ \hat{b})\\ &= value\ (((m\ \text{`}bind\text{`}f)\ \text{`}bind\text{`}k)\ \text{`}orelse\text{`}b)\\ &= value\ ((m\ \text{`}bind\text{`}(\lambda x \to f\ x\ \text{`}bind\text{`}k))\ \text{`}orelse\text{`}b) && \text{[3rd monad law]}\\ &= m\ (VBO\ \hat{k}'\ \hat{b})\end{aligned}$$

provided $\hat{k}'$ satisfies

$$\hat{k}'\ x\ (\mathit{value}\ b') = \mathit{value}\ ((f\ x\ \mathit{‘bind‘}\, k)\ \mathit{‘orelse‘}\, b')$$

But the right hand side of this equation is equal to

$$f\ x\ (\mathit{VBO}\ \hat{k}\ (\mathit{value}\ b'))$$

and so we can satisfy the condition by taking

$$\hat{k}'\ x\ \hat{b}' = f\ x\ (\mathit{VBO}\ \hat{k}\ \hat{b}')$$

So completing the derivation,

$$(m\ \mathit{‘bind‘}\, f)\ (\mathit{VBO}\ \hat{k}\ \hat{b}) = m\ (\mathit{VBO}\ (\lambda x\ \hat{b}' \to f\ x\ (\mathit{VBO}\ \hat{k}\ \hat{b}'))\ \hat{b})$$

The derivation of *orelse* is straightforward:

$$\begin{array}{ll}
(m\ \mathit{‘orelse‘}\, n)\ (\mathit{VBO}\ \hat{k}\ \hat{b}) & \\
\ = \mathit{value}\ (((m\ \mathit{‘orelse‘}\, n)\ \mathit{‘bind‘}\, k)\ \mathit{‘orelse‘}\, b) & \\
\ = \mathit{value}\ (((m\ \mathit{‘bind‘}\, k)\ \mathit{‘orelse‘}\,(n\ \mathit{‘bind‘}\, k))\ \mathit{‘orelse‘}\, b) & \\
\ = \mathit{value}\ ((m\ \mathit{‘bind‘}\, k)\ \mathit{‘orelse‘}\,((n\ \mathit{‘bind‘}\, k)\ \mathit{‘orelse‘}\, b)) & \text{[associativity]} \\
\ = m\ (\mathit{VBO}\ \hat{k}\ (\mathit{value}\ ((n\ \mathit{‘bind‘}\, k)\ \mathit{‘orelse‘}\, b))) & \text{[prop. } \mathit{VBO}\text{]} \\
\ = m\ (\mathit{VBO}\ \hat{k}\ (n\ (\mathit{VBO}\ \hat{k}\ (\mathit{value}\ b)))) & \text{[prop. } \mathit{VBO}\text{]} \\
\ = m\ (\mathit{VBO}\ \hat{k}\ (n\ (\mathit{VBO}\ \hat{k}\ \hat{b})))) &
\end{array}$$

Finally, we derive *value*:

$$\begin{aligned}
\mathit{value}\ m &= \mathit{value}\ (m\ \mathit{‘orelse‘}\, \mathit{fail}) \\
&= \mathit{value}\ ((m\ \mathit{‘bind‘}\, \mathit{unit}\ )\ \mathit{‘orelse‘}\, \mathit{fail}) \\
&= m\ (\mathit{VBO}\ \hat{k}'\ (\mathit{value}\ \mathit{fail})) \\
&= m\ (\mathit{VBO}\ \hat{k}'\ \mathit{No})
\end{aligned}$$

provided

$$\hat{k}'\ x\ (\mathit{value}\ b') = \mathit{value}\ (\mathit{unit}\ x\ \mathit{‘orelse‘}\, b')$$

But the right hand side here is equal to $\mathit{Yes}\ x$, so we take $k' = \lambda x\ \hat{b}' \to \mathit{Yes}\ x$ to complete the derivation.

We can simplify the definitions slightly further by dropping the *VBO* constructor and replacing every context argument by two arguments, $\hat{k}$ and $\hat{b}$. Putting the results together, we obtain the definitions in figure 5, a continuation passing implementation of backtracking.

Exercise 2. Consider the *state monad*, with additional operations

$$\begin{aligned}
&\mathit{fetch} :: M\ \mathit{St} \\
&\mathit{store} :: \mathit{St} \to M\ () \\
&\mathit{run} :: M\ a \to \mathit{St} \to a
\end{aligned}$$

$$\begin{array}{l}
\mathbf{type}\ M\ a\ ans = (a \rightarrow Maybe\ ans \rightarrow Maybe\ ans) \rightarrow Maybe\ ans \rightarrow Maybe\ ans \\
unit\ x\ \hat{k}\ \hat{b} = \hat{k}\ x\ \hat{b} \\
(m\ \text{`}bind\text{`} f)\ \hat{k}\ \hat{b} = m\ (\lambda x\ \hat{b}' \rightarrow f\ x\ \hat{k}\ \hat{b}')\ \hat{b} \\
fail\ \hat{k}\ \hat{b} = \hat{b} \\
(m\ \text{`}orelse\text{`} n)\ \hat{k}\ \hat{b} = m\ \hat{k}\ (n\ \hat{k}\ \hat{b}) \\
value\ m = m\ (\lambda x\ \hat{b}' \rightarrow Yes\ x)\ No
\end{array}$$

Fig. 5. A Context-passing Implementation of Backtracking.

satisfying

$$\begin{aligned}
fetch\ \text{`}bind\text{`}\,\lambda s \rightarrow store\ s &= unit\ () \\
store\ s\ \text{`}bind\text{`}\,\lambda() \rightarrow fetch &= store\ s\ \text{`}bind\text{`}\,\lambda() \rightarrow unit\ s \\
store\ s\ \text{`}bind\text{`}\,\lambda() \rightarrow store\ s' &= store\ s' \\
run\ (unit\ x)\ s &= x \\
run\ (fetch\ \text{`}bind\text{`} f)\ s &= run\ (f\ s)\ s \\
run\ (store\ s\ \text{`}bind\text{`} f)\ s' &= run\ (f\ ())\ s
\end{aligned}$$

Derive term and context passing implementations of these operations.

7 Specifying Pretty-printing

Now we shall return to our case study: pretty-printing. Before we can start to derive implementations of the pretty-printing combinators we must develop a specification. But in this case, it isn't intuitively obvious what laws the pretty-printing combinators should satisfy! We need some way to guide our intuition, to lead us to write down the *right* laws for the combinators.

In mathematics, we often guide our intuition with the help of an example. If we are formulating hypothesis about certain topological spaces, we might think about the reals. It is even more important when formulating a new concept, such as a group, to have a concrete model in mind. We are trying to formulate a theory of pretty-printing, but as yet we have no model to guide us. So we shall start off by looking for an abstract model of documents, on which we can agree what the behaviour of the combinators should be. Our model will not be — and is not intended to be — a reasonable implementation, but it can be thought of as a kind of 'denotational semantics' for the combinators. Using the model we can establish algebraic properties which the combinators should satisfy — in *any* implementation. And then once these properties are established, we can use them as in the previous sections to derive implementations.

7.1 Abstract Layouts

We'll begin by looking for an abstract model of a pretty-printer's output — that is, prettily indented text. We could say that the output is just a string, but a string has so little structure that we can derive no intuition from it. Let us say instead, that a *layout* is a sequence of indented lines, which we can model as

$$\textbf{type}\ \mathit{Layout} = [(\mathit{Int}, \mathit{String})]^{+}$$

Notice that we shall allow indentations to be negative: later on this will contribute to a nicer algebra, just as integers have a nicer algebra than natural numbers. But notice also that we restrict layouts to be *non-empty* (we use $[-]^{+}$ for the type of non-empty lists). We'll return to this point below.

We can now specify *text*, *nest* and ($$) very easily:

$$\begin{array}{l}
\mathit{text}\ s = [(0, s)] \\
\mathit{nest}\ k\ l = [(i + k, s) | (i, s) \leftarrow l] \\
l_1\ \$\$\ l_2 = l_1 +\!\!+ l_2
\end{array}$$

The right definition of horizontal composition ($<>$) is not so obvious. The desired behaviour is clear enough when *text s* is placed beside *text t*, but what if both layouts are indented? What if the arguments occupy more than one line each?

Our choice is guided by the following principles:

- The two dimensional structure of each argument should be preserved; that is, the appearance of $x <> y$ on the page should consist of some combination of a translation of x and a translation of y.
- Our intention is that a layout is just a pretty way of displaying a string. What string? We define

  $$\begin{array}{l}
  \mathit{string} :: \mathit{Layout} \rightarrow \mathit{String} \\
  \mathit{string}\ l = \mathit{foldr1}\ (\oplus)\ (\mathit{map}\ \mathit{snd}\ l) \\
  \quad \textbf{where}\ \ s \oplus t = s +\!\!+ \text{“ ”} +\!\!+ t
  \end{array}$$

  (We interpret a line break as white space — equivalent to a single space). Then we expect that $\mathit{string}\ (x <> y) = \mathit{string}\ x +\!\!+ \mathit{string}\ y$. This property enables the programmer to predict the string that $x <> y$ represents, without thinking about how x and y are laid out.
- Indentation cannot appear in the middle of a line — since our abstract model (fortunately) cannot represent this.

There is really only one choice for ($<>$) that meets these three criteria: to translate the second operand so that its first character abuts against the last character of the first operand. Formally,

$$(x +\!\!+ [(i, s)]) <> ([(j, t)] +\!\!+ y) = x +\!\!+ [(i, s +\!\!+ t)] +\!\!+ \mathit{nest}\ (i + \mathit{length}\ s - j)\ y$$

To see that the definition is reasonable, consider the following two examples:

$$\boxed{\textsf{one:}} <> \boxed{\begin{array}{l}\textsf{two}\\ \textsf{three}\end{array}} = \boxed{\begin{array}{l}\textsf{one: two}\\ \textsf{\quad\ \ \ three}\end{array}}$$

$$\boxed{\begin{array}{l}\textsf{while x>0 do}\\ \textsf{\ \ x := x-2}\end{array}} <> \boxed{\textsf{;}} = \boxed{\begin{array}{l}\textsf{while x>0 do}\\ \textsf{\ \ x := x-2;}\end{array}}$$

So at least in cases where one of the operands is a single line, the result is reasonable and useful.

Now look again at the formal definition of (<>). It is only defined for non-empty arguments! This is the reason for the restriction to non-empty layouts that we made above: there is simply no sensible definition of (<>) for empty arguments. The restriction is unfortunate: the empty layout would be a unit for $\$\$$, so improving the combinator algebra, and moreover would be useful in practice. But if we allow empty layouts and simply make some arbitrary choice for the value of <> in these cases, many algebraic laws involving <> cease to hold. A way out of the dilemma would be to allow empty layouts, and define <> to be a partial operator. But since this would complicate the development we have not done so.

7.2 The Algebra of Layouts

Now that we have formal definitions of the layout operators we can study their algebra. The laws in figure 6 are easily proved, although the proofs are not included here.

$$
\begin{aligned}
(x <> y) <> z &= x <> (y <> z)\\
(x \mathbin{\$\$} y) \mathbin{\$\$} z &= x \mathbin{\$\$} (y \mathbin{\$\$} z)\\
x <> \mathit{text}\ \text{""} &= x\\
\mathit{nest}\ k\ (x \mathbin{\$\$} y) &= \mathit{nest}\ k\ x \mathbin{\$\$} \mathit{nest}\ k\ y\\
\mathit{nest}\ k\ (x <> y) &= \mathit{nest}\ k\ x <> y\\
x <> \mathit{nest}\ k\ y &= x <> y\\
\mathit{nest}\ k\ (\mathit{nest}\ k'\ x) &= \mathit{nest}\ (k + k')\ x\\
\mathit{nest}\ 0\ x &= x\\
(x \mathbin{\$\$} y) <> z &= x \mathbin{\$\$} (y <> z)\\
\mathit{text}\ s <> ((\mathit{text}\ \text{""} <> y) \mathbin{\$\$} z) &= (\mathit{text}\ s <> y) \mathbin{\$\$} \mathit{nest}\ (\mathit{length}\ s)\ z\\
\mathit{text}\ s <> \mathit{text}\ t &= \mathit{text}\ (s +\!\!+ t)
\end{aligned}
$$

Fig. 6. Algebraic laws for layout operations.

First, both <> and $\$\$$ are associative, and <> has *text* "" as a right unit. However, <> has no left unit because the indentation of the second operand is always lost. For example,

$$\mathit{text}\ \text{""} <> \boxed{\quad\ \mathsf{foo}} = \boxed{\mathsf{foo}}$$

Since we excluded empty layouts, $\$\$$ has no units at all.

The indentation combinator *nest* distributes over $\$\$$, and distributes over <> on the left. We do not need to indent the right operand of <> here, because it is translated to abut against the left operand and so its indentation is lost. For the same reason *nest* can be cancelled to the right of <>. Of course consecutive *nest*s can be combined, and *nest*ing by zero is the identity operation.

Moreover $\$\$$ and $<>$ are related to one another by a kind of associative law: we may say they 'associate with' one another. For example,

$$(\boxed{\mathsf{a}} \, \$\$ \, \boxed{\mathsf{b}}) <> \boxed{\begin{array}{l}\mathsf{c}\\ \mathsf{d}\end{array}} = \boxed{\begin{array}{l}\mathsf{a}\\ \mathsf{bc}\\ \phantom{\mathsf{b}}\mathsf{d}\end{array}} = \boxed{\mathsf{a}} \, \$\$ \, (\boxed{\mathsf{b}} <> \boxed{\begin{array}{l}\mathsf{c}\\ \mathsf{d}\end{array}})$$

On the other hand,

$$(x <> y) \ \$\$ \ z \neq x <> (y \ \$\$ \ z)$$

Here the indentation of z is different in the two cases: for example,

$$(\boxed{\mathsf{a}} <> \boxed{\mathsf{b}}) \ \$\$ \boxed{\mathsf{c}} = \boxed{\begin{array}{l}\mathsf{ab}\\ \mathsf{c}\end{array}} \neq \boxed{\begin{array}{l}\mathsf{ab}\\ \phantom{\mathsf{a}}\mathsf{c}\end{array}} = \boxed{\mathsf{a}} <> (\boxed{\mathsf{b}} \, \$\$ \boxed{\mathsf{c}})$$

It is the failure of this law to hold that makes the pretty-printing algebra interesting! We have to have some way to transform expressions of the form $x <> (y \ \$\$ \ z)$, and we can in the special case when we know the position where x ends, and the indentation of the first line of y. For example, when x is just a *text*, and y is of the form *text* "" $<> y'$. The following law is sufficient:

$$\mathit{text}\ s <> ((\mathit{text}\ \text{""} <> y) \ \$\$ \ z) = (\mathit{text}\ s <> y) \ \$\$ \ \mathit{nest}\ (\mathit{length}\ s)\ z$$

One might say that the difficult part of pretty-printing is transforming expressions so that this law is applicable.

Finally there is a simple law relating $<>$ and *text*.

In a sense these laws completely specify the bahaviour of the layout operators: any two closed terms which denote the same layout can be proved equal using these laws.

Exercise 3. Prove this remark, by choosing a canonical form for layout expressions such that every layout is denoted by a unique canonical form, and by deriving implementations of the operators that map canonical forms to canonical forms.

Remark on the benefits of a formal approach: This formal specification of the layout operators is an after-the-fact reconstruction. The first implementation was constructed using seat-of-the-pants intuition, and the combinators' behaviour was very subtly different. The *nest* combinator inserted spaces 'in a vertical context': that is, when used as an operand of $\$\$$ or at the top level.

As a consequence the law

$$\mathit{nest}\ k\ x <> y = x <> y$$

held in the implementation — the context here is 'horizontal'. But since the behaviour of a layout depended on its context, we could not give a simple abstract model such as that in the previous section. Moreover, of the eleven laws in figure 6, four did not hold (which four?) Both the user and the developer of the library were deprived of a useful algebra.

For the user (that is the author of a pretty-printer) each law means one less worry: there is no need to think about whether to write the left or the right hand side. For the developer, each law simplifies optimisation: the original library was very hard to optimise without changing its behaviour. The program we are following now, of deriving implementations from the algebra, would have been extremely difficult to follow.

And all these problems stemmed from a very subtle error that was only revealed by writing a formal specification...

7.3 Abstract Documents

The layout operations enable us to construct individual layouts, but a pretty-printer must of course choose between many alternative layouts. We make a design decision: to separate the construction of alternatives from the choice of the prettiest layout. We represent a collection of alternatives by a set:

$$\textbf{type}\ \mathit{Doc} = \mathcal{P}(\mathit{Layout})$$

We will require that every layout in a *Doc* represent the same string, so that the programmer knows which string is being pretty-printed.

The choice of a particular layout will be made by a function

$$\mathit{best} :: \mathit{Doc} \rightarrow \mathit{Layout}$$

Thus the author of a pretty-printer need only construct a set of alternatives; the hard work of selecting the best alternative is done just by reusing the function *best*.

Since *Doc*s are just sets of layouts, there is a natural way to promote the layout operations to work on *Doc*s too. We just apply the operation to the elements of the operand sets and form a set of the results — for example,

$$d_1 <> d_2 = \{l_1 <> l_2 | l_i \in d_i\}$$

The promoted operations distribute over ∪ and preserve ∅ — for example,

$$\begin{aligned}(x \cup y) <> z &= (x <> z) \cup (y <> z)\\ \emptyset <> z &= \emptyset\end{aligned}$$

Moreover, since the laws of the layout algebra are all *linear* in the sense that no variable appears more than once on either the left or right hand side, then they hold for documents also. So all the laws in figure 6 remain true for *Doc*s.

Of course, if we confine ourselves to the layout operations we can only construct *Doc*s with a single element. We must add an operation with multiple possible layouts. Since we require that all layouts in a document represent the same string, union is not an appropriate operator to provide — rather we should define an operation that forms the union of two *Doc*s that are guaranteed to represent the same string. Noting that

$$\mathit{string}\ (x\ \$\$\ y) = \mathit{string}\ x ++ \text{`` ''} ++ \mathit{string}\ y$$

it is tempting to define

$$x <+> y = x <> \mathit{text}\ \text{`` ''} <> y$$

and define an operator that forms the union of $x \$\$ y$ and $x <+> y$.

However, this isn't quite enough. Sometimes we want to make several choices consistently: for example, we may want to allow

$$\boxed{\textsf{if e}} <+> \boxed{\textsf{then s1}} <+> \boxed{\textsf{else s2}} = \boxed{\textsf{if e then s1 else s2}}$$

$$\boxed{\textsf{if e}} \$\$ \boxed{\textsf{then s1}} \$\$ \boxed{\textsf{else s2}} = \boxed{\begin{array}{l}\textsf{if e}\\ \textsf{then s1}\\ \textsf{else s2}\end{array}}$$

as alternatives, without also allowing

$$\boxed{\textsf{if e}} <+> (\boxed{\textsf{then s1}} \$\$ \boxed{\textsf{else s2}}) = \boxed{\begin{array}{l}\textsf{if e then s1}\\ \textsf{else s2}\end{array}}$$

$$\boxed{\textsf{if e}} \$\$ \boxed{\textsf{then s1}} <+> \boxed{\textsf{else s2}} = \boxed{\begin{array}{l}\textsf{if e}\\ \textsf{then s1 else s2}\end{array}}$$

We therefore define an n-ary operation, which makes $n-1$ choices consistently:

$$\begin{array}{l} \mathit{sep} :: [\mathit{Doc}]^{+} \rightarrow \mathit{Doc} \\ \mathit{sep}\ \mathit{xs} = \mathit{foldr1}\ (<+>)\ \mathit{xs} \cup \mathit{foldr1}\ (\$\$)\ \mathit{xs} \end{array}$$

We'll revise this definition slightly below, but first let us observe a pleasing interaction between *sep* and *nest*. Consider for example,

$$\mathit{sep}\ [\boxed{\textsf{while x}\geq\textsf{0 do}}, \mathit{nest}\ 2\ \boxed{\textsf{x := x-2}}]$$

The alternative layouts here are

$$\begin{array}{rl} \boxed{\textsf{while x}\geq\textsf{0 do}} <+> \mathit{nest}\ 2\ \boxed{\textsf{x := x-2}} & = \boxed{\textsf{while x}\geq\textsf{0 do}} <+> \boxed{\textsf{x := x-2}} \\ & = \boxed{\textsf{while x}\geq\textsf{0 do x := x-2}} \end{array}$$

and

$$\boxed{\textsf{while x}\geq\textsf{0 do}} \$\$ \mathit{nest}\ 2\ \boxed{\textsf{x := x-2}} = \boxed{\begin{array}{l}\textsf{while x}\geq\textsf{0 do}\\ \textsf{x := x-2}\end{array}}$$

In the horizontal form no unwanted extra indentation appears, because *nest* can be cancelled on the right of $(<>)$.

Now let us consider an example which motivates a slight refinement to the definition of *sep*. Suppose we wish to pretty-print pairs of statements separated by a semicolon, choosing between horizontal and vertical layouts. We might define

$$\mathit{semic}\ x\ y = \mathit{sep}\ [x <> \mathit{text}\ ``;", y]$$

Now for example,

$$\mathit{semic}\ \boxed{\textsf{x:=0}}\ \boxed{\textsf{y:=0}} = \left\{\boxed{\textsf{x:=0; y:=0}}, \boxed{\begin{array}{l}\textsf{x:=0;}\\ \textsf{y:=0}\end{array}}\right\}$$

in which both horizontal and vertical layouts look fine. But consider

$$\begin{aligned} & \mathit{semic}\ \boxed{\begin{array}{l}\text{while x}\geq\text{2 do}\\ \quad\text{x:=x-2}\end{array}}\ \boxed{\begin{array}{l}\text{while y}\geq\text{2 do}\\ \quad\text{y:=y-2}\end{array}} \\ = & \left\{ \boxed{\begin{array}{l}\text{while x}\geq\text{2 do}\\ \quad\text{x:=x-2; while y}\geq\text{2 do}\\ \qquad\qquad\quad\text{y:=y-2}\end{array}}, \boxed{\begin{array}{l}\text{while x}\geq\text{2 do}\\ \quad\text{x:=x-2;}\\ \text{while y}\geq\text{2 do}\\ \quad\text{y:=y-2}\end{array}} \right\} \end{aligned}$$

In cases such as this, the horizontal layout is ugly or even misleading. We therefore redefine

$$\begin{aligned} & \mathit{sep}\ \mathit{xs} = \mathit{fit}\ (\mathit{foldr1}\ (\mathord{<}\!+\!\mathord{>})\ \mathit{xs}) \cup \mathit{foldr1}\ (\$\$)\ \mathit{xs} \\ & \quad \textbf{where}\ \ \mathit{fit}\ d = \{l \in d | \mathit{length}\ l = 1\} \end{aligned}$$

which restricts the horizontal form of a *sep* to fit on one line.

The algebraic properties of *fit* are very simply stated — see figure 7. The *sep* operator has fewer useful properties, and we will develop them as we need them.

$$\begin{aligned} \mathit{fit}\ (\mathit{text}\ s) &= \mathit{text}\ s \\ \mathit{fit}\ (\mathit{nest}\ k\ x) &= \mathit{nest}\ k\ (\mathit{fit}\ x) \\ \mathit{fit}\ (x <> y) &= \mathit{fit}\ x <> \mathit{fit}\ y \\ \mathit{fit}\ (x\ \$\$\ y) &= \emptyset \\ \mathit{fit}\ (x \cup y) &= \mathit{fit}\ x \cup \mathit{fit}\ y \\ \mathit{fit}\ \emptyset &= \emptyset \end{aligned}$$

Fig. 7. The *fit* laws.

Exercise 4. Define a type of abstract syntax trees for a simple imperative language, with assignment statements, **if–then–else**, **while–do**, and **begin–end**. Use the combinators to write a pretty-printer for this type.

7.4 Choosing a Pretty Layout

Now that we have designed combinators for constructing documents with many possible layouts, it is time to discuss choosing among those alternatives. Many pretty-printers aim simply to avoid exceeding a given page width. However, we found that using this criterion alone tends to produce layouts such as

$$\boxed{\text{for i = 1 to 100 do for j = 1 to 100 do for k = 1 to 100 do a[i,j,k] := 0}}$$

which fits on a page, but cannot be described as pretty. We therefore impose an additional constraint limiting the number of characters on each line (excluding indentation) to a smaller number. The idea is to avoid placing too much information on a line — even a line that begins at the left margin. Under this constraint the example above might instead be laid out as

```
for i = 1 to 100 do
   for j = 1 to 100 do
      for k = 1 to 100 do a[i,j,k] := 0
```

In general a pretty layout will consist of a ribbon of text snaking across the page. To see that this is reasonable, ask yourself: 'is the prettiest layout on an infinitely wide page really to place everything on one line?'

We will say that a line that meets both constraints is *nice*, and define

$$nice_r^w\ (i, s) \iff i + length\ s \leq w \land length\ s \leq r$$

where w is the page width and r is the width of the ribbon.

We might be tempted to specify that the pretty-printer choose a layout all of whose lines are nice, but we must be careful: some documents have no such layout at all. For example, *text* "13 characters" cannot be made to fit within a pagewidth of 12. Even in such cases we want the pretty-printer to produce something, and rather than adopt an ad hoc solution we accept that the niceness criteria will not always be met.

Moreover, even if a nice layout exists we may not want the pretty-printer to choose it! Consider for example the document

$$sep\ [\boxed{\text{a}}, \boxed{\text{b}}] <> \boxed{\begin{array}{l}\text{1}\\ \vdots\\ \text{999}\\ \text{13 characters}\end{array}}$$

$$= \left\{ \boxed{\begin{array}{l}\text{a b1}\\ \quad\vdots\\ \quad\text{999}\\ \quad\text{13 characters}\end{array}}, \boxed{\begin{array}{l}\text{a}\\ \text{b1}\\ \ \vdots\\ \ \text{999}\\ \ \text{13 characters}\end{array}} \right\}$$

This document has a nice layout on a page of width 14 characters — the second one. But it would be unreasonably inefficient for a pretty-printer to decide whether or not to split the first line of a document on the basis of the contents of the last. An efficient pretty-printer should need only a limited look-ahead, and so we must expect the first layout to be chosen despite the trouble ahead[8]. The question of which layout a pretty-printer chooses is thus trickier than it at first appears. Of course, it could never have been sufficient to say simply that a nice layout is chosen, since even if all layouts are nice, some will be preferable to others. We must instead define an *ordering* on layouts and choose the best.

We begin by defining an ordering $\rhd_r^w$ on individual lines. Our guiding principles are

[8] A different design decision is possible: we might choose to 'play safe' and split the first line unless a limited look-ahead shows that it is definitely unnecessary. We have not explored this alternative.

- a nice line is always better than an overflowing line,
- if one cannot avoid overflowing, it is better to overflow by a little than by a lot,
- unnecessary line breaks should be avoided.

We therefore define

$$x \rhd_r^w y \iff \begin{array}{l} (\mathit{nice}_r^w\ x \wedge \neg \mathit{nice}_r^w\ y) \\ \vee\ (\neg \mathit{nice}_r^w\ x \wedge \neg \mathit{nice}_r^w\ y \wedge \sharp x < \sharp y) \\ \vee\ (\mathit{nice}_r^w\ x \wedge \mathit{nice}_r^w\ y \wedge \sharp x > \sharp y) \end{array}$$

where the length of a line is given by $\sharp(i, s) = i + \mathit{length}\ s$.

If we know that $\sharp x > \sharp y$ then we can test $\rhd_r^w$ particularly simply:

$$\mathit{nice}_r^w\ x \Longrightarrow x \rhd_r^w y$$
$$\neg \mathit{nice}_r^w\ x \Longrightarrow y \rhd_r^w x$$

In the first case y must also be nice, but not as nice because it is shorter. In the second case either y is nice (and therefore nicer than x), or it is not nice, but nicer than x because it is shorter. We will use this property in the implementations.

Unfortunately an ordering on lines does not extend in a unique way to an ordering on layouts, and so we must make an arbitrary decision. We choose to order layouts by the lexicographic extension of the ordering on lines, which we will also write as $\rhd_r^w$. The reason for this choice is simple: lexicographic ordering can be decided left-to-right, and we hope to pretty-print documents from left to right without much look-ahead. We define

$$\nabla_r^w :: \mathit{Layout} \to \mathit{Layout} \to \mathit{Layout}$$

to select the lexicographically nicer of its arguments, and

$$\mathit{best} :: \mathit{Int} \to \mathit{Int} \to \mathit{Doc} \to \mathit{Layout}$$

such that *best w r* selects the lexicographically nicest layout in the set. It's also convenient to introduce a unit ∞ for ∇_r^w, representing a layout uglier than any other.

The careful reader will have noticed that $\rhd_r^w$ is only a *partial* order — if x and y are both lines of equal length, then neither $x \rhd_r^w y$ nor $y \rhd_r^w x$ holds, even though x and y need not be equal. Consequently both ∇_r^w and *best* are partial operations. But this will not trouble us, because all the document operations construct sets which are totally ordered by $\rhd_r^w$. This will become evident when we derive implementations of the library. Consider this: our task is to define when one layout is nicer than another layout *of the same document*; we have no need to (and indeed, we cannot) define when a layout is nicer than a layout of an unrelated document.

Let us now investigate the properties of $\rhd_r^w$, ∇_r^w and *best*. Since the ordering on layouts is lexicographic,

$$x \rhd_r^w y \Longrightarrow \left(\begin{array}{ll} & z \$\$ x \rhd_r^w z \$\$ y \\ \wedge & x \rhd_r^w y \$\$ z \\ \wedge & x \$\$ z \rhd_r^w y \end{array} \right)$$

Moreover,

$$x \rhd_r^w y \Longrightarrow \mathit{nest}\ k\ x \rhd_r^{w+k} \mathit{nest}\ k\ y$$

and therefore

$$\mathit{nest}\ k\ (x \nabla_r^w y) = \mathit{nest}\ k\ x \nabla_r^{w+k} \mathit{nest}\ k\ y$$

Finally, we can reformulate the observation about a simple test for $\rhd_r^w$ as follows:

$$\mathit{length}\ s > \mathit{length}\ t \Longrightarrow \left(\wedge \begin{array}{l} \mathit{length}\ s \leq w\ \text{`}\mathit{min}\text{`}\ r \Rightarrow \mathit{text}\ s \rhd_r^w \mathit{text}\ t \\ \mathit{length}\ s > w\ \text{`}\mathit{min}\text{`}\ r \Rightarrow \mathit{text}\ t \rhd_r^w \mathit{text}\ s \end{array} \right)$$

From these properties, and from the fact that *best* chooses the nicest element from a set, we can derive the laws in figure 8 for *best*.

$$\begin{aligned} \mathit{best}\ w\ r\ (x\ \$\$\ y) &= \mathit{best}\ w\ r\ x\ \$\$\ \mathit{best}\ w\ r\ y \\ \mathit{best}\ w\ r\ (\mathit{nest}\ k\ x) &= \mathit{nest}\ k\ (\mathit{best}\ _r^{w-k}\ x) \\ \mathit{best}\ w\ r\ (\mathit{text}\ s) &= \mathit{text}\ s \\ \mathit{best}\ w\ r\ (x \cup y) &= \mathit{best}\ w\ r\ x \nabla_r^w \mathit{best}\ w\ r\ y \\ \mathit{best}\ w\ r\ \emptyset &= \infty \end{aligned}$$

Fig. 8. The *best* laws.

8 Implementing Pretty-printing: A Term Representation

Now that we have developed a collection of algebraic properties of the pretty-printing operators, we can apply the methods presented in the earlier sections of the chapter to construct implementations.

(The reader may be wondering why we can't just use the abstract representation of documents as an implementation, say representing a *Doc* as a list of the possible *Layout*s. Consider for a moment a medium sized syntax tree for an imperative language, which contains 100 occurrences of **if–then–else**, each pretty-printed using *sep*. Ignoring the fact that nesting may force some *sep*s to make related choices, such a *Doc* has 2^{100} alternative layouts, and so would be represented by a list with this many elements. There is no reason to expect the best layout to be near the beginning, and so it should be clear that searching for it in such a list is a hopeless exercise.)

We will begin by deriving an implementation based on a term representation of *Doc*s. We choose simplified terms to which the *best* laws are easily applicable, which suggests

$$E ::= \mathit{text}\ S \mid \mathit{nest}\ N\ \ E \mid E\ \$\$\ E \mid E \cup E \mid \emptyset$$

However, we also want to be sure that we can apply our simplified test for $\rhd_r^w$, and so we will restrict the form of unions further. We can define a class of documents with a 'manifest' first line by

$$M ::= \mathit{text}\ S \mid \mathit{text}\ S\ \$\$\ E$$

The simplified test is easily applicable to documents of this form provided one has a longer first line than the other. We will therefore only permit unions of the form

$$U ::= M \mid U \cup U$$

and moreover we shall impose an invariant that the first line of every layout in the left operand of $\cup$ must be strictly longer than the first line of every layout in the right operand. Since both operands represent the same string it follows that all layouts in the right operand consist of at least two lines.

Now we can define simplified terms by

$$E ::= U \mid \mathit{nest}\ N\ \ E$$

We allow $\emptyset$ only at the top level of the result of *fit*,

$$E0 ::= E \mid \emptyset$$

With these restrictions *best* of a union is easily determined.

We can represent *Doc*s by the type

$$\begin{array}{lll}
\textbf{data}\ \mathit{Doc} = & \mathit{Text\ String} & \text{---}\ \mathit{text}\ s \\
 & \mid\ \mathit{String}\ \text{`}\mathit{TextAbove}\text{`}\ \mathit{Doc} & \text{---}\ \mathit{text}\ s\ \$\$\ x \\
 & \mid\ \mathit{Doc}\ \text{`}\mathit{Union}\text{`}\ \mathit{Doc} & \text{---}\ x \cup y \\
 & \mid\ \mathit{Empty} & \text{---}\ \emptyset \\
 & \mid\ \mathit{Nest\ Int\ Doc} & \text{---}\ \mathit{nest}\ k\ x
\end{array}$$

although we must be careful only to construct documents of the form described above.

We can use the same type to represent *Layout*s: a *Doc* not involving *Union* or *Empty* represents a *Layout*.

The definition of *best* is now easy to derive by applying the *best* laws — see figure 9. We'll discuss only the *Union* case. We know from the *best* laws that

$$\mathit{best}\ w\ r\ (x\ \text{`}\mathit{Union}\text{`}\ y) = \mathit{best}\ w\ r\ x \nabla_r^w \mathit{best}\ w\ r\ y$$

But since *best* must choose one of the layouts in its argument, the datatype invariant implies that if $\mathit{best}\ w\ r\ x$ is either $\mathit{text}\ s$ or $\mathit{text}\ s\ \$\$\ x'$, and $\mathit{best}\ w\ r\ y$ is $\mathit{text}\ t\ \$\$\ y'$, then $\mathit{length}\ s > \mathit{length}\ t$. So the simplified niceness comparison is applicable. If $\mathit{nice}_r^w(\mathit{text}\ s)$ then $\mathit{text}\ s \rhd_r^w\ \mathit{text}\ t$, and by the lexicographic properties it follows that $\mathit{text}\ s \rhd_r^w\ \mathit{text}\ t\ \$\$\ y'$ and $\mathit{text}\ s\ \$\$\ x' \rhd_r^w\ \mathit{text}\ t\ \$\$\ y'$. So in this case ∇_r^w chooses its left operand. If $\neg \mathit{nice}_r^w(\mathit{text}\ s)$ then the opposite holds. So we can implement ∇_r^w in this case by the function *nicest*, which simply inspects the first line of its first operand.

Haskell's lazy evaluation is exploited here, in two ways. Firstly, *shorter xs n* is defined to test whether $\mathit{length}\ \mathit{xs} \leq n$ *without* evaluating all of *xs* if it is not. Since

```
best w r (Text s) = Text s
best w r (s ‘TextAbove‘x) = s ‘TextAbove‘best w r x
best w r (Nest k x) = Nest k (best (w − k) r x)
best w r (x ‘Union‘y) = nicest  w r (best w r x) (best w r y)

nicest  w r x y = if shorter (firstline x) (w ‘min‘r) then x else y
shorter xs n = null (drop n xs)
firstline (Text s) = s
firstline (s ‘TextAbove‘x) = s
```

Fig. 9. The definition of *best*.

some layouts may have very long first lines — for example, the layout produced when all *seps* adopt a horizontal form — this is an important optimisation. Secondly, since *nicest* makes its decision on the basis of the first line of each argument *only*, then when we select the best layout from a *Union* the layout of the unsuccessful branch is evaluated only as far as the first line. Although the *Doc* we apply *best* to may be a large tree, we follow (and therefore evaluate) only a single path through it.

Definitions of *text*, *nest*, (<>) and ($$) are obtained by simple algebraic manipulation. To take just two examples,

$$\begin{aligned}(Nest\ k\ x)\ \$\$\ y &= (nest\ k\ x)\ \$\$\ y\\ &= (nest\ k\ x)\ \$\$\ (nest\ k\ (nest\ (-k)\ y))\\ &= nest\ k\ (x\ \$\$\ nest\ (-k)\ y)\\ &= Nest\ k\ (x\ \$\$\ Nest\ (-k)\ y)\end{aligned}$$

$$\begin{aligned}Text\ s <> (t\ \text{‘}TextAbove\text{‘}x) &= text\ s <> (text\ t\ \$\$\ x)\\ &= text\ s <> ((text\ \text{""} <> text\ t)\ \$\$\ x)\\ &= (text\ s <> text\ t)\ \$\$\ nest\ (length\ s)\ x \qquad [<>/\ \$\$\ \text{law}]\\ &= text\ (s +\!\!+ t)\ \$\$\ nest\ (length\ s)\ x\\ &= (s +\!\!+ t)\ \text{‘}TextAbove\text{‘}Nest\ (length\ s)\ x\end{aligned}$$

The remaining equations are derived similarly; the complete definitions appear in figure 10. It is easy to verify that the definitions terminate. We leave it to the reader to check that if the datatype invariant holds for the arguments, it also holds for the result of each these operators.

It is interesting to look at the way *Union*s are treated in these definitions. In almost every case *Union*s in arguments are 'floated upwards' to give a *Union* in the result. The exception is a *Union* in the right argument of ($$): we do not use the property

$$x\ \$\$\ (y \cup z) = (x\ \$\$\ y) \cup (x\ \$\$\ z)$$

One good reason is that to do so would violate the datatype invariant: the operands of the union on the right hand side have the *same* first lines. Another good reason is efficiency: the *Doc* form we have chosen groups together all layouts with the same first line in a value of the form *s ‘TextAbove‘ x*. The *best* function can then reject all these layouts in one go, if *s* is not nice. Here *x* may represent many billions of

$$\mathit{text}\ s = \mathit{Text}\ s$$

$$\mathit{nest}\ k\ x = \mathit{Nest}\ k\ x$$

$$\begin{aligned}
&\mathit{Text}\ s\ \$\$\ y = s\ `\mathit{TextAbove}`y\\
&(s\ `\mathit{TextAbove}`x)\ \$\$\ y = s\ `\mathit{TextAbove}`(x\ \$\$\ y)\\
&(\mathit{Nest}\ k\ x)\ \$\$\ y = \mathit{Nest}\ k\ (x\ \$\$\ \mathit{Nest}\ (-k)\ y)\\
&(x\ `\mathit{Union}`y)\ \$\$\ z = (x\ \$\$\ z)\ `\mathit{Union}`(y\ \$\$\ z)
\end{aligned}$$

$$\begin{aligned}
&\mathit{Text}\ s <> \mathit{Text}\ t = \mathit{Text}\ (s +\!\!+ t)\\
&\mathit{Text}\ s <> (t\ `\mathit{TextAbove}`x) = (s +\!\!+ t)\ `\mathit{TextAbove}`\mathit{Nest}\ (\mathit{length}\ s)\ x\\
&\mathit{Text}\ s <> (\mathit{Nest}\ k\ x) = \mathit{Text}\ s <> x\\
&\mathit{Text}\ s <> (x\ `\mathit{Union}`y) = (\mathit{Text}\ s <> x)\ `\mathit{Union}`(\mathit{Text}\ s <> y)\\
&(s\ `\mathit{TextAbove}`x) <> y = s\ `\mathit{TextAbove}`(x <> y)\\
&\mathit{Nest}\ k\ x <> y = \mathit{Nest}\ k(x <> y)\\
&(x\ `\mathit{Union}`y) <> z = (x <> z)\ `\mathit{Union}`(y <> z)
\end{aligned}$$

Fig. 10. The definitions of *text*, *nest*, (<>) and ($$).

alternative layouts, and if all *Union*s were floated to the top level then *best* would have to reject each one individually. The cost would be prohibitive, and the library simply would not work.

We still need to implement *sep* — recall its specification

$$\mathit{sep}\ xs = \mathit{fit}\ (\mathit{foldr1}\ (<\!+\!>)\ xs) \cup \mathit{foldr1}\ (\$\$)\ xs$$

We can almost use this directly as the implementation, but we must ensure that the *Union* is well-formed. Firstly, if the result of *fit* is $\emptyset$ we must avoid creating a *Union* with an empty operand. Secondly, we must ensure that the first line of the result of the *fit* is strictly longer than the first lines in the second operand. Provided *xs* consists of at least two documents this is guaranteed, since the longest first line in $(x_1\ \$\$\ x_2 \ldots\ \$\$\ x_n)$ is the longest first line in x_1, and the horizontal form contains at least one extra space. But if *xs* consists of exactly one document then the horizontal and vertical forms are the same, and a *Union* would be badly formed. So we must treat this as a special case. Thirdly, we must avoid constructing a *Union* with *nest*ed operands: this can only happen if the first *Doc* in the list is of the form *Nest k x*. In that case we factor out the *Nest*:

$$\begin{aligned}
\mathit{sep}\ (\mathit{nest}\ k\ x : xs) &= \mathit{fit}\ (\mathit{nest}\ k\ x <\!+\!> \mathit{foldr1}\ (<\!+\!>) xs) \cup\\
&\quad\ (\mathit{nest}\ k\ x\ \$\$\ \mathit{foldr1}\ (\$\$) xs)\\
&= \mathit{nest}\ k\ (\mathit{fit}\ (x <\!+\!> \mathit{foldr1}\ (<\!+\!>) xs)) \cup\\
&\quad\ \mathit{nest}\ k\ (x\ \$\$\ \mathit{foldr1}\ (\$\$)(\mathit{map}\ (\mathit{nest}\ (-k))\ xs))\\
&= \mathit{nest}\ k(\mathit{sep}\ (x : \mathit{map}\ (\mathit{nest}\ (-k))\ xs)
\end{aligned}$$

The definitions of *sep* and *fit* appear in figure 11. Notice that the datatype invariant lets us define *fit* of a *Union* very efficiently, since we know the layouts in the second operand consist of at least two lines.

```
sep [x] = x
sep (Nest k x : xs) = Nest k (sep (x : map (nest (−k)) xs))
sep xs = fit (foldr1 (<+>) xs) `u` foldr1 ($$) xs
   where Empty `u` y = y
         x `u` y = x `Union` y

fit (Text s) = Text s
fit (s `TextAbove` x) = Empty
fit (Nest k x) = case fit x of
                   Empty → Empty
                   y     → Nest k y
fit (x `Union` y) = fit x
```

Fig. 11. The definition of *sep*.

To complete the implementation of the library we just need to define a function mapping *Layout*s to appropriate strings. Let us define

$$layout :: Int \to Doc \to String$$

such that *layout k x* constructs a string displaying *nest k x*. A suitable definition appears in figure 12.

```
layout k (Text s) = indent k s
layout k (s `TextAbove` x) = indent k s ++ layout k x
layout k (Nest k' x) = layout (k + k') x

indent k s | k ≥ 8 = '\t' : indent (k − 8) s
indent k s | k ≥ 1 = ' ' : indent (k − 1) s
indent 0 s = s ++ "\n"
```

Fig. 12. Mapping layouts to strings.

One or two minor optimisations can be made. For example,

$$best\ w\ r\ ((x\ `Union`\ y)\ `Union`\ z)$$

tests x for niceness twice if it is nice — once to reject y, and once to reject z. This is easily avoided, say by redefining *best* to return a pair of the best layout and a boolean indicating whether the first line is nice. Such measures can bring a useful improvement in performance, but in fact a much more serious problem remains.

Consider for example

$$sep\ [sep\ [sep\ [\boxed{\text{hello}}, \boxed{\text{a}}], \boxed{\text{b}}], \boxed{\text{c}}]$$

$$= \left\{ \boxed{\text{hello a b c}}, \boxed{\begin{array}{l}\text{hello a b}\\ \text{c}\end{array}}, \boxed{\begin{array}{l}\text{hello a}\\ \text{b}\\ \text{c}\end{array}}, \boxed{\begin{array}{l}\text{hello}\\ \text{a}\\ \text{b}\\ \text{c}\end{array}} \right\}$$

If this document is displayed on a page of width 5 then the last layout must be chosen, but since each layout has a different first line, our implementation must first construct and reject each of the first three. Yet as soon as the length of $\boxed{\text{hello}}$ is known it is clear that the innermost *sep*, and therefore all the others, must be laid out vertically. We could therefore go immediately to the fourth layout. For large documents in which *sep* may be nested very deep, this optimisation is important. Without it the complexity of prettyprinters is at least $O(n^2)$ in the depth of *sep* nesting, and in practice they pause for an embarrassingly long time at the beginning of pretty-printing, gradually speeding up as more and more *sep* decisions are resolved.

But to incorporate this optimisation we will need to change our representation of documents.

9 Optimised Pretty-printing: A Term Representation

Looking back at the problematic example, we can see that the three first layouts have a common prefix — "hello a" — and moreover we can tell just from the prefix that none of the layouts has a nice first line. Our goal will be to factor out this common prefix, express the union of the three layouts as

$$\boxed{\text{hello a}} <> (x \cup y \cup z)$$

for suitable x, y and z, and then reject all of them together in favour of the fourth.

But to be able to observe this situation, we must introduce *text* $S <> E$ as a simplified form. At the same time we can replace the simplified forms *text* S by *text* "" and *text* S \$\$ E by *text* "" \$\$ E, because the old forms can be expressed in terms of the new ones as follows

$$\begin{aligned} \mathit{text}\ s &= \mathit{text}\ s <> \mathit{text}\ \text{""} \\ \mathit{text}\ s\ \$\$\ x &= \mathit{text}\ s <> (\mathit{text}\ \text{""}\ \$\$\ \mathit{nest}\ (-\mathit{length}\ s)\ x) \end{aligned}$$

We will need to allow $\emptyset$ in more places than before, because we intend to use the property

$$\mathit{fit}\ (\mathit{text}\ s <> x) = \mathit{text}\ s <> \mathit{fit}\ x$$

where the right hand side is a canonical form with a component (*fit* x) that might very well be empty. We don't want to *test* for an empty set here, of course, because that would make *fit* hyper-strict with disastrous consequences.

Our new simplified forms are therefore given by the grammar

$$\begin{aligned} E &::= U \mid \mathit{nest}\ N\ E \\ U &::= M \mid U \cup U \mid \emptyset \\ M &::= \mathit{text}\ \text{""} \mid \mathit{text}\ \text{""}\ \$\$\ E \mid \mathit{text}\ S <> E \end{aligned}$$

We impose the same condition on unions as before: every layout in the first operand must have a longer first line than every layout in the second.

These simplified forms can be represented by the datatype

$$\begin{array}{lll}
\mathbf{data}\ \mathit{Doc} = & \mathit{Nil} & \text{---}\ \mathit{text}\ \text{``''} \\
 & \mid\ \mathit{NilAbove}\ \mathit{Doc} & \text{---}\ \mathit{text}\ \text{``''}\ \$\$\ x \\
 & \mid\ \mathit{String}\ \text{`}\mathit{TextBeside}\text{`}\ \mathit{Doc} & \text{---}\ \mathit{text}\ s <> x \\
 & \mid\ \mathit{Nest}\ \mathit{Int}\ \mathit{Doc} & \text{---}\ \mathit{nest}\ k\ x \\
 & \mid\ \mathit{Doc}\ \text{`}\mathit{Union}\text{`}\ \mathit{Doc} & \text{---}\ x \cup y \\
 & \mid\ \mathit{Empty} & \text{---}\ \emptyset
\end{array}$$

And now the key problem is to rederive *sep* so as to *delay* introducing a *Union* until after the common prefix of the two branches of the *sep* is produced.

We need an algebraic law permitting us to draw a prefix out of a *sep*. Let us try to prove one. Assuming *xs* is non-empty, then

$$\begin{array}{ll}
 & \mathit{sep}\ ((\mathit{text}\ s <> x) : \mathit{xs}) \\
= & \mathit{fit}\ (\mathit{text}\ s <> x <+> (\mathit{foldr1}\ (<+>)\ \mathit{xs})) \cup \\
 & ((\mathit{text}\ s <> x)\ \$\$\ \mathit{foldr1}\ (\$\$)\ \mathit{xs}) \\
= & (\mathit{text}\ s <> \mathit{fit}\ (\mathit{text}\ \text{``''} <> x <+> \mathit{foldr1}\ (<+>)\ \mathit{xs})) \cup \\
 & (\mathit{text}\ s <> ((\mathit{text}\ \text{``''} <> x)\ \$\$\ \mathit{foldr1}\ (\$\$)\ (\mathit{map}\ (\mathit{nest}\ (-\mathit{length}\ s))\ \mathit{xs}) \\
= & \mathit{text}\ s <> \mathit{sep}\ ((\mathit{text}\ \text{``''} <> x) : \mathit{map}\ (\mathit{nest}\ (-\mathit{length}\ s))\ \mathit{xs})
\end{array}$$

This last step holds because *nest* can be either cancelled or introduced freely in the horizontal alternative. We have already seen that we can move a *Nest* out of a *sep*, and indeed we can even move a *Union* out of *sep*'s first argument *without* splitting the *sep* into two branches which must be explored separately. In fact the only time that we have to do this is when the first argument is *Nil* — and by that point the horizontal and vertical alternatives differ at the very next character, so there is really no alternative. The derived definition of *sep* is given in figure 13. We have used an auxiliary function specified by

$$\mathit{sep}'\ x\ k\ \mathit{ys} = \mathit{sep}\ (x : \mathit{map}\ (\mathit{nest}\ k)\mathit{ys})$$

to avoid repeated applications of *nest* to the remaining arguments.

Implementations of the other four operators can be derived in the usual way — this time we skip the details. The resulting definitions are presented in figure 14. Once again we leave it to the reader to check that the datatype invariant is satisfied.

In fact, these are not quite the implemented definitions. Heap profiling revealed that the derived definition of ($$) leaks space: unevaluated calls of ($$) and *nest* collect on the heap. These are introduced in the 3rd and 4th equations for ($$), and unfortunately passed to a recursive call of ($$) which usually introduces still more unevaluated applications. A solution is to avoid constructing these unevaluated applications at all by using an auxiliary function

$$\mathit{aboveNest}\ x\ k\ y = x\ \$\$\ \mathit{nest}\ k\ y$$

instead. This is of course just the specification of *aboveNest*; the derived implementation appears in figure 15. It is important that *aboveNest*s second parameter is

```
sep [x] = x
sep (x : ys) = sep' x 0 ys

sep' Nil k ys = fit (foldl (<+>) Nil ys) ‘Union‘vertical Nil k ys
sep' (NilAbove x) k ys = vertical (NilAbove x) k ys
sep' (s ‘TextBeside‘x) k ys = s ‘TextBeside‘sep' (Nil <> x) (k − length s) ys
sep' (Nest n x) k ys = Nest n (sep' x (k − n) ys)
sep' (x ‘Union‘y) k ys = sep' x k ys ‘Union‘vertical y k ys
sep' Empty k ys = Empty

vertical x k ys = x $$ nest k (foldr1 ($$) ys)
```

Fig. 13. *sep* optimised to delay *Union*.

```
text s = s ‘TextBeside‘Nil

nest k x = Nest k x

Nil <> (Nest k x) = Nil <> x
Nil <> x = x
NilAbove x <> y = NilAbove (x <> y)
(s ‘TextBeside‘x) <> y = s ‘TextBeside‘(x <> y)
Nest k x <> y = Nest k (x <> y)
(x ‘Union‘y) <> z = (x <> z) ‘Union‘(y <> z)
Empty <> z = Empty

Nil $$ x = NilAbove x
NilAbove x $$ y = NilAbove (x $$ y)
(s ‘TextBeside‘x) $$ y = s ‘TextBeside‘((Nil <> x) $$ nest (−length s)y)
Nest k x $$ y = Nest k (x $$ nest (−k) y)
(x ‘Union‘y) $$ z = (x $$ z) ‘Union‘(y $$ z)
Empty $$ y = Empty
```

Fig. 14. Implementations of *text*, *nest*, (<>) and ($$).

evaluated strictly — otherwise the heap would fill up with unevaluated subtractions instead. We can arrange this using `hbc`'s standard function $seq\ a\ b$, which evaluates a and returns the value of b.

And now we must derive an implementation of *best*.

The trickiest case is $best\ w\ r\ (text\ s <> x)$. We know that this must be equal to $text\ s <> y$ for some y — but what is y? It clearly depends on both x and s, because the length of s affects the width of ‘ribbon’ available to the first line of x. Let us introduce a new function *best'* , whose defining property is

$$best\ w\ r\ (text\ s <> x) = text\ s <> best'\ w\ r\ s\ x$$

We can derive a definition for *best'* using the algebra; we present the details this

```
x $$ y = aboveNest x 0 y

aboveNest Nil k y = NilAbove (nest k y)
aboveNest (NilAbove x) k y = NilAbove (aboveNest x k y)
aboveNest (s `TextBeside` x) k y = seq k' (s `TextBeside` aboveNest (Nil <> x) k' y)
                                        where k' = k - length s
aboveNest (Nest k' x) k y = seq k'' (Nest k' (aboveNest x k'' y))
                                where k'' = k - k'
aboveNest (x `Union` y) k z = aboveNest x k z `Union` aboveNest y k z
aboveNest Empty k z = Empty
```

Fig. 15. Defining $\$\$$ without a space leak.

time.

$$\begin{aligned} \mathit{text}\ s <> \mathit{best}'\ w\ r\ s\ \mathit{Nil} &= \mathit{best}\ w\ r\ (\mathit{text}\ s <> \mathit{text}\ \text{""}) \\ &= \mathit{best}\ w\ r\ (\mathit{text}\ s) \\ &= \mathit{text}\ s \\ &= \mathit{text}\ s <> \mathit{Nil} \end{aligned}$$

so we can take $\mathit{best}'\ w\ r\ s\ \mathit{Nil} = \mathit{Nil}$.

$$\begin{aligned} &\mathit{text}\ s <> \mathit{best}'\ w\ r\ s\ (\mathit{NilAbove}\ s) \\ &= \mathit{best}\ w\ r\ (\mathit{text}\ s <> (\mathit{text}\ \text{""}\ \$\$\ x)) \\ &= \mathit{best}\ w\ r\ (\mathit{text}\ s\ \$\$\ \mathit{nest}\ (\mathit{length}\ s)\ x) \\ &= \mathit{text}\ s\ \$\$\ \mathit{nest}\ (\mathit{length}\ s)(\mathit{best}\ (w - \mathit{length}\ s)\ r\ x) \\ &= \mathit{text}\ s <> (\mathit{text}\ \text{""}\ \$\$\ \mathit{best}\ (w - \mathit{length}\ s)\ r\ x) \end{aligned}$$

so we can take

$$\mathit{best}'\ w\ r\ s\ (\mathit{NilAbove}\ x) = \mathit{NilAbove}\ (\mathit{best}\ (w - \mathit{length}\ s)\ r\ x)$$

For the *TextBeside* case,

$$\begin{aligned} &\mathit{text}\ s <> \mathit{best}'\ w\ r\ s\ (t\ \text{`}\mathit{TextBeside}\text{`}\ x) \\ &= \mathit{best}\ w\ r\ (\mathit{text}\ s <> \mathit{text}\ s <> x) \\ &= \mathit{text}\ s <> \mathit{text}\ t <> \mathit{best}'\ w\ r\ (s +\!\!+ t)\ x \end{aligned}$$

so we can take

$$\mathit{best}'\ w\ r\ s\ (t\ \text{`}\mathit{TextBeside}\text{`}\ x) = t\ \text{`}\mathit{TextBeside}\text{`}\ \mathit{best}'\ w\ r\ (s +\!\!+ t)\ x$$

The *Nest* case is very simple:

$$\begin{aligned} \mathit{text}\ s <> \mathit{best}'\ w\ r\ s\ (\mathit{Nest}\ k\ x) &= \mathit{best}\ w\ r\ (\mathit{text}\ s <> \mathit{nest}\ k\ x) \\ &= \mathit{best}\ w\ r\ (\mathit{text}\ s <> x) \\ &= \mathit{text}\ s <> \mathit{best}'\ w\ r\ s\ x \end{aligned}$$

so $\mathit{best}'\ w\ r\ s\ (\mathit{Nest}\ k\ x) = \mathit{best}'\ w\ r\ s\ x$. Finally,

$$\begin{aligned}
&\textit{text}\ s \Diamond \textit{best}'\ w\ r\ s\ (x\ \text{`}\textit{Union}\text{`}y)\\
&= \textit{best}\ w\ r\ (\textit{text}\ s \Diamond (x \cup y))\\
&= \textit{best}\ w\ r\ (\textit{text}\ s \Diamond x)\nabla^w_r \textit{best}\ w\ r\ (\textit{text}\ s \Diamond y)\\
&= (\textit{text}\ s \Diamond \textit{best}'\ w\ r\ s\ x)\nabla^w_r (\textit{text}\ s \Diamond \textit{best}'\ w\ r\ s\ y)\\
&= \textit{text}\ s \Diamond (\textit{best}'\ w\ r\ s\ x\ \nabla^w_r{}'(s)\ \textit{best}'\ w\ r\ s\ y)
\end{aligned}$$

where we have introduced a new operator whose defining property is that

$$\textit{text}\ s \Diamond (x\ \nabla^w_r{}'(s)\ y) = (\textit{text}\ s \Diamond x)\nabla^w_r (\textit{text}\ s \Diamond y)$$

But recall that because of the invariant that *Union*s satisfy, ∇^w_r chooses its left argument if and only if its first line is nice. But if s is already longer than (w '*min*'r), then no *text* $s \Diamond x$ can have a nice first line. So in this case $\nabla^w_r{}'(s)$ can choose its right argument without looking at either one! This is the optimisation we have been trying to capture: just by looking at the common prefix we can select the right branch, and thereby the vertical form for the *sep* from which the *Union* came. The complete definition of *best* appears in figure 16.

```
best w r Nil = Nil
best w r (NilAbove x) = NilAbove (best w r x)
best w r (s ‘TextBeside‘x) = s ‘TextBeside‘best′ w r s x
best w r (Nest k x) = Nest k (best (w − k) r x)
best w r (x ‘Union‘y) = nicest w r (best w r x) (best w r y)
best w r Empty = ∞

best′ w r s Nil = Nil
best′ w r s (NilAbove x) = NilAbove (best (w − length s) r x)
best′ w r s (t ‘TextBeside‘x) = t ‘TextBeside‘best′ w r (s ++t)x
best′ w r s (Nest k x) = best′ w r s x
best′ w r s (x ‘Union‘y) = nicest′ w r s (best′ w r s x) (best′ w r s y)
best′ w r s Empty = ∞
nicest w r x y = nicest′ w r "" x y

nicest′ w r s x y = if fits (w ‘min‘r) (length s) x then x else y

fits n k x = if n < k then false else
                    case x of
                       Nil              → true
                       NilAbove y       → true
                       t ‘TextBeside‘y  → fits n (k + length t) y
                       ∞                → false
```

Fig. 16. The optimised definition of *best*.

Once again minor improvements can be made to the implementation. Quite a substantial speed-up is obtained by storing strings with their length — that is, strings are represented within the library by a pair of their length and their characters.

String concatenation is used heavily in the library and is performed in constant time: it consists of addition of the lengths and composition of the characters, which are represented by a function as in section 4.2.

This implementation of the library is a major improvement on the previous ones. There are no 'embarrassing pauses'. While the cost of pretty-printing seems to grow slightly faster than linearly, the library is able to produce large outputs (>200K) in little space and reasonable time. On a SPARC ELC a benchmark program with deeply nested *sep*s evaluated between 500 and 1000 *sep*s per second. Performance is quite acceptable, and far superior to both the earlier term-based implementation (sometimes $O(n^2)$) and the seat-of-the-pants implementation (which was actually sometimes exponential).

10 A Context-passing Pretty-printer

The key observation in the development of the efficient combinators in the last section was that

$$sep\ ((text\ s <> x) : xs) = text\ s <> sep\ ((text\ \text{""} <> x) : map\ (nest\ (-length\ s))\ xs)$$

and so we can 'factor out' all the text in the first element of a *sep* before splitting the computation into a *Union* of two alternatives. We exploited the observation by making $text\ s <> x$ into a simplified form, and testing for it in *sep*. But we could equally well have derived a context-passing implementation, in which *text* tests for the presence of an enclosing *sep*. Indeed, it seems natural to think of a *Doc* as a function that chooses a layout depending on the context, and this is how the very first implementation of the combinators was constructed.

What kind of contexts should we consider? Certainly observations of the form $best\ w\ r\ [\bullet]$ — that is, we should be able to lay out a document with a given page and ribbon width. We will also need to lay out documents with a given indentation, that is consider contexts of the form $best\ w\ r\ (nest\ k\ [\bullet])$. If we take $k = 0$ then this form subsumes the first.

Now imagine that a union appears in such a context. We can simplify as follows:

$$best\ w\ r\ (nest\ k\ (x \cup y)) = best\ w\ r\ (nest\ k\ x) \nabla_r^w best\ w\ r\ (nest\ k\ y)$$

We expect to continue working on x, so we must be able to represent contexts of the form $best\ w\ r\ (nest\ k\ [\bullet]) \nabla_r^w b$ also. We can think of b as the layout to choose if we are forced to backtrack. Once again, the conditions on unions will enable us to decide which of x and b to choose purely on the basis of the value of x.

Of course, in order to apply the key optimisation we must be able to recognise when a document is the first element of an enclosing *sep*. We shall therefore need contexts of the form $C[sep\ [[\bullet], y_1 \ldots y_n]]$. Moreover, the optimisation applies to documents of the form $text\ s <> x$. But when such a document appears at the top level, we shall need to evaluate

$$best\ w\ r\ (nest\ k\ (text\ s <> x))$$

To do so we must be able to evaluate x, and we therefore need to be able to represent its context in this expression. We shall add contexts of the form $C[\mathit{text}\ s <> [\bullet]]$ to cover this case.

When we lay out $x <> y$ and $x \$\$ y$, we shall start by laying out x. We therefore have to represent the contexts $C[[\bullet] <> y]$ and $C[[\bullet] \$\$ y]$. And when we expand a *sep* into a union of two alternatives, the horizontal alternative appears inside *fit*. We must therefore represent contexts of the form $C[\mathit{fit}\ [\bullet]]$ also.

So we choose contexts of the forms

$$\begin{aligned} C[\bullet] ::= {} & \mathit{best}\ N\ N\ (\mathit{nest}\ N\ [\bullet]) \\ \mid {} & \mathit{best}\ N\ N\ (\mathit{nest}\ N\ [\bullet]) \nabla_r^w E \\ \mid {} & C[\mathit{text}\ s <> [\bullet]] \\ \mid {} & C[\mathit{sep}\ [[\bullet], E \ldots E]] \\ \mid {} & C[\mathit{fit}\ [\bullet]] \\ \mid {} & C[[\bullet] <> E] \\ \mid {} & C[[\bullet] \$\$ E] \end{aligned}$$

where N represents integer expressions, and E represents document expressions. Contexts can be represented by the following Haskell datatype:

$$\begin{aligned} \mathbf{data}\ \mathit{Cxt}\ = {} & \mathit{BestNest}\ \mathit{Int}\ \mathit{Int}\ \mathit{Int} \\ \mid {} & \mathit{BestNestOr}\ \mathit{Int}\ \mathit{Int}\ \mathit{Int}\ \mathit{Doc} \\ \mid {} & \mathit{TextBeside}\ \mathit{String}\ \mathit{Cxt} \\ \mid {} & \mathit{Sep}\ [\mathit{Doc}]\ \mathit{Cxt} \\ \mid {} & \mathit{Fit}\ \mathit{Cxt} \\ \mid {} & \mathit{Beside}\ \mathit{Doc}\ \mathit{Cxt} \\ \mid {} & \mathit{Above}\ \mathit{Doc}\ \mathit{Cxt} \end{aligned}$$

Must we consider such complex contexts, or can we apply the laws of the pretty-printing algebra to simplify them? Unfortunately, we have been unable to eliminate any of the forms of context given above. Certainly, some context simplifications are possible. In particular, we can always move *TextBeside* up to the top level — this is after all the observation that the key optimisation is based on. But we cannot usefully combine *TextBeside* with the enclosing *BestNest* or *BestNestOr*, because there would then be no way to express a *BestNest* *without* a *TextBeside*: no instance of

$$\mathit{best}\ w\ r\ (\mathit{nest}\ k\ (\mathit{text}\ s <> [\bullet]))$$

is equal to

$$\mathit{best}\ w\ r\ (\mathit{nest}\ k\ [\bullet])$$

because $\mathit{text}\ \text{``''} <> x \neq x$ in general.

We can also use the facts

$$\begin{aligned} \mathit{fit}\ (x \$\$ y) &= \emptyset \\ \mathit{sep}\ ((x \$\$ y) : zs) &= x \$\$ y \$\$ \mathit{foldr1}\ (\$\$) zs \\ (x \$\$ y) \$\$ z &= x \$\$ (y \$\$ z) \\ (x \$\$ y) <> z &= x \$\$ (y <> z) \end{aligned}$$

to simplify contexts in which *Above* occurs inside *Fit*, *Sep*, *Above* or *Beside*. If we could always move *Above* to the top level, we could apply

$$best\ w\ r\ (nest\ k\ (x\ \$\$\ y)) = best\ w\ r\ (nest\ k\ x)\ \$\$\ best\ w\ r\ (nest\ k\ y)$$

But alas, we cannot simplify $text\ s <> (x\ \$\$\ y)$ without knowing more about x.

In fact there is no form of context which can *always* be simplified away, and we must just work with this rather complex set.

Now that the contexts have been chosen, the actual derivation of an implementation follows exactly the same method as in earlier sections. We will not go through the details. We simply remark that, just as in the previous section, the implementation has a space leak. 'Pending' applications of *nest* fill up the heap. And to avoid this, just as in the previous section, we combine an application of *nest* with other operators. In this case we define two forms of context with a 'built-in' *nest*:

$$\begin{aligned} AboveNest\ k\ y\ C &= C[[\bullet]\ \$\$\ nest\ k\ y] \\ SepNest\ k\ ys\ C &= C[sep\ ([\bullet] : map\ (nest\ k)\ ys)] \end{aligned}$$

In the derived implementation, when we exploit

$$sep\ ((text\ s <> x) : xs) = text\ s <> sep\ ((text\ \text{""} <> x) : map\ (nest\ (-length\ s))\ xs)$$

and the corresponding property for (\$\$), we just have to change a number in x's context, instead of building applications of *nest*.

Evaluation of the Context-passing Combinators This version of the pretty-printing library is definitely more complex than the term-based versions, as a consequence of the rather complex forms of context we were forced to work with. It is also harder to modify: in particular, a change to the way the best layout is chosen would have far reaching effects. In the term-based libraries, *best* is a separate function and may be replaced with another without altering the rest of the library. But in the context-passing library, every combinator knows how to behave in a *BestNest* context: the criterion for selecting the best layout is distributed throughout the code.

This could be a fair price to pay for better performance. But at least in my implementation, the context passing library is (a little) slower than the term based one, and uses (a little) more space. Its only advantage seems to be that it does not require lazy evaluation, as the term based library does (to make traversing one path through an enormous tree efficient). If one were to reimplement the pretty-printing library in a strict functional language such as ML, the context passing version might prove more efficient than simulating laziness with references and nullary functions.

Relationship to the Original Implementation The first implementation of the pretty-printing combinators was indeed based on context-passing, with contexts represented by a five-tuple containing the page width, ribbon width, length of text to the left (*c.f.* $C[text\ s <> [\bullet]]$), a boolean forcing a one-line layout (*c.f.* $C[fit\ [\bullet]]$, and a boolean indicating whether the surrounding context was horizontal or vertical. Such a design seems natural, if one intuitively expects a pretty-printer just to maintain a little state (the context) to guide layout choices. But as we have seen, this context information is not sufficient to implement the correct behaviour of the combinators — which was an obstacle to the *discovery* of the simple specification they now satisfy.

Moreover the performance of the combinators was poor, at first exponential in the depth of *sep*-nesting, later improved to square. Further optimisations were hard to find, because of the lack of a good algebra, and no doubt also because of the necessary complexity of the solution — the efficient context-passing library described in this section is nothing one would stumble on by accident.

The first implementation was developed rapidly, and its usefulness was certainly an inspiration to develop the solutions presented in this chapter. But in retrospect, the seemingly natural choice of a context-passing implementation was unfortunate. Abandoning that choice, and working with a more abstract specification and systematic program development, led both to better behaviour and much more efficient implementations.

11 A Comparison with Oppen's Pretty-printer

The classic work in 'language independent pretty-printing' is Oppen's pretty-printer [3]. He defined a small language for expressing documents, and an interpreter for the language which generates a pretty layout. The output of a user's pretty-printer is thus intended to be piped through the interpreter. The interpreter is written in an imperative language, and its space requirements are small.

The similarity between Oppen's language and my pretty-printing combinators is striking. Oppen provides equivalents of *text*, *sep*, and *nest*, and his language can also express (<>), although well-formed documents should not contain it. Oppen also provides a variant of *sep* which places as many elements as will fit on one line, then places more on the next line, and so on. An equivalent combinator could very usefully be added to my pretty-printing library.

On the other hand, Oppen's interpreter is quite large and hard to understand. His paper describes its behaviour for 'well-formed' inputs, but the interpreter accepts a wider class of inputs, and its behaviour on the others is hard to predict. The interpreter defines the meaning of every program, but in a monolithic way — there is no way to describe the meaning of one construct in isolation. Moreover it isn't clear which of the possible layouts the interpreter actually chooses. One way to regard the pretty-printing combinators is as a candidate for a denotational semantics of Oppen's language.

Oppen's interpreter is probably more efficient than our combinators, but on the other hand our libraries are probably easier to modify. For example, to make the pretty-printer look ahead a few lines and avoid imminent line overflows by breaking lines earlier, rather than making decisions only on the basis of the current line, we would just need to redefine the *best* function. At least with the first two implementations we described, the other combinators could be reused as they are. It is not at all clear what changes would need to be made to Oppen's interpreter to achieve the same effect.

Exercise 5. Specify and implement Oppen's *sep*-variant, which allows several elements per line in a vertical layout. *Warning* this is a substantial exercise!

12 Conclusions

In this chapter we have considered the design of combinator libraries. We saw how studying the algebraic properties of the combinators desired can both help to suggest natural choices of representation, and guide the implementation of the operators. We saw several examples — lists, monads, and a pretty-printing library. For this kind of program development we need a language with higher-order functions and lazy evaluation, for which equational reasoning is valid; in other words, Haskell is ideally suited.

In the case of pretty-printing, studying the algebra led to the correction of a subtle error in the combinators' behaviour, and to the development of much more efficient implementations. The pretty-printing algebra is just too intricate to rely on intuition alone: working informally I could not see how to implement the optimisation considered in section 9, nor could I invent the representation used there. The formal approach has been invaluable.

The pretty-printing library itself has proved useable, despite its simplicity. Indeed, versions of it have seen quite extensive use, in program transformation tools, proof assistants, and compilers. All the pretty-printers in both the Chalmers and the Glasgow Haskell compilers are written using variants of this design.

References

1. Lennart Augustsson, *Haskell B. user's manual*, available over WWW from `http://www.cs.chalmers.se:80/pub/haskell/chalmers/`.
2. Konstantin Läufer, *Combining Type Classes and Existential Types*, Proc. Latin American Informatics Conference (PANEL), ITESM-CEM, Mexico, September 1994.
3. Derek C. Oppen, *Pretty-printing,* in ACM Transactions on Programming Languages and Systems, Vol. 2, No. 4, October 1980.

A The Optimised Pretty-printing Library

```
module NewPP(Doc,(<>),($$),text,sep,nest,pretty) where
import Seq

infixl <>
infixl $$

data Doc = Nil                 -- text ""
         | NilAbove Doc        -- text "" $$ x
         | Str `TextBeside` Doc-- text s <> x
         | Nest Int Doc        -- nest k x
         | Doc `Union` Doc     -- x U y
         | Empty               -- {}
         deriving (Text)

type Str = (Int,String->String)
    -- optimised rep of strings: fast length, fast concat.
len (i,_) = i
(i,s) `cat` (j,t) = (i+j,s.t)
str s = (length s,(s++))
string (i,s) = s []

text s = str s `TextBeside` Nil

nest k x = Nest k x

x $$ y = aboveNest x 0 y

aboveNest Nil k y = NilAbove (nest k y)
aboveNest (NilAbove x) k y = NilAbove (aboveNest x k y)
aboveNest (s `TextBeside` x) k y =
  seq k'
  (s `TextBeside` (aboveNest (Nil<>x) k' y))
  where k' = k-len s
aboveNest (Nest k' x) k y =
  seq k'' (Nest k' (aboveNest x k'' y))
  where k'' = k-k'
aboveNest (x `Union` y) k z =
  aboveNest x k z `Union` aboveNest y k z
aboveNest Empty k x = Empty
```

```
Nil <> (Nest k x) = Nil <> x
Nil <> x = x
NilAbove x <> y = NilAbove (x <> y)
(s `TextBeside` x) <> y = s `TextBeside` (x <> y)
Nest k x <> y = Nest k (x <> y)
Empty <> y = Empty
(x `Union` y) <> z = (x <> z) `Union` (y <> z)

sep [x] = x
sep (x:ys) = sep' x 0 ys

sep' Nil k ys = fit (foldl (<+>) Nil ys)
                `Union` vertical Nil k ys
sep' (NilAbove x) k ys = vertical (NilAbove x) k ys
sep' (s `TextBeside` x) k ys =
  s `TextBeside` sep' (Nil <> x) (k-len s) ys
sep' (Nest n x) k ys = Nest n (sep' x (k-n) ys)
sep' (x `Union` y) k ys = sep' x k ys `Union` vertical y k ys
sep' Empty k ys = Empty

vertical x k ys = x $$ nest k (foldr1 ($$) ys)
x <+> y = x <> text " " <> y

fit Nil = Nil
fit (NilAbove x) = Empty
fit (s `TextBeside` x) = s `TextBeside` (fit x)
fit (Nest n x) = Nest n (fit x)
fit (x `Union` y) = fit x
fit Empty = Empty

best w r Nil = Nil
best w r (NilAbove x) = NilAbove (best w r x)
best w r (s `TextBeside` x) = s `TextBeside` best' w r s x
best w r (Nest k x) = Nest k (best (w-k) r x)
best w r (x `Union` y) = nicest w r (best w r x) (best w r y)
best w r Empty = Empty

best' w r s Nil = Nil
best' w r s (NilAbove x) = NilAbove (best (w-len s) r x)
best' w r s (t `TextBeside` x) =
  t `TextBeside` best' w r (s `cat` t) x
best' w r s (Nest k x) = best' w r s x
best' w r s (x `Union` y) =
  nicest' w r s (best' w r s x) (best' w r s y)
best' w r s Empty = Empty
```

```
nicest w r x y = nicest' w r (str "") x y
nicest' w r s x y = if fits (w `min` r) (len s) x then x else y

fits n k x = if n<k then False else
               case x of
                 Nil -> True
                 NilAbove y -> True
                 t `TextBeside` y -> fits n (k+len t) y
                 Empty -> False

layout k (Nest k' x) = layout (k+k') x
layout k x = [' ' | i<-[1..k]] ++ layout' k x

layout' k Nil = "\n"
layout' k (NilAbove x) = "\n" ++ layout k x
layout' k (s `TextBeside` x) = string s ++ layout' (k+len s) x

pretty w r d = layout 0 (best w r d)
```

This article was processed using the LaTeX macro package with LLNCS style