Correct-by-Construction Pretty-Printing

Nils Anders Danielsson
University of Gothenburg & Chalmers University of Technology
nad@cse.gu.se

Abstract

A new approach to correct-by-construction pretty-printing is presented. The basic methodology is the one of classical (not necessarily correct) pretty-printing: users convert values to pretty-printer documents, and a general rendering algorithm turns documents into strings. The main novelty is that dependent types are used to ensure that, for each value, the constructed document is correct with respect to the value and a given grammar. Other parts of the development use well-established technology: the pretty-printer document interface is basically that of Wadler (2003), but with more precise types, and a single additional primitive combinator; and Wadler's rendering algorithm is used.

It is proved that if a given value is pretty-printed, and the resulting string parsed (with respect to the same, unambiguous grammar), then the original value is obtained. No guarantees are made about "prettiness".

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Mechanical verification; F.4.2 [*Mathematical Logic and Formal Languages*]: Grammars and Other Rewriting Systems

Keywords dependent types; pretty-printing

1. Introduction

Pretty-printing is concerned with formatting text in a "pretty" way, given a bounded line width. For instance, given a line width of at least 23 the expression $1 + 2 * (3 + 4) + 5 * 6$ may be formatted as follows:

```
1 + 2 * (3 + 4) + 5 * 6
```

However, if the line width is smaller (but at least 13), then the following layout may be used instead:

```
1 +
2 * (3 + 4) +
5 * 6
```

Other options are possible.

There are several programs/combinator libraries that can be used to construct pretty-printers, for instance those due to Oppen (1980), Hughes (1995), Wadler (2003), and Swierstra and Chitil (2009). Libraries based on Hughes' approach tend to be set up more or less in the following way: There is a type *Doc* of pretty-printer documents, and a renderer $render : Doc \to String$; the renderer often takes additional arguments, for instance the line width. There are also a number of combinators for constructing documents. A library user who wants to pretty-print values of type A can write a function of type $A \to Doc$, using the combinators, and compose this function with *render* to get a function of type $A \to String$.

Often a pretty-printer is constructed together with a parser. In this case one is typically interested in at least one round-tripping property: the result of pretty-printing some value x, and then parsing the resulting string, should be x. Rendel and Ostermann (2010) argue that separate definitions of parsers and pretty-printers lead to redundancy and perhaps inconsistencies, and present combinators that allow the simultaneous definition of parsers and printers. Matsuda and Wang (2013) attack the problem in a somewhat different way. They show how, starting from an extended pretty-printer, one can use program inversion techniques to automatically derive a parser that satisfies the round-tripping property (assuming that the underlying grammar is unambiguous). A typical pretty-printer does not contain enough information to construct a parser, so they introduce a biased choice operator: $p <\!\!+\ q$ means "pretty-print according to p, but be ready to parse also according to q". This makes it possible to encode many grammars in their extended pretty-printing language.

In this paper I propose a different approach to correct-by-construction pretty-printing. I do not (always) want to conflate grammars and pretty-printers, because there are situations in which I do not want to make grammars harder to construct or understand. Furthermore, as pointed out by Boulton (1996), one may want to support several different pretty-printers corresponding to a given grammar.

Instead of using hybrid pretty-printers/grammars I follow the approach due to Hughes, but with a twist. The library user is not only asked to convert values to documents, but is also asked to define a grammar. Types are used to ensure that, for each value, the generated document matches the grammar and the value.

Section 2 presents a grammar data type: *Grammar A* stands for grammars with semantic actions, where the results have type A. In the paper I do not focus on parsing, but give a formal semantics for grammars: $x \in g \cdot s$ means that the string s and corresponding result x are generated by the grammar g. The type family $\_\!\in\!\_\cdot\_$ is defined as a data type, and values of type $x \in g \cdot s$ can be seen as parse trees.

Pretty-printers for the grammar $g : Grammar\ A$ are defined as functions from values to corresponding pretty-printer documents:

$$(x : A) \to Doc\ g\ x$$

Here the simple type $A \to Doc$ used by Hughes has been refined to a dependent type. *Doc*, defined in Section 3, is based on the document type used by Wadler (2003), but is indexed by a grammar

DTP '13, September 24, 2013, Boston, MA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2384-0/13/09…$15.00.
http://dx.doi.org/10.1145/2502409.2502410

and a value. Note that, unlike a parse tree of type $x \in g \cdot s$, a document of type *Doc g x* is not indexed by a string: a given document can (potentially) be turned into a string in many different ways.

When a user has defined a grammar g : *Grammar A* and a corresponding pretty-printer *pretty* : $(x : A) \to$ *Doc g x* the remaining job is taken care of by a renderer:

$$render : Doc\ g\ x \to String$$

The renderer turns documents into strings, and may, depending on its inputs, have some leeway in deciding how to handle line breaks and indentation. Note that, unlike the type of *pretty*, the type of *render* does not ensure that *render* is correct by construction. Renderers are instead required to satisfy the following property, which can be proved once per renderer:

$$(d : Doc\ g\ x) \to x \in g \cdot render\ d$$

If we instantiate d with *pretty x*, then we get

$$(x : A) \to x \in g \cdot render\ (pretty\ x),$$

which for unambiguous grammars g implies the round-tripping property: the result of parsing *render* (*pretty x*) is x. The paper does not contain any new rendering algorithm; the focus is on grammatical correctness. However, two renderers are presented in Section 5, one of them based on Wadler's pretty-printing algorithm (2003).

In short, the paper makes the following contributions:

- A framework for correct-by-construction pretty-printing, based on indexed pretty-printer documents, is presented. Note that correctness only concerns grammatical correctness: no guarantee is made that the output will be pretty.
- Several small examples, indicating that the framework is usable in practice, are presented (see Section 4). Larger examples are available in the accompanying code.
- As far as I am aware this is the first example of a formal, mechanised correctness proof for a pretty-printer (as opposed to an "ugly-printer" that ignores line widths, word wrapping, indentation, etc.).

Related work is discussed in Section 6.

The pretty-printing framework and examples described in the paper (along with omitted proofs, and with minor differences) have been made available to download. The code is implemented in the dependently typed programming language Agda (Norell 2007; Agda Team 2013). In the paper I deviate somewhat from Agda notation in order to avoid clutter and aid readability; for instance, Agda's somewhat idiosyncratic notation for coinductive types and corecursive definitions is not used.

2. Grammars

This section presents the type of grammars that is used. I have chosen to use monadic, potentially infinite grammars in order to emphasise that the approach can handle very general grammars. However, the approach is not restricted to such grammars. It is for instance possible to use regular expressions (extended with semantic actions) instead. Regular expressions are not general enough to handle all of the examples in Section 4, but the basic ideas carry over unchanged.

I use two definitions of grammars. A simple one presented here, and an equally expressive variant with more constructors, defined in the accompanying code. The extra constructors are used for some proof automation described in Section 4.3, but are not strictly necessary.[1] The simple type of grammars is defined in the following way:[2]

```
data Grammar : Set → Set₁ where    -- Coinductive.
  return : A → Grammar A
  token  : Grammar Char
  _|_    : Grammar A → Grammar A → Grammar A
  _>>=_  : Grammar A → (A → Grammar B) →
           Grammar B
```

The constructor return x accepts only the empty string, and returns x; token accepts and returns arbitrary single tokens (characters); _|_ is symmetric choice; and _>>=_ is monadic sequencing.

The type of grammars should be read *coinductively*. This means that one can construct infinite grammars, for instance a grammar for the empty language:

```
fail : Grammar A
fail = fail | fail
```

If the grammar type were read inductively, then the grammar formalism would be quite restrictive: it would be impossible to construct a grammar that accepted strings of arbitrary length (assuming that the number of characters is finite). However, the coinductive grammars above are very expressive: they can represent every recursively enumerable language (using grammars of the form $g_0 \mid (g_1 \mid (g_2 \mid \ldots)))$). This means that it is not always possible to implement a parser for these grammars. In practice one may want to restrict attention to a smaller class of grammars, for which parsing is always possible or perhaps even efficient.

The semantics of a grammar is defined by the following data type, which should be read inductively; $x \in g \cdot s$ means that the string s and corresponding result x are generated by the grammar g:

```
data _∈_·_ : A → Grammar A → String → Set₁ where
  return-sem : x ∈ return x · [ ]
  token-sem  : t ∈ token · [t]
  >>=-sem    : x ∈ g₁ · s₁ → y ∈ g₂ x · s₂ →
               y ∈ g₁ >>= g₂ · s₁ ++ s₂
  left-sem   : x ∈ g₁ · s → x ∈ g₁ | g₂ · s
  right-sem  : x ∈ g₂ · s → x ∈ g₁ | g₂ · s
```

(Strings are taken to be lists of characters: *String* = *List Char*.) Readers who are unfamiliar with this kind of definition may want to see it as an inductively defined inference system, as in Figure 1.

An alternative reading of $x \in g \cdot s$ is "one of the results of parsing the string s using the grammar g is x", but note that it may not be possible to implement a (total) parser for g. As mentioned in Section 1, values of type $x \in g \cdot s$ can be seen as parse trees. In some cases it may also be appropriate to see the values x as abstract syntax trees.

As an example we can show that the language defined by *fail* is empty, or, in other words, that for any x and s it is impossible that $x \in fail \cdot s$ is inhabited (*Empty* is the empty type):

```
fail-empty : x ∈ fail · s → Empty
fail-empty (left-sem  p) = fail-empty p
fail-empty (right-sem p) = fail-empty p
```

[1] The extra constructors also have a second use: Agda's productivity checker requires corecursive definitions to be syntactically guarded, and the extra constructors can make it easier to write guarded definitions. In the paper I ignore guardedness, and present some corecursive definitions that are not syntactically guarded (but still productive).

[2] *Set* is a type of small types, and *Set*$_1$ is a type of types that includes *Set*. Here and later I omit most implicit argument declarations; for instance, the full type of return is $\{A : Set\} \to A \to Grammar\ A$. (If a function has type $\{x : A\} \to T$, then its first argument x is implicit, and does not need to be given explicitly as long as Agda can infer it.)

$$\frac{}{x \in \mathsf{return}\, x \cdot [\,]}\ (\textsf{return-sem})$$

$$\frac{}{t \in \mathsf{token} \cdot [t]}\ (\textsf{token-sem})$$

$$\frac{x \in g_1 \cdot s_1 \qquad y \in g_2\, x \cdot s_2}{y \in g_1 \ggg\!= g_2 \cdot s_1 +\!\!+\ s_2}\ (\ggg\!=\textsf{-sem})$$

$$\frac{x \in g_1 \cdot s}{x \in g_1 \mid g_2 \cdot s}\ (\textsf{left-sem}) \qquad \frac{x \in g_2 \cdot s}{x \in g_1 \mid g_2 \cdot s}\ (\textsf{right-sem})$$

Figure 1. Alternative presentation of the definition of the semantics of grammars (_∈_·_).

The proof proceeds by induction on the structure of the parse tree: Assume that we have a parse tree $q : x \in \mathit{fail} \cdot s$. Our goal is to show that the empty type is inhabited. The grammar *fail* can be unfolded to *fail* | *fail*, so we get $q : x \in \mathit{fail} \mid \mathit{fail} \cdot s$. By case analysis we get that q either has the form left-sem p, with $p : x \in \mathit{fail} \cdot s$, or right-sem p, also with $p : x \in \mathit{fail} \cdot s$. In both cases the inductive hypothesis (*fail-empty p*) gives us an inhabitant of the empty type.

Given the basic grammar combinators above we can define a number of derived ones—taken more or less directly from the world of parser combinators—including the following ones for mapping and sequencing:

```
_<$>_ : (A → B) → Grammar A → Grammar B
f <$> g = g >>= λ x → return (f x)

_<$_ : A → Grammar B → Grammar A
x <$ g = (λ _ → x) <$> g

_⊛_ : Grammar (A → B) → Grammar A → Grammar B
g1 ⊛ g2 = g1 >>= λ f → f <$> g2

_<⊛_ : Grammar A → Grammar B → Grammar A
g1 <⊛ g2 = g1 >>= λ x → x <$ g2

_⊛>_ : Grammar A → Grammar B → Grammar B
g1 ⊛> g2 = g1 >>= λ _ → g2
```

Here f <\$> g is "map": if g generates s and x, then f <\$> g generates s and $f\,x$. The application x <\$ g generates exactly the same strings as g, but always returns x. If g_1 generates s_1 and f, and g_2 generates s_2 and x, then $g_1 \circledast g_2$ generates $s_1 +\!\!+\ s_2$ and $f\,x$. The combinators _<⊛_ and _⊛>_ are variants of _⊛_ that discard the second and first argument's result, respectively. All these operators should be parsed left-associatively. For instance, f <\$> $g_1 \circledast g_2$ should be parsed as (f <\$> g_1) $\circledast\, g_2$.

We can also define the Kleene star and plus operators, which are taken to bind tighter than the mapping and sequencing operators above; these operators are in turn taken to bind tighter than _|_:

```
mutual
  _⋆ : Grammar A → Grammar (List A)
  g ⋆ = return []
      | (uncurry _::_) <$> g +

  _+ : Grammar A → Grammar (List⁺ A)
  g + = (_::_) <$> g ⊛ g ⋆
```

Here *List A* stands for lists of finite length containing *A*s, and $\mathit{List}^+\ A$ stands for *non-empty* lists: $\mathit{List}^+\ A = A \times \mathit{List}\ A$. I overload cons, _::_, so that its second argument and its result can both be either regular or non-empty lists. The application *uncurry* _::_ above has type $\mathit{List}^+\ A \to \mathit{List}\ A$.

It is also possible to define combinators with dependent types. The function *True* maps booleans to types:

```
True : Bool → Set
True true  = Unit
True false = Empty
```

True true is the unit type (with a single inhabitant tt), and *True* false is the empty type. *True* is used in the types of *if-true* and *sat*:

```
if-true : (b : Bool) → Grammar (True b)
if-true true  = return tt
if-true false = fail

sat : (p : Char → Bool) →
      Grammar (Σ Char (λ t → True (p t)))
sat p = token >>= λ t →
          (λ x → (t, x)) <$> if-true (p t)
```

The grammar *if-true b* stands for the empty string if b is true, and otherwise the empty language. The *sat* combinator takes a boolean-valued predicate p as argument, and accepts single tokens t for which $p\ t$ is true. The result of *sat p* is a pair consisting of a token t plus a proof of *True* ($p\ t$), witnessing the truth of $p\ t$. (If B has type $A \to \mathit{Set}$, then $\Sigma\ A\ B$ consists of pairs (x, y) where $x : A$ and $y : B\ x$.)

We can use *sat* to define a combinator for a given token:

```
tok : Char → Grammar Char
tok t = t <$ sat (λ t' → t == t')
```

Here the witness of equality is thrown away. (See Section 4.4 for an example where the witness is retained.) Using *tok* we can define a combinator for whitespace, for simplicity taken to mean "space or newline":

```
whitespace : Grammar Char
whitespace = tok ' ' | tok '\n'
```

We can also define *string s*, a grammar for the string *s*:

```
string : String → Grammar String
string []       = return []
string (t :: s) = (_::_) <$> tok t ⊛ string s
```

3. Pretty-Printers

The type of pretty-printer documents is defined (inductively) in the following way:[3]

```
data Doc : Grammar A → A → Set₁ where
  _◇_   : Doc g1 x → Doc (g2 x) y →
          Doc (g1 >>= g2) y
  text  : Doc (string s) s
  line  : Doc (tt <$ whitespace +) tt
  group : Doc g x → Doc g x
  nest  : ℕ → Doc g x → Doc g x
  emb   : {x1 : A1} → {x2 : A2} →
          (∀ {s} → x1 ∈ g1 · s → x2 ∈ g2 · s) →
          Doc g1 x1 → Doc g2 x2
```

As mentioned in the introduction a pretty-printer for a grammar g : *Grammar A* is a function that transforms all values $x : A$ to documents matching g and x:

```
Pretty-printer : Grammar A → Set₁
Pretty-printer g = ∀ x → Doc g x
```

[3] The notation $\forall \{x\} \to T$ means the same as $\{x : A\} \to T$, except that Agda is asked to try to infer the domain type A. A similar notation, $\forall\ x \to T$, can be used for explicit arguments, and n-ary variants like $\forall \{x\ y\ z\} \to T$ are also available.

The first five document constructors are taken from Wadler (2003). If rendering is implemented in the same way as in Wadler's work, then these combinators have the following meanings:

- $\_\diamond\_$: Sequencing. The second document's text is placed directly after the first document's text.
- text: A concrete string. Note that the string s is not an explicit argument to this combinator: in many cases the string can be inferred from the context. (It can be given explicitly using the notation $\mathsf{text}\ \{s = \ldots\}$.)

  Wadler adopts the convention that s does not contain newline characters. I could enforce this invariant using the type system, but I have refrained from doing so, because the invariant is not needed to prove that the renderers in Section 5 produce grammatically correct strings.
- line: A newline character plus some indentation consisting of zero or more space characters. The amount of indentation is specified by nest combinators (the default is zero).

  The grammar used for this combinator is $\mathsf{tt} <\$\ \mathit{whitespace}\ +$, where tt is the sole inhabitant of the unit type. I do not use the grammar $\mathit{whitespace}\ +$, because then the result (s in $\mathit{Doc}\ (\mathit{whitespace}\ +)\ s$) would have to be a fixed non-empty string, containing a predefined amount of whitespace.

  Note that the grammar $\mathsf{tt} <\$\ \mathit{whitespace}\ +$ is more liberal than "newline plus indentation". The reason is partly that outer group combinators can change the meaning of line combinators, but also that I want to support multiple rendering algorithms (see Section 5).
- group: The document $\mathsf{group}\ d$ is rendered either as d, or as d with all line combinators replaced by single spaces, depending on what is "best" (see Section 5.2).
- nest: The document $\mathsf{nest}\ i\ d$ behaves as d, except that if a line combinator in d is rendered as a line break, then the following line is indented i steps more.

The final combinator, not present in Wadler's library, is emb. This combinator's first argument is a proof that transforms parse trees: "for all strings s, if s and x_1 are generated by g_1, then s and x_2 are generated by g_2". The combinator can be used to "embed" one grammar-result pair (g_1, x_1) into another, (g_2, x_2), and is included so that grammar constructions that are not supported by the other combinators can be handled. For instance, here are two combinators that can be used when a grammar contains a choice:

$$
\begin{aligned}
&\mathit{left} : \mathit{Doc}\ g_1\ x \to \mathit{Doc}\ (g_1 \mid g_2)\ x\\
&\mathit{left}\ d = \mathsf{emb\ left\text{-}sem}\ d
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{right} : \mathit{Doc}\ g_2\ x \to \mathit{Doc}\ (g_1 \mid g_2)\ x\\
&\mathit{right}\ d = \mathsf{emb\ right\text{-}sem}\ d
\end{aligned}
$$

The renderers in Section 5 ignore emb constructors (except for in their correctness proofs). To avoid long chains of emb constructors one can use the following smart constructor instead of emb:

$$
\begin{aligned}
\mathit{embed} :\ & \{x_1 : A_1\} \to \{x_2 : A_2\} \to\\
& (\forall\ \{s\} \to x_1 \in g_1 \cdot s \to x_2 \in g_2 \cdot s) \to\\
& \mathit{Doc}\ g_1\ x_1 \to \mathit{Doc}\ g_2\ x_2\\
\mathit{embed}\ f\ (\mathsf{emb}\ g\ d) &= \mathsf{emb}\ (f \circ g)\ d\\
\mathit{embed}\ f\ d &= \mathsf{emb}\ f\ d
\end{aligned}
$$

We can also define document combinators corresponding to the mapping and sequencing combinators introduced in Section 2. I overload the names, and omit proofs—I write *embed* instead of *embed proof*. Note that some arguments are omitted (made implicit), because they can often be inferred from the context. For instance, $<\$>\_$ only takes one (explicit) argument:

$$
\begin{aligned}
&<\$>\_ : \mathit{Doc}\ g\ x \to \mathit{Doc}\ (f <\$> g)\ (f\ x)\\
&<\$>\ d = \mathit{embed}\ d
\end{aligned}
$$

$$
\begin{aligned}
&<\$\_ : \mathit{Doc}\ g\ y \to \mathit{Doc}\ (x <\$\ g)\ x\\
&<\$\ d = \mathit{embed}\ d
\end{aligned}
$$

$$
\begin{aligned}
&\_\circledast\_ : \mathit{Doc}\ g_1\ f \to \mathit{Doc}\ g_2\ x \to \mathit{Doc}\ (g_1 \circledast g_2)\ (f\ x)\\
&d_1 \circledast d_2 = d_1 \diamond (<\$>\ d_2)
\end{aligned}
$$

$$
\begin{aligned}
&\_{<}\!\circledast\_ : \mathit{Doc}\ g_1\ x \to \mathit{Doc}\ g_2\ y \to \mathit{Doc}\ (g_1 <\!\circledast\ g_2)\ x\\
&d_1 <\!\circledast\ d_2 = d_1 \diamond (<\$\ d_2)
\end{aligned}
$$

$$
\begin{aligned}
&\_\circledast\!>\_ : \mathit{Doc}\ g_1\ x \to \mathit{Doc}\ g_2\ y \to \mathit{Doc}\ (g_1 \circledast\!> g_2)\ y\\
&d_1 \circledast\!> d_2 = d_1 \diamond d_2
\end{aligned}
$$

I omit most of the embedding proofs, because I do not think that the proof terms are very interesting. However, it may be instructive to see a couple of concrete proofs. Here is a more complete definition of $<\$>\_$:

$$
\begin{aligned}
&<\$>\_ : \mathit{Doc}\ g\ x \to \mathit{Doc}\ (f <\$> g)\ (f\ x)\\
&<\$>\ d = \mathit{embed}\ (\lambda\ p \to \mathit{cast}\ \mathit{right\text{-}identity}\\
&\qquad\qquad\qquad\qquad (\gg\!=\!\mathsf{\text{-}sem}\ p\ \mathsf{return\text{-}sem}))\ d
\end{aligned}
$$

The type of $\gg\!=\!\mathsf{\text{-}sem}\ p\ \mathsf{return\text{-}sem}$ is not $f\ x \in f <\$> g \cdot s$, but rather $f\ x \in f <\$> g \cdot s +\!\!+ [\,]$, so *cast* and *right-identity* are used to correct the string index:

$$
\begin{aligned}
\mathit{cast} &: s_1 \equiv s_2 \to x \in g \cdot s_1 \to x \in g \cdot s_2\\
\mathit{right\text{-}identity} &: s +\!\!+ [\,] \equiv s
\end{aligned}
$$

Here $x \equiv y$ is a type of proofs of equalities between x and y.

The proof above does not involve pattern matching on the parse tree p. However, pattern matching is sometimes necessary. Let us consider the definition of the combinator *nil*, a combinator that produces the empty string if Wadler's rendering algorithm is used. This definition includes a proof that does use pattern matching:

$$
\begin{aligned}
&\mathit{nil} : \mathit{Doc}\ (\mathsf{return}\ x)\ x\\
&\mathit{nil} = \mathit{embed}\ \mathit{proof}\ \mathsf{text}\\
&\quad \textbf{where}\\
&\quad \mathit{proof} : [\,] \in \mathit{string}\ [\,] \cdot s \to x \in \mathsf{return}\ x \cdot s\\
&\quad \mathit{proof}\ \mathsf{return\text{-}sem} = \mathsf{return\text{-}sem}
\end{aligned}
$$

Note that text's string argument—the empty string—is inferred automatically. The proof can be read as follows: We should prove $x \in \mathsf{return}\ x \cdot s$, given $[\,] \in \mathit{string}\ [\,] \cdot s$. Note that $\mathit{string}\ [\,]$ reduces to $\mathsf{return}\ [\,]$. By exhaustive case analysis we see that $[\,] \in \mathit{string}\ [\,] \cdot s$ is true iff s is $[\,]$, so it suffices to prove $x \in \mathsf{return}\ x \cdot [\,]$, which follows by $\mathsf{return\text{-}sem}$.

More pretty-printing combinators will be introduced below.

4. Examples

Let us now consider some examples. Note that my focus is not on the *design* of a pretty-printer (where to use group, nest and line, how to achieve pretty output, etc.), but rather on the specifics of using the strongly typed combinators introduced in this paper.

4.1 Boolean Literals

The following is a grammar for boolean literals:

$$
\begin{aligned}
&\mathit{bool} : \mathit{Grammar}\ \mathit{Bool}\\
&\mathit{bool} = \mathsf{true} <\$\ \mathit{string}\ \texttt{"true"}\\
&\qquad\ \mid \mathsf{false} <\$\ \mathit{string}\ \texttt{"false"}
\end{aligned}
$$

There are only two valid strings, `"true"` (corresponding to the value true), and `"false"` (corresponding to the value false).

In order to illustrate the types at play I will give a detailed, step-by-step description of how a pretty-printer for the grammar *bool* can be constructed interactively in Agda. I start by pattern matching on the boolean (note that *Pretty-printer bool* unfolds to $(b : Bool) \to Doc\ bool\ b$):

```
bool_P : Pretty-printer bool
bool_P true  = ?
bool_P false = ?
```

The question marks are *goals* (or *holes*) that have not yet been replaced by concrete terms. I will focus on the first goal. Agda states that the type of this goal is *Doc bool* true, i.e., the question mark should be replaced by something of this type. The value true is generated by the grammar's left branch, so let us refine the right-hand side using *left*:

```
bool_P true = left ?
```

The new goal type is *Doc* (true <$ *string* "true") true, so I choose to use the combinator <$_:

```
bool_P true = left (<$ ?)
```

The grammar combinator _<$_ discards its second argument's result, and I have not specified what this result should be, so we get the goal type *Doc* (*string* "true") s for some unconstrained meta-variable s. If the question mark is replaced by text, then s is unified with "true", leaving us with a complete right-hand side:

```
bool_P true = left (<$ text)
```

Note that there is no need to specify the string used by text: it is inferred by the type checker, and an attempt to specify a concrete string distinct from "true" would lead to a type error.

The other clause can be completed in a similar way:

```
bool_P : Pretty-printer bool
bool_P true  = left  (<$ text)
bool_P false = right (<$ text)
```

The grammar *bool* is not very flexible: there is only one valid parse tree for true, and similarly for false. The grammar in the next example gives more freedom to pretty-printer implementors.

4.2 Expressions

The following example is based on one discussed by Matsuda and Wang (2013). Expressions are defined inductively as follows:

```
data Expr : Set where
  one : Expr
  sub : Expr → Expr → Expr
```

An expression is either a one or a subtraction.

We can define the following grammar for expressions:

```
mutual
  expr : Grammar Expr
  expr = term
       | sub <$> expr <⊛ whitespace ⋆ <⊛ string "-"
                      <⊛ whitespace ⋆ ⊛ term

  term : Grammar Expr
  term = one <$ string "1"
       | string "(" ⊛> whitespace ⋆ ⊛> expr
                    <⊛ whitespace ⋆ <⊛ string ")"
```

Here *expr* stands for terms and subtractions of the form "expression − term", whereas *term* stands for literal ones and parenthesised expressions.

The grammar contains four textual occurrences of *whitespace* ⋆, and implementors of pretty-printers must choose how to handle these. Matsuda and Wang do not use any whitespace right after an opening parenthesis, or right before a closing one; they always use a single space character after a minus sign; and they use the line combinator to handle the last occurrence of *whitespace* ⋆.

In the present setting the line combinator's type is not quite right: its grammar is tt <$ *whitespace* +, not *whitespace* ⋆. To address this problem I introduce two new reusable combinators:

```
line_⋆ : Doc (tt <$ whitespace ⋆) tt
line_⋆ = embed line

_<⊛tt_ : Doc g_1 x → Doc (tt <$ g_2) tt →
         Doc (g_1 <⊛ g_2) x
d_1 <⊛tt d_2 = embed (d_1 <⊛ d_2)
```

I also introduce some combinators that, when Wadler's rendering algorithm is used, produce a single space character (*space*) or an empty string ($nil_\star$):

```
space : Doc (whitespace ⋆) " "
space = embed (text {s = " "})

nil_⋆ : Doc (g ⋆) [ ]
nil_⋆ = left nil
```

These combinators can be used to define a pretty-printer that matches Matsuda and Wang's:

```
one_D : Doc term one
one_D = left (<$ text)

mutual
  expr_P : Pretty-printer expr
  expr_P one          = left one_D
  expr_P (sub e_1 e_2) =
    group (right (<$>  expr_P e_1
                 <⊛tt nest 2 line_⋆
                 <⊛   text
                 <⊛   space
                 ⊛    nest 2 (term_P e_2)))

  term_P : Pretty-printer term
  term_P one   = one_D
  term_P e     =
    right (text ⊛> nil_⋆ ⊛> expr_P e <⊛ nil_⋆ <⊛ text)
```

Note the use of group and nest. Note also that *embed* is not used directly in the definition of the pretty-printer, only in reusable combinators.

If $expr_P$ is used to pretty-print

```
sub (sub one one) (sub one one),
```

using the implementation of Wadler's rendering algorithm described in Section 5, then the following outputs can be obtained (depending on the line width):

```
1 - 1 - (1 - 1)  |  1 - 1          |  1 - 1
                 |    - (1 - 1)    |    - (1
                 |                 |       - 1)
```

Matsuda and Wang list exactly the same example outputs.

For comparison I also include Matsuda and Wang's pretty-printer (I have adapted the notation to that used in the present paper, and have made use of some enhancements—described later in Matsuda and Wang's paper—to reduce code duplication):[4]

[4] The overlapping clauses in the definition of $term_P'$ are in principle problematic, as Matsuda and Wang state that the pretty-printing semantics of overlapping clauses is non-deterministic. However, the implementation that accompanies their paper uses a first-match semantics. One may believe that one can avoid overlapping patterns by replacing the final clause of $term_P'$ by $term_P'$ (sub e_1 e_2) = *par* ($expr_P$ (sub e_1 e_2)), but in Matsuda and Wang's language the argument to $expr_P$ in the right-hand side has to be a variable (to ensure that the pretty-printer can be turned into a parser).

$$\begin{aligned}
&\mathit{nil} &&= \mathsf{text}\ \texttt{""} <\!+\ \mathit{space}\\
&\mathit{space} &&= (\mathsf{text}\ \texttt{"\ "} <\!+\ \mathsf{text}\ \texttt{"\textbackslash n"}) \diamond \mathit{nil}\\
&\mathit{space}' &&= \mathit{space} <\!+\ \mathsf{text}\ \texttt{""}\\
&\mathit{line}' &&= \mathsf{line} \quad <\!+\ \mathsf{text}\ \texttt{""}
\end{aligned}$$

$$\begin{aligned}
&\mathit{many\text{-}pars}\ d &&= d <\!+\ \mathit{par}\ (\mathit{many\text{-}pars}\ d)\\
&\mathit{par}\ d &&= \mathsf{text}\ \texttt{"("} \diamond \mathit{nil} \diamond d \diamond \mathit{nil} \diamond \mathsf{text}\ \texttt{")"}
\end{aligned}$$

$$\mathit{expr}_{\mathrm{P}}\ x = \mathit{many\text{-}pars}\ (\mathit{expr}_{\mathrm{P}}'\ x)$$

$$\begin{aligned}
&\mathit{expr}_{\mathrm{P}}'\ \mathsf{one} &&= \mathsf{text}\ \texttt{"1"}\\
&\mathit{expr}_{\mathrm{P}}'\ (\mathsf{sub}\ e_1\ e_2) &&= \mathsf{group}\ (\mathit{expr}_{\mathrm{P}}\ e_1 \diamond\\
& && \qquad \mathsf{nest}\ 2\ (\mathit{line}' \quad \diamond\\
& && \qquad\qquad \mathsf{text}\ \texttt{"-"} \diamond\\
& && \qquad\qquad \mathit{space}' \quad \diamond\\
& && \qquad\qquad \mathit{term}_{\mathrm{P}}\ e_2))
\end{aligned}$$

$$\mathit{term}_{\mathrm{P}}\ x = \mathit{many\text{-}pars}\ (\mathit{term}_{\mathrm{P}}'\ x)$$

$$\begin{aligned}
&\mathit{term}_{\mathrm{P}}'\ \mathsf{one} &&= \mathsf{text}\ \texttt{"1"}\\
&\mathit{term}_{\mathrm{P}}'\ e &&= \mathit{par}\ (\mathit{expr}_{\mathrm{P}}\ e)
\end{aligned}$$

Note that this definition contains both a specification of a grammar, and a specification of a pretty-printer. Recall that $p <\!+\ q$ means "pretty-print according to p, but be ready to parse also according to q". This means, for instance, that *nil* renders as the empty string, but the corresponding grammar accepts arbitrary sequences of whitespace. (The grammar corresponding to the line combinator also accepts arbitrary sequences of whitespace.)

4.3 Expressions, Take Two

The grammar used for expressions above contains four textual occurrences of $\mathit{whitespace}\ \star$. To avoid this kind of clutter one can use grammar combinators that "swallow" trailing whitespace (Hutton and Meijer 1998). The grammar *symbol s* stands for the string *s* plus trailing whitespace:

$$\begin{aligned}
&\mathit{symbol} : \mathit{String} \to \mathit{Grammar\ String}\\
&\mathit{symbol}\ s = \mathit{string}\ s <\!\circledast\ \mathit{whitespace}\ \star
\end{aligned}$$

The following expression grammar uses *symbol* instead of *string* and $\mathit{whitespace}\ \star$:

$$\begin{aligned}
&\textbf{mutual}\\
&\quad \mathit{expr} : \mathit{Grammar\ Expr}\\
&\quad \mathit{expr} = \mathit{term}\\
&\quad\qquad \mid\ \mathsf{sub} <\!\$\!>\ \mathit{expr} <\!\circledast\ \mathit{symbol}\ \texttt{"-"} \circledast \mathit{term}\\
&\quad \mathit{term} : \mathit{Grammar\ Expr}\\
&\quad \mathit{term} = \mathsf{one} <\!\$\ \mathit{symbol}\ \texttt{"1"}\\
&\quad\qquad \mid\ \mathit{symbol}\ \texttt{"("} \circledast\!>\ \mathit{expr} <\!\circledast\ \mathit{symbol}\ \texttt{")"}
\end{aligned}$$

This grammar is not quite equivalent to the one in Section 4.2, as that one does not accept final trailing whitespace (as in `"1 "`).

Let us now define a pretty-printer for the updated grammar *expr*. I first introduce some document combinators that can be used to handle the grammar *symbol s*:

$$\begin{aligned}
&\mathit{symbol} : \mathit{Doc}\ (\mathit{symbol}\ s)\ s\\
&\mathit{symbol} = \mathsf{text} <\!\circledast\ \mathit{nil}_{\star}
\end{aligned}$$

$$\begin{aligned}
&\mathit{symbol\text{-}space} : \mathit{Doc}\ (\mathit{symbol}\ s)\ s\\
&\mathit{symbol\text{-}space} = \mathsf{text} <\!\circledast\ \mathit{space}
\end{aligned}$$

Using these combinators I can construct the following incomplete pretty-printer:

$$\begin{aligned}
&\mathit{one}_{\mathrm{D}} : \mathit{Doc\ term}\ \mathsf{one}\\
&\mathit{one}_{\mathrm{D}} = \mathit{left}\ (<\!\$\ \mathit{symbol})
\end{aligned}$$

$$\begin{aligned}
&\textbf{mutual}\\
&\quad \mathit{expr}_{\mathrm{P}} : \mathit{Pretty\text{-}printer\ expr}\\
&\quad \mathit{expr}_{\mathrm{P}}\ \mathsf{one} \qquad\quad = \mathit{left}\ \mathit{one}_{\mathrm{D}}\\
&\quad \mathit{expr}_{\mathrm{P}}\ (\mathsf{sub}\ e_1\ e_2) =\\
&\qquad \mathsf{group}\ (\mathit{right}\ (<\!\$\!>\ ?\\
&\qquad\qquad\qquad\qquad <\!\circledast\ \ \mathit{symbol\text{-}space}\\
&\qquad\qquad\qquad\qquad \circledast\ \ \mathsf{nest}\ 2\ (\mathit{term}_{\mathrm{P}}\ e_2)))\\
&\quad \mathit{term}_{\mathrm{P}} : \mathit{Pretty\text{-}printer\ term}\\
&\quad \mathit{term}_{\mathrm{P}}\ \mathsf{one} \qquad = \mathit{one}_{\mathrm{D}}\\
&\quad \mathit{term}_{\mathrm{P}}\ e \qquad\quad = \mathit{right}\ (\mathit{symbol} \circledast\!>\ \mathit{expr}_{\mathrm{P}}\ e <\!\circledast\ \mathit{symbol})
\end{aligned}$$

What should the question mark be replaced with? The previous implementation of $\mathit{expr}_{\mathrm{P}}$ contains the subexpression

$$<\!\$\!>\ \mathit{expr}_{\mathrm{P}}\ e_1 <\!\circledast_{\mathtt{tt}}\ \mathsf{nest}\ 2\ \mathit{line}_{\star}.$$

We can use something similar here,

$$\mathit{embed}\ \ (\mathit{expr}_{\mathrm{P}}\ e_1 <\!\circledast_{\mathtt{tt}}\ \mathsf{nest}\ 2\ \mathit{line}_{\star}),$$

provided that we can prove the following statement:

$$\begin{aligned}
\forall \{s\} &\to e_1 \in \mathit{expr} <\!\circledast\ \mathit{whitespace}\ \star \cdot s\\
&\to e_1 \in \mathit{expr} \qquad\qquad\qquad\quad \cdot s
\end{aligned}$$

It is not very hard to prove this statement manually. However, I think that this kind of proof is rather tedious. Fortunately we can let the computer prove the statement for us, by writing a program that analyses the grammar *expr* and produces a proof.

The grammar data type introduced in Section 2 is quite tricky to analyse programmatically, partly because bind's second argument is a function, and partly because grammars are potentially infinite (it is for instance impossible to check if a grammar has the form $\mathit{whitespace}\ \star$). These problems could presumably be circumvented through the use of meta-programming techniques, using which one gets access to the grammars' source code. Then one could identify unproblematic uses of bind such as those in $\_<\!\$\!>\_$ and $\_\circledast\_$, as well as regular recursion such as that in $\_\star$ and context-free grammars. Another option is to rule out problematic constructions entirely by switching to a different grammar type, for instance a suitable representation of context-free grammars (with semantic actions).

I have chosen to use a different approach, that does not limit expressiveness, and that does not require that the host language has support for meta-programming. As mentioned in Section 2 the accompanying code contains a variant of the *Grammar* type with more constructors. Some of the extra constructors (corresponding to *tok*, $\_<\!\$\!>\_$, $\_<\!\$\_$, $\_\circledast\_$, $\_<\!\circledast\_$ and $\_\circledast\!>\_$) make it possible to avoid certain uses of bind,[5] and the remaining extra constructors (corresponding to *fail* and $\_\star$) make it possible to avoid certain uses of corecursion.

Using this extended grammar data type I have implemented a simple, heuristic procedure that tries to prove statements like the one above:

$$\begin{aligned}
\mathit{trailing\text{-}whitespace} : \mathbb{N} \to (g : \mathit{Grammar}\ A) \to&\\
\mathit{Maybe}\ (\mathit{Trailing\text{-}whitespace}\ g)&
\end{aligned}$$

The first argument is a natural number. The procedure is implemented by structural recursion on this number—recursion on the structure of a potentially infinite grammar could lead to non-termination. The predicate *Trailing-whitespace* is satisfied by a grammar *g* if the grammar can swallow trailing whitespace:

[5] Perhaps it is not necessary to include all of these combinators as primitives, but as mentioned in Section 2 the extra constructors can make it easier to write guarded definitions.

```
Trailing-whitespace : Grammar A → Set₁
Trailing-whitespace g =
  ∀ {x s} → x ∈ g <⊛ whitespace ⋆ · s → x ∈ g · s
```

Note that *Trailing-whitespace expr* is a slightly more general version of the statement that should be proved.

There are many ways to implement *trailing-whitespace*, and I do not think the details are central to this paper, so no implementation is included here. The main point is that, assuming that *expr* is implemented using the extra grammar constructors,

```
trailing-whitespace 6 expr
```

returns just *proof*, where *proof* has type *Trailing-whitespace expr*. Thus there is no need to prove this statement manually.

The following reusable combinator can be used to add nest *i* $line_\star$ to the end of a document for which *trailing-whitespace* succeeds:

```
final-line :
  (n : ℕ) →
  {trailing : True (is-just (trailing-whitespace n g))} →
  Doc g x → ℕ → Doc g x
final-line _ d i = embed (d <⊛tt nest i line⋆)
```

The omitted embedding proof makes use of the implicit argument *trailing*. If *trailing-whitespace n g* evaluates to just *proof*, then the type of *trailing* is *Unit* (because *is-just* (just *proof*) is true). Omitted arguments of type *Unit* are automatically inferred to be tt, so in this case *trailing* does not need to be given explicitly. We can thus complete the definition of the pretty-printer in the following way:

```
exprP (sub e₁ e₂) =
  group (right (<$> final-line 6 (exprP e₁) 2
                 <⊛  symbol-space
                 ⊛   nest 2 (termP e₂)))
```

If we compare the examples in Sections 4.2 and 4.3, then we see that the grammar in 4.3 is more compact, but it seems fair to say that the pretty-printer is more complicated. The main complication is that the pretty-printer deviates from the grammar's structure, thus necessitating an embedding proof (going from *expr* <⊛ *whitespace* ⋆ to *expr*). In general one may find that it is easier to implement a pretty-printer if the grammar is defined in such a way that the pretty-printer can follow the grammar's structure, potentially at the cost of a less natural or more complicated grammar. This can be contrasted with Matsuda and Wang's approach, in which the pretty-printer *must* (by construction) follow the grammar's structure. The approach presented in this paper is thus more flexible.

4.4 Identifiers

Consider the following grammar for identifiers consisting of one or more lower-case letters:

```
identifier : Grammar (List⁺ Char)
identifier = (fst <$> sat is-lower) +
```

This grammar is problematic: it is impossible to implement a pretty-printer for *identifier*. The problem is that the result type, *List*$^+$ *Char*, contains junk: there are non-empty strings that do not consist solely of lower-case letters, and a pretty-printer $identifier_P$: *Pretty-printer identifier* must be able to handle such strings. For instance, $identifier_P$ `['A']` must return a document of type *Doc identifier* `['A']`, and this type is empty.

Fortunately there is a simple workaround—make the type more precise:

```
Identifier : Set
Identifier = List⁺ (Σ Char (λ t → True (is-lower t)))
identifier : Grammar Identifier
identifier = sat is-lower +
```

Identifier stands for non-empty lists of pairs, where each pair consists of a token *t* and a proof of *True* (*is-lower t*). Note that *sat is-lower* returns this kind of pair.

It is easy to implement a pretty-printer corresponding to the grammar *sat p*:

```
token : Doc token t
token {t = t} = embed (text {s = [t]})
if-true : (b : Bool) → Pretty-printer (if-true b)
if-true true _ = nil
if-true false ()    -- Impossible case.
sat : (p : Char → Bool) → Pretty-printer (sat p)
sat p (t, proof) = token ◇ (<$> if-true (p t) proof)
```

It is also straightforward to implement "mapping" combinators corresponding to the Kleene star and plus operators:

```
mutual
  map⋆ : Pretty-printer g → Pretty-printer (g ⋆)
  map⋆ p []        = nil⋆
  map⋆ p (x :: xs) = embed (map+ p (x :: xs))
  map+ : Pretty-printer g → Pretty-printer (g +)
  map+ p (x :: xs) = <$> p x ⊛ map⋆ p xs
```

With the functions above in place it is easy to implement a pretty-printer for identifiers:

```
identifierP : Pretty-printer identifier
identifierP = map+ (sat is-lower)
```

4.5 Other Examples

The accompanying code includes some larger examples:

- A pretty-printer for expressions, parametrised by a collection of operators, each with a given precedence and associativity.
- A pretty-printer for a kind of simplified XML documents, based on a pretty-printer described by Wadler (2003). The simplified XML grammar uses the bind operator to define the syntax of matching opening and closing tags. The pretty-printer makes use of an additional primitive pretty-printing combinator, fill, which is based on a combinator due to Peyton Jones (1996). Wadler's version of the combinator is described as "put[ting] a space between two documents when this leads to reasonable layout, and a newline otherwise" (2003). My version of the combinator has the following type:

  ```
  fill : Docs g xs → Doc (g sep-by (whitespace +)) xs
  ```

  Here *Docs g xs* stands for a list containing one or more *g*-indexed documents, and *g sep-by sep* stands for one or more occurrences of *g*, separated by *sep*:

  ```
  data Docs (g : Grammar A) : List⁺ A → Set₁ where
    one  : Doc g x              → Docs g (x :: [])
    cons : Doc g x → Docs g xs → Docs g (x :: xs)
  _sep-by_ : Grammar A → Grammar B →
             Grammar (List⁺ A)
  g sep-by sep = (_::_) <$> g ⊛ (sep ⊛> g) ⋆
  ```

It may be worth noting that these two examples do not use *embed* at all: manual proofs are relegated to (more or less) reusable library combinators. However, some library combinators not mentioned above were introduced as part of the implementation of these examples. I do not claim that the current library's set of combinators is sufficient to avoid every use of *embed*.

5. Renderers

As mentioned in the introduction a renderer consists of two parts: a function

```
render : Doc g x → String
```

that maps documents to strings, and a correctness proof:

```
parsable : (d : Doc g x) → x ∈ g · render d
```

The *parsable* property can be used to prove a round-tripping property for unambiguous grammars.

I define unambiguity in the following way:

```
Unambiguous : Grammar A → Set₁
Unambiguous g =
  ∀ {s x y} → x ∈ g · s → y ∈ g · s → x ≡ y
```

A grammar g is unambiguous if, whenever the string s and the result x are generated by g, and also s and y are generated by g, then x is equal to y. This is a weak form of unambiguity: I do not require the two parse trees to be equal.

I also define a type of parsers that are guaranteed to be correct:

```
Parser : ∀ {A} → Grammar A → Set₁
Parser {A = A} g = ∀ s → Dec (Σ A (λ x → x ∈ g · s))
```

Dec X ("decided *X*") has two constructors,

```
yes : X → Dec X
```

and

```
no : (X → Empty) → Dec X.
```

A parser must thus either return a pair consisting of a result and a corresponding parse tree, or return a proof showing that there is no such pair.

Given the definitions above the following round-tripping property can be formulated:

```
Unambiguous g →
(parse : Parser g) →
(pretty : Pretty-printer g) →
∀ x → Σ (x ∈ g · render (pretty x))
          (λ p → parse (render (pretty x)) ≡ yes (x, p))
```

This property is easy to prove using *parsable*. (The precondition *Unambiguous g* can be weakened: the grammar only needs to be unambiguous for the string *render* (*pretty x*).)

Another property can also be proved. Assume that *render* ignores top-level emb constructors, i.e., assume that the string *render* (emb *f d*) is equal to *render d*. Then, for every grammatically correct string, there is a document that renders to that string:

```
x ∈ g · s → Σ (Doc g x) (λ d → render d ≡ s)
```

(Both renderers below ignore top-level emb constructors.) This is not a very deep property—my proof returns the document *embed* text, with a suitable embedding proof. However, the property gives a kind of weak guarantee that the document interface is not too limited.

Let us now consider two different renderers.

5.1 An Ugly-Renderer

The following "ugly-renderer" renders each occurrence of line as a single space character, and is included in order to illustrate that the document interface does not require the use of Wadler's rendering algorithm:

```
render : Doc g x → String
render (d₁ ◇ d₂)       = render d₁ ++ render d₂
render (text {s = s})  = s
render line            = " "
render (group d)       = render d
render (nest _ d)      = render d
render (emb _ d)       = render d
```

Note that group, nest and emb constructors are ignored. (The use of _++_ above can lead to quadratic behaviour, and can, as usual, be replaced by something without this behaviour.)

The correctness proof is very easy. The emb case may be of interest:

```
parsable : (d : Doc g x) → x ∈ g · render d
...
parsable (emb f d) = f (parsable d)
```

5.2 Wadler's Renderer

Let us now turn to Wadler's rendering algorithm (2003). My implementation is close to Wadler's. However, a direct reimplementation would not be accepted by Agda's termination checker. The code below is structurally recursive.

The renderer works in three steps:

1. First a document is converted into a different document type, without group but instead containing a constructor union, which is a kind of binary choice combinator for documents.
2. In the second step the converted document is transformed into a flat "layout". When a union constructor is encountered the two argument documents (along with a continuation) are converted into layouts, and the "best" one is chosen.
3. Finally the layout is turned into a string.

This renderer is intended to be executed (at least partly) lazily.

Layouts A layout is a list of layout elements, text *s* or line *i*:

```
data Layout-element : Set where
  text : String → Layout-element
  line : ℕ      → Layout-element

Layout : Set
Layout = List Layout-element
```

The meaning of text and line is specified by *show-element*; text *s* is mapped to the string *s*, and line *i* is mapped to a newline character followed by *i* space characters.

```
show-element : Layout-element → String
show-element (text s) = s
show-element (line i)  = '\n' :: replicate i ' '
```

The renderer's third step, conversion of layouts to strings, is performed by *show*:

```
show : Layout → String
show = concat ∘ map show-element
```

Document Conversion The new document type is defined inductively as follows (the N subscript stands for "nesting"):

```
data Doc_N : ℕ → Grammar A → A → Set₁ where
  _◇_ : Doc_N i g₁ x → Doc_N i (g₂ x) y →
        Doc_N i (g₁ >>= g₂) y
  text : (s : String) → Doc_N i (string s) s
```

$$
\begin{array}{lll}
\mathsf{line} & : & (i : \mathbb{N}) \to \\
 & & \textbf{let } s = \textit{show-element}\ (\mathsf{line}\ i)\ \textbf{in} \\
 & & \mathit{Doc}_{\mathrm{N}}\ i\ (\mathit{string}\ s)\ s \\
\mathsf{union} & : & \mathit{Doc}_{\mathrm{N}}\ i\ g\ x \to \mathit{Doc}_{\mathrm{N}}\ i\ g\ x \to \mathit{Doc}_{\mathrm{N}}\ i\ g\ x \\
\mathsf{nest} & : & (j : \mathbb{N}) \to \mathit{Doc}_{\mathrm{N}}\ (j+i)\ g\ x \to \mathit{Doc}_{\mathrm{N}}\ i\ g\ x \\
\mathsf{emb} & : & \{x_1 : A_1\} \to \{x_2 : A_2\} \to \\
 & & (\forall \{s\} \to x_1 \in g_1 \cdot s \to x_2 \in g_2 \cdot s) \to \\
 & & \mathit{Doc}_{\mathrm{N}}\ i\ g_1\ x_1 \to \mathit{Doc}_{\mathrm{N}}\ i\ g_2\ x_2
\end{array}
$$

There are four changes, compared to *Doc*:

- The type has an extra natural number index that stands for the current nesting level. This level is modified by nest.
- The text constructor's string argument has been made explicit. (This is a purely cosmetic change.)
- The type of line is more precise: the grammar is

  string (*show-element* (line *i*)),

  where *i* is the nesting level index. Unlike the previous line combinator this one always stands for a newline character plus indentation.
- The constructor group has been replaced by union. Wadler sees documents as representing sets of strings, and union d_1 d_2 represents the union of the strings represented by d_1 and those represented by d_2.

  Wadler also states that union d_1 d_2 should satisfy two invariants: the first is that the sets of strings represented by d_1 and d_2 should be equal, if every occurrence of line is replaced by a single space character; and the second is that, for every string *s* represented by d_1, the first line of *s* should be at least as long as the first line of any string represented by d_2. I do not enforce these invariants using the type system, as they are not needed to prove grammatical correctness. (Furthermore, as discussed at the end of this section, the invariants are not strong enough to prove that the renderer returns the "best" string, for a certain definition of "best".)

Just as in Section 3 I define a smart variant of emb, called *embed*. I also define the following documents; *imprecise-space* stands for a single space character, and *imprecise-line i* for a newline character followed by indentation:

$$
\begin{array}{l}
\textit{imprecise-space} : \mathit{Doc}_{\mathrm{N}}\ i\ (\mathsf{tt} \mathbin{<\$}\ \mathit{whitespace}\ +)\ \mathsf{tt} \\
\textit{imprecise-space} = \mathit{embed}\ (\mathsf{text}\ \texttt{" "}) \\[4pt]
\textit{imprecise-line} : (i : \mathbb{N}) \to \mathit{Doc}_{\mathrm{N}}\ i\ (\mathsf{tt} \mathbin{<\$}\ \mathit{whitespace}\ +)\ \mathsf{tt} \\
\textit{imprecise-line}\ i = \mathit{embed}\ (\mathsf{line}\ i)
\end{array}
$$

The name prefix *imprecise* refers to the fact that the grammar indices are less precise than they could be.

There are two functions that convert from *Doc* to $\mathit{Doc}_{\mathrm{N}}$. The function *flatten* replaces line with *imprecise-space*, and removes group and nest constructors, thus constructing documents that render as a single line (assuming that text's string argument never contains newline characters):

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathit{flatten} : \mathit{Doc}\ g\ x \to \mathit{Doc}_{\mathrm{N}}\ i\ g\ x} \\
\mathit{flatten}\ (d_1 \diamond d_2) & = & \mathit{flatten}\ d_1 \diamond \mathit{flatten}\ d_2 \\
\mathit{flatten}\ \mathsf{text} & = & \mathsf{text}\ \_ \\
\mathit{flatten}\ \mathsf{line} & = & \textit{imprecise-space} \\
\mathit{flatten}\ (\mathsf{group}\ d) & = & \mathit{flatten}\ d \\
\mathit{flatten}\ (\mathsf{nest}\ \_\ d) & = & \mathit{flatten}\ d \\
\mathit{flatten}\ (\mathsf{emb}\ f\ d) & = & \mathit{embed}\ f\ (\mathit{flatten}\ d)
\end{array}
$$

The function *expand* implements the renderer's first step. Recall that group *d* should be rendered either as *d*, or as *d* with all line combinators replaced by single spaces. The *expand* function "expands" groups into unions, replacing group *d* with the union of *flatten d* and *expand d*:

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathit{expand} : \mathit{Doc}\ g\ x \to \mathit{Doc}_{\mathrm{N}}\ i\ g\ x} \\
\mathit{expand}\ (d_1 \diamond d_2) & = & \mathit{expand}\ d_1 \diamond \mathit{expand}\ d_2 \\
\mathit{expand}\ \mathsf{text} & = & \mathsf{text}\ \_ \\
\mathit{expand}\ \mathsf{line} & = & \textit{imprecise-line}\ \_ \\
\mathit{expand}\ (\mathsf{group}\ d) & = & \mathsf{union}\ (\mathit{flatten}\ d)\ (\mathit{expand}\ d) \\
\mathit{expand}\ (\mathsf{nest}\ j\ d) & = & \mathsf{nest}\ j\ (\mathit{expand}\ d) \\
\mathit{expand}\ (\mathsf{emb}\ f\ d) & = & \mathit{embed}\ f\ (\mathit{expand}\ d)
\end{array}
$$

As mentioned in Section 4.5 the accompanying code contains an additional primitive combinator, fill. This combinator is a *Doc* constructor, but there is no corresponding $\mathit{Doc}_{\mathrm{N}}$ constructor: *flatten* and *expand* can be modified to translate fill into uses of the constructors given above.

Choosing the "Best" Layout The renderer's second step is implemented by *best*. This function takes three (explicit) arguments and produces a layout. The first argument is a document, and the third the current column position. The second argument is a continuation: a function from a column position to a layout. The result of *best* is the document's layout, followed by the layout computed by the continuation:

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathit{best} : \mathit{Doc}_{\mathrm{N}}\ i\ g\ x \to (\mathbb{N} \to \mathit{Layout}) \to (\mathbb{N} \to \mathit{Layout})} \\
\mathit{best}\ (d_1 \diamond d_2) & = & \mathit{best}\ d_1 \circ \mathit{best}\ d_2 \\
\mathit{best}\ (\mathsf{text}\ \texttt{""}) & = & \mathit{id} \\
\mathit{best}\ (\mathsf{text}\ s) & = & \lambda\ \kappa\ c \to \mathsf{text}\ s :: \kappa\ (\mathit{length}\ s + c) \\
\mathit{best}\ (\mathsf{line}\ i) & = & \lambda\ \kappa\ \_ \to \mathsf{line}\ i\ :: \kappa\ i \\
\mathit{best}\ (\mathsf{union}\ d_1\ d_2) & = & \lambda\ \kappa\ c \to \mathit{better}\ c\ (\mathit{best}\ d_1\ \kappa\ c) \\
 & & \qquad\qquad\qquad\qquad (\mathit{best}\ d_2\ \kappa\ c) \\
\mathit{best}\ (\mathsf{nest}\ \_\ d) & = & \mathit{best}\ d \\
\mathit{best}\ (\mathsf{emb}\ \_\ d) & = & \mathit{best}\ d
\end{array}
$$

Wadler includes *nil* as a document constructor; the first text case is based on his *nil* case.

Note that in the second text case the continuation is called with a column position computed from the string's length, and in the line case the continuation is called with the indentation as the column position. In the union case the best layout is computed for each document, and then *better* is used to choose the best one:

$$
\begin{array}{l}
\mathit{better} : \mathbb{N} \to \mathit{Layout} \to \mathit{Layout} \to \mathit{Layout} \\
\mathit{better}\ c\ x\ y = \mathit{if}\ \mathit{fits}\ (\mathit{width} - c)\ x\ \mathit{then}\ x\ \mathit{else}\ y
\end{array}
$$

The function *better* uses the line width, *width*, which is a parameter of this renderer. If the first line of the first layout fits in the remaining part of the current line, then this layout is chosen, and otherwise the other one. The function *fits* is used to decide if the first line of a layout has at most a certain number of characters:

$$
\begin{array}{llcl}
\multicolumn{4}{l}{\mathit{fits} : \mathbb{Z} \to \mathit{Layout} \to \mathit{Bool}} \\
\mathit{fits}\ (\mathsf{neg}\ w)\ \_ & & = & \mathsf{false} \\
\mathit{fits}\ w & [\,] & = & \mathsf{true} \\
\mathit{fits}\ w & (\mathsf{text}\ s :: x) & = & \mathit{fits}\ (w - \mathit{length}\ s)\ x \\
\mathit{fits}\ w & (\mathsf{line}\ i\ :: x) & = & \mathsf{true}
\end{array}
$$

The first clause treats the case where the number of characters is negative.

The Renderer Given all the pieces above it is easy to assemble a complete rendering function:

$$
\begin{array}{l}
\mathit{render}_{\mathrm{N}} : \mathit{Doc}_{\mathrm{N}}\ i\ g\ x \to \mathit{String} \\
\mathit{render}_{\mathrm{N}}\ d = \mathit{show}\ (\mathit{best}\ d\ (\lambda\ \_ \to [\,])\ 0) \\[4pt]
\mathit{render} : \mathit{Doc}\ g\ x \to \mathit{String} \\
\mathit{render}\ d = \mathit{render}_{\mathrm{N}}\ (\mathit{expand}\ \{i = 0\}\ d)
\end{array}
$$

The initial continuation just returns an empty layout, and the initial indentation and column position are both 0.

Grammatical Correctness My correctness proof is straightforward. The following lemma about *best* can be proved using recursion on the structure of the document:

$$\begin{array}{l} \textit{best-lemma} : \\ \quad (s : \textit{String}) \to (c : \mathbb{N}) \to (d : \textit{Doc}_{\mathrm{N}}\ i\ g\ x) \to \\ \quad ((s' : \textit{String}) \to (c' : \mathbb{N}) \to \\ \quad\quad x \in g \cdot s' \to \\ \quad\quad y \in g' \cdot s \mathbin{+\!\!+} s' \mathbin{+\!\!+} \textit{show}\ (\kappa\ c')) \to \\ \quad y \in g' \cdot s \mathbin{+\!\!+} \textit{show}\ (\textit{best}\ d\ \kappa\ c) \end{array}$$

The lemma uses continuation-passing style, to match the structure of *best*. The most interesting case is perhaps the one for $\_\diamond\_$, in which the inductive hypothesis is used twice:

$$\begin{array}{l} \textit{best-lemma}\ s\ c\ (d_1 \diamond d_2)\ h = \\ \quad \textit{best-lemma}\ s\ c\ d_1\ (\lambda\ s_1\ c_1\ p_1 \to \\ \quad\quad \textit{cast}\ (\textit{best-lemma}\ (s \mathbin{+\!\!+} s_1)\ c_1\ d_2\ (\lambda\ s_2\ c_2\ p_2 \to \\ \quad\quad\quad \textit{cast}\ (h\ (s_1 \mathbin{+\!\!+} s_2)\ c_2\ (\gg\!\!=\text{-sem}\ p_1\ p_2))))) \end{array}$$

Here I have omitted *cast*'s equality argument.

The lemma has the following corollary:

$$(d : \textit{Doc}_{\mathrm{N}}\ i\ g\ x) \to x \in g \cdot \textit{render}_{\mathrm{N}}\ d$$

The correctness property,

$$(d : \textit{Doc}\ g\ x) \to x \in g \cdot \textit{render}\ d,$$

follows immediately from the corollary, because *expand* preserves the grammar and result indices.

Other Properties Wadler lists a number of algebraic laws that his combinators should satisfy. Let us define document equivalence (for $\textit{Doc}_{\mathrm{N}}$) in the following way:

$$\begin{array}{l} \_\approx\_ : \textit{Doc}_{\mathrm{N}}\ i_1\ g_1\ x_1 \to \textit{Doc}_{\mathrm{N}}\ i_2\ g_2\ x_2 \to \textit{Set} \\ d_1 \approx d_2 = \\ \quad \forall\ \textit{width} \to \textit{render}_{\mathrm{N}}\ \textit{width}\ d_1 \equiv \textit{render}_{\mathrm{N}}\ \textit{width}\ d_2 \end{array}$$

(Here the renderer's *width* parameter has been given explicitly.) One can perhaps imagine other definitions of document equivalence, but if two documents are related by any "reasonable" notion of equivalence, then they should arguably also be related by this one.

With this definition of document equivalence *some* of Wadler's equivalences can be proved. For instance, $\_\diamond\_$ distributes from the right over union:[6]

$$\text{union}\ d_1\ d_2 \diamond d_3 \approx \text{union}\ (d_1 \diamond d_3)\ (d_2 \diamond d_3)$$

However, the following equivalence, also given by Wadler, cannot be proved:

$$d_1 \diamond \text{union}\ d_2\ d_3 \approx \text{union}\ (d_1 \diamond d_2)\ (d_1 \diamond d_3)$$

If we let d_1 and d_3 be *imprecise-line* 0, and d_2 be *imprecise-space*, then

$$\textit{render}_{\mathrm{N}}\ 0\ (d_1 \diamond \text{union}\ d_2\ d_3)$$

is `"\n\n"`, whereas

$$\textit{render}_{\mathrm{N}}\ 0\ (\text{union}\ (d_1 \diamond d_2)\ (d_1 \diamond d_3))$$

is `"\n "`.

Let us see what is going on here. In the second case *best* chooses between $d_1 \diamond d_2$ and $d_1 \diamond d_3$, and because the first line of $d_1 \diamond d_2$ fits in the allotted width, this document is the one that is rendered. In the first case *best* instead chooses between d_2 and d_3. At this stage it is clear that d_2 does not fit in the allotted width, so d_3 is rendered instead.

Note that these expressions both satisfy Wadler's invariants: one for `text`, mentioned in Section 3, and two for `union`, mentioned in this section. Thus there seems to be a problem in Wadler's paper (the counterexample above can be ported to Wadler's own implementation).

Wadler specifies that, given a document representing a (finite) set of strings, his renderer should return the "best" string. "Best" is defined—following Hughes (1995)—by the lexicographic extension of the following binary relation on lines: if both lines fit in the available width, then the longer one (if any) is better; if neither line fits, then the shorter one (if any) is better; and if exactly one line fits, then that line is better. This criterion, together with the second `union` invariant, is used to motivate the implementation of the `union` case of *best*. However, the invariant is not strong enough: union $(d_1 \diamond d_2)\ (d_1 \diamond d_3)$ represents the strings `"\n "` and `"\n\n"`, and `"\n\n"` is better than `"\n "`, but the implementation gives us `"\n "`. This means that the notion of "best" implemented by *best* is not the one specified by Wadler. Fortunately it is not possible to construct union $(d_1 \diamond d_2)\ (d_1 \diamond d_3)$ using Wadler's public document interface (which does not include `union`). Thus it may be possible to prove that Wadler's renderer returns the best string by modifying his invariants in some way.

6. Related Work

There are at least four approaches to correct-by-construction pretty-printing:

- *Grammars with embedded pretty-printing directives.* Oppen (1980) mentions that one can add pretty-printing directives to grammars. Rubin (1983) and Boulton (1996) both describe how context-free grammars can be extended with embedded pretty-printing directives, from which parsers and pretty-printers can be generated. The Ergo Support System's Syntax Facility seems to have similar features (Lee et al. 1988).

  None of the papers referred to here contain proofs of a round-tripping property. However, it seems plausible to me that this approach to correct-by-construction pretty-printing *could* (when done right) be proved to be correct.

- *Combinators for invertible programming.* Alimarine et al. (2005) present combinators for invertible programming, and when introducing the definition of a parser they claim that they "will get the inverse, a pretty-printer, for free". However, they later write that "To show correctness, global reasoning is required", so this does not appear to be an example of *correct-by-construction* pretty-printing.

  As mentioned in Section 1 Rendel and Ostermann (2010) describe combinators that allow the simultaneous definition of parsers and printers. The combinators are based on "partial isomorphisms". No round-tripping property is proved. The approach is arguably somewhat fragile: if a choice "*p* or *q*" is replaced by "*q* or *p*", then a working pretty-printer can be turned into a non-terminating pretty-printer.[7] Rendel and Ostermann do not explain in detail how to handle line widths, word wrapping, etc., but suggest that it may be possible to support more advanced pretty-printing features.

- *Pretty-printers with extra grammar information.* This is the approach taken by Matsuda and Wang (2013), described in Sections 1 and 4.2. Matsuda and Wang outline the proof of a

[6] Agda cannot infer all the implicit arguments in this expression. One can give the arguments explicitly using the notation $\_\diamond\_\ \{g_1 = g_1\}\ \{g_2 = g_2\}$.

[7] This was pointed out by Lennart Augustsson when Rendel and Ostermann's work was presented at Haskell'10 in Baltimore.

round-tripping property (but the proof is not mechanised, and no guarantee is given that the pretty-printers will terminate). They also include a "nondeterministic printing semantics" that is reminiscent of the grammar indices that I use for *Doc*: a line is non-deterministically printed as one or more whitespace characters, and the group and nest combinators are ignored. Matsuda and Wang's development is limited to context-free languages. The development is arguably quite complicated: it uses program inversion, fusion and partial evaluation.

- *Grammars along with pretty-printers that construct indexed pretty-printer documents.* This is the approach taken in the present paper. A formal, mechanised proof of round-tripping is provided. Furthermore the approach supports any recursively enumerable language.

  The round-tripping property is perhaps not so interesting if there is no parser. Danielsson (2010) describes parser combinators that can handle any finitely ambiguous language for which it is possible to implement a parser in the host language, and proves correctness formally.

If we compare these approaches then we can see that a potential disadvantage of the one presented in this paper is that the user may have to write manual proofs. However, one can capture many commonly occurring patterns in reusable libraries—for instance, the user can write *left d* instead of *embed proof d*. Given suitable library combinators it seems as if manual proofs should mostly be needed when the pretty-printer deviates from the grammar's structure (as in Section 4.3)—and such deviation is not even possible when Matsuda and Wang's approach is used. It is also possible to automate some of the proofs. One example is provided in Section 4.3; it is conceivable that many other proofs can also be automated, but I have not investigated this in detail.

Another potential disadvantage of the approach presented in this paper is that the user has to write two things: a grammar and a pretty-printer. In Section 1 I argue for the separation of grammars and pretty-printers. However, this is my subjective view. Rose and Welsh (1981) argue that pretty-printing information *should* be part of the definition of a programming language's syntax.

Let me finally mention the work of Foster et al. (2008) on quotient lenses. A central part of this development concerns "canonisers". A canoniser from A to $B/\sim$ (where A and B are sets, and $\sim$ is an equivalence relation on B) consists of two functions $\mathit{canonise} : A \to B$ and $\mathit{choose} : B \to A$, satisfying the law

$$\forall\, b.\ \mathit{canonise}\ (\mathit{choose}\ b) \sim b.$$

Foster et al. describe a canoniser where, basically, *canonise* replaces newline characters with spaces, and *choose* selectively replaces spaces with newline characters to wrap long lines.

7. Conclusion

I have presented a new approach to correct-by-construction pretty-printing. The approach is very close to the one of classical (not-necessarily-correct) pretty-printing. The main difference is that pretty-printer documents are more precisely typed, which ensures that documents are correct with respect to given values and grammars. The development is based on very general grammars and Wadler's pretty-printing combinators, but the ideas should carry over to other grammar and/or pretty-printing frameworks.

The round-tripping property that is proved in Section 5 depends on having an unambiguous grammar (or at least a grammar that is unambiguous for strings in the image of the pretty-printer). I have not discussed how one can prove that a grammar is unambiguous—I see this as an orthogonal problem.

The correctness property that renderers have to satisfy only concerns grammatical correctness, not "prettiness": renderers have some freedom in how to interpret their inputs, as witnessed by the two very different renderers in Section 5. The ugly-renderer can be used to generate compact output, and the other one to generate pretty output—in both cases the result is guaranteed to be grammatically correct (although perhaps ambiguous). Other renderers could also be useful. One may for instance want to use a "ribbon width", a limit on the number of non-indentation characters occurring on a line (Hughes 1995).

Acknowledgements

I would like to thank Kazutaka Matsuda and Meng Wang for inspiring me to do the work reported in this paper. I would also like to thank Conor McBride, Aaron Stump, Philip Wadler and some anonymous reviewers for useful feedback.

The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement n° 247219.

References

The Agda Team. The Agda Wiki. Available at `http://wiki.portal.chalmers.se/agda/`, 2013.

Artem Alimarine, Sjaak Smetsers, Arjen van Weelden, Marko van Eekelen, and Rinus Plasmeijer. There and back again: Arrows for invertible programming. In *Haskell'05, Proceedings of the ACM SIGPLAN 2005 Haskell Workshop*, pages 86–97, 2005. doi:10.1145/1088348.1088357.

Richard J. Boulton. Syn: A single language for specifiying abstract syntax trees, lexical analysis, parsing and pretty-printing. Technical Report UCAM-CL-TR-390, University of Cambridge Computer Laboratory, 1996.

Nils Anders Danielsson. Total parser combinators. In *ICFP'10, Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, pages 285–296, 2010. doi:10.1145/1863543.1863585.

J. Nathan Foster, Alexandre Pilkiewicz, and Benjamin C. Pierce. Quotient lenses. In *ICFP'08, Proceedings of the 2008 SIGPLAN International Conference on Functional Programming*, pages 383–395, 2008. doi:10.1145/1411204.1411257.

John Hughes. The design of a pretty-printing library. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques*, volume 925 of *LNCS*, pages 53–96, 1995. doi:10.1007/3-540-59451-5_3.

Graham Hutton and Erik Meijer. Monadic parsing in Haskell. *Journal of Functional Programming*, 8(4):437–444, 1998. doi:10.1017/S0956796898003050.

Peter Lee, Frank Pfenning, Gene Rollins, and William Scherlis. The Ergo Support System: An integrated set of tools for prototyping integrated environments. In *Proceedings of the third ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 25–34, 1988. doi:10.1145/64135.65006.

Kazutaka Matsuda and Meng Wang. FliPpr: A prettier invertible printing system. In *Programming Languages and Systems, 22nd European Symposium on Programming, ESOP 2013*, volume 7792 of *LNCS*, pages 101–120, 2013. doi:10.1007/978-3-642-37036-6_6.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology and Göteborg University, 2007.

Derek C. Oppen. Prettyprinting. *ACM Transactions on Programming Languages and Systems*, 2(4):465–483, 1980. doi:10.1145/357114.357115.

Simon Peyton Jones. John Hughes's and Simon Peyton Jones's pretty printer combinators. Haskell source code, 1996. A more recent version of the library is, at the time of writing, available from `http://hackage.haskell.org/package/pretty`.

Tillmann Rendel and Klaus Ostermann. Invertible syntax descriptions: Unifying parsing and pretty printing. In *Haskell'10, Proceedings of the*

2010 ACM SIGPLAN Haskell Symposium, pages 1–12, 2010. doi:10.1145/1863523.1863525.

G. A. Rose and J. Welsh. Formatted programming languages. *Software: Practice and Experience*, 11(7):651–669, 1981. doi:10.1002/spe.4380110702.

Lisa F. Rubin. Syntax-directed pretty printing—a first step towards a syntax-directed editor. *IEEE Transactions on Software Engineering*, SE-9(2):119–127, 1983. doi:10.1109/TSE.1983.236456.

S. Doaitse Swierstra and Olaf Chitil. Linear, bounded, functional pretty-printing. *Journal of Functional Programming*, 19(1):1–16, 2009. doi:10.1017/S0956796808006990.

Philip Wadler. A prettier printer. In *The Fun of Programming*. Palgrave Macmillan, 2003.