

# An expressive and optimal pretty printer

ANONYMOUS AUTHOR(S)

Pretty printers make trade-offs between the *expressiveness* of their pretty printing language, the *optimality* objective that they minimize when choosing between different ways to lay out a document, and the *performance* of their layout algorithm. This paper presents a new pretty printer,  $\Pi_e$ , that is strictly more expressive than all pretty printers in the literature and provably minimizes the optimality objective. Furthermore, the time complexity of  $\Pi_e$  is better than many existing pretty printers. When choosing among different ways to lay out a document,  $\Pi_e$  consults a user-supplied *cost factory*, which determines the optimality objective, giving  $\Pi_e$  a unique degree of flexibility. We use the Lean theorem prover to verify the correctness and optimality of  $\Pi_e$ , and implement  $\Pi_e$  concretely as a pretty printer that we call SNOWWHITE. To evaluate our pretty printer against others, we develop a formal framework for reasoning about the expressiveness of pretty printing languages, and survey pretty printers in the literature, comparing their expressiveness, optimality, worst-case time complexity, and practical running time. Our evaluation shows that SNOWWHITE is efficient and effective at producing optimal layouts. SNOWWHITE has also seen real-world adoption: it serves as a foundation of a code formatter for Racket.

## 1 INTRODUCTION

General-purpose pretty printers (or, simply, *printers*) are widely used to convert structured data—typically an AST—into human-readable text. Their applications include code reformatting, software reengineering, and synthesized code printing [De Jonge 2002; Prettier 2016; Torlak and Bodik 2014; Yelland 2015]. These printers take as inputs (1) a document in a pretty printing language (PPL), which encodes the structured data along with styling choices, and (2) a page width limit. Choices in the document can yield (exponentially) many possible layouts. The task of the printers then is to efficiently choose an *optimal* layout from all possible layouts. Existing printers use a variety of built-in optimality objectives. A good objective reflects the informal notion of “prettiness,” such as not having an overflow over the page width limit whenever possible while having as few lines as possible.

Different printers make different trade-offs in the *expressiveness* of the PPL, the *optimality* objective, and the *performance*. With the current landscape of pretty printing, developers who wish to use expressive features from various PPLs, optimize certain objectives, and achieve practical running time are left with two possibilities. They can write an ad-hoc printer, which is time-consuming and error-prone. Alternatively, they can pick an existing general-purpose printer, potentially giving up some desired features. This is further complicated by some common misunderstandings about the three aspects of these existing printers.

This paper presents a printer that we call  $\Pi_e$ . It targets  $\Sigma_e$ , a PPL that is strictly more expressive than all published PPLs. This can be shown via our formal framework for reasoning about the expressiveness of PPLs.  $\Pi_e$  is parameterized by a *cost factory*, which enables developers to specify an optimality objective for  $\Pi_e$  to minimize. The cost factory is versatile. For example, it can express non-linear costs and define concepts such as soft page width limits [Yelland 2016]. As a result, the optimal layout that  $\Pi_e$  chooses can have higher quality compared to existing printers. The time complexity of  $\Pi_e$  is  $O(nW^4)$ , where  $n$  is the size of the document and  $W$  is the computation width limit (defined in Section 5). This is better than the time complexity of many printers in the literature, and it is improved to  $O(nW^3)$  when  $\Pi_e$  is restricted to process documents in some well-known but less expressive PPLs. We prove the correctness of  $\Pi_e$  in the Lean theorem prover [Moura and Ullrich 2021], ensuring the validity and optimality of the output layout, and demonstrate  $\Pi_e$ ’s efficiency by evaluating our implementation of  $\Pi_e$ , which we call SNOWWHITE. We believe these attributes make  $\Pi_e$  not only a good printer by itself, but also a good building block to construct other derived printers.

2023. 2475-1421/2023/4-ART1 \$15.00

<https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

```

50 text "function_append(first,second,third){"
51 <> nest 4 (
52   let f = text "first_" in
53   let s = text "second_" in
54   let t = text "third" in
55   nl <> text "return_" <>
56   group (nest 4 (f <> nl <> s <> nl <> t))
57 ) <> nl <> text "}"

```

```

1 function append(first,second,third){
2   return first +
3     second +
4     third
5 }

```

```

1 function append(first,second,third){
2   return first + second + third
3 }

```

(a) A document in the traditional PPL and its corresponding layouts. The `nest` construct increments the current indentation level by some specified amount, causing `nl` (newline) to insert indentation spaces. `<>` is the unaligned concatenation operator, which places the right sub-layout after the left sub-layout on the current indentation level. Lastly, the `group` construct creates two formatting choices: one where the sub-layouts are left alone and one where the sub-layouts are flattened by replacing newlines and indentation spaces due to `nls` in the group with single spaces.

```

63 text "function_append(first,second,third){" <$>
64 ( let f = text "first_" in
65   let s = text "second_" in
66   let t = text "third" in
67   let sp = text "_" in
68   let indentation = text "    " in
69   let ret = text "return_" in
70   indentation <+>
71   (((ret <+> text "(") <$>
72     (indentation <+> (f <$> s <$> t)) <$>
73     text ")") <|>
74     (ret <+> f <+> sp <+> s <+> sp <+> t)))
75 <$> text "}"

```

```

1 function append(first,second,third){
2   return (
3     first +
4     second +
5     third
6   )
7 }

```

```

1 function append(first,second,third){
2   return first + second + third
3 }

```

(b) A document in the arbitrary-choice PPL and its corresponding layouts. `<|>` is the arbitrary-choice operator, which per its namesake, creates two formatting choices from layouts by arbitrary sub-document. `<$>` is the vertical concatenation operator, which joins two sub-layouts with a newline. Lastly, `<+>` is the aligned concatenation operator, which joins two sub-layouts horizontally, aligning the whole right sub-layout at the column where it is to be placed in.

Fig. 1. The traditional and arbitrary-choice PPLs, embedded in the host language OCaml. Colored regions in a document and corresponding layouts indicate the correspondence between the colored sub-documents and the colored sub-layouts. We use the `let` construct to make the documents easier to read, even though it is usually not a part of PPLs. Dotted lines illustrate different page width limits at 22 and 36 characters.

*A survey of printers in the wild.* To evaluate  $\Pi_e$ , we conducted a broad survey of the literature on pretty printing. Most PPLs are embedded in a host programming language, provide a small set of core constructs that allow developers to create a document with text, concatenate documents together, set indentation level, and express formatting choices. High-level constructs can then be built on top of the core constructs. The details of these core constructs can differ from PPL to PPL. We found that there are two main schools of PPLs in the wild, which we call the *traditional* and *arbitrary-choice* PPLs. The traditional PPL centers around manipulation of `nls` (newlines) and current indentation level, while the arbitrary-choice PPL is characterized by the ability to express arbitrary formatting choices and the use of aligned concatenation to supplant the concept of indentation level. Figure 1 illustrate documents in both PPLs that pretty-print the function definition `append` in a hypothetical programming language with slightly different styling.

*Expressiveness.* The literature contains informal claims about the expressiveness of PPLs [Chitil 2005; Podkopaev and Boulytchev 2015; Wadler 2003]. We develop two formal notions of expressiveness: the ability to *express layouts* and the ability to *express features*. The former reflects the

functionality of a PPL, while the latter reflects the ease of document construction. Using our framework, we can show that neither the traditional PPL nor the arbitrary-choice PPL is more expressive than the other. For example, the set of layouts in Figure 1b cannot be expressed by any document in the traditional PPL. This is because all layouts due to a particular document in the traditional PPL must be the same modulo whitespace, but one of the layouts in the figure has an extra pair of parentheses.<sup>1</sup> As another example, the document in Figure 1b is awkwardly constructed. It would be more natural to use unaligned concatenation, but the feature cannot be expressed by any combination of features in the arbitrary-choice PPL.<sup>2</sup> To that end, we develop a PPL called  $\Sigma_e$  that is provably strictly more expressive than both the traditional and arbitrary-choice PPLs, facilitating both functionality and ease of document construction.

*Optimality.* The optimality objective of a printer indicates what it optimizes for when resolving choices. Most printers targeting the traditional PPL minimize overflow over the page width limit line-by-line, preferring a longer line when there is no overflow. For example, given the document in Figure 1a, the first layout is optimal when the page width limit is 22 (red dotted line), while the second layout is optimal when the page width limit is 36 (green dotted line). Contrary to prior claims [Chitil 2005; Wadler 2003], we discovered that this strategy guarantees neither the absence of overflow whenever possible nor the minimality of the number of lines. In contrast, most printers targeting the arbitrary-choice PPL minimize the number of lines among layouts with no overflow. However, they *error* when all possible layouts have an overflow, resulting in a poor user experience (e.g., when the width limit is 22 in Figure 1b). Recognizing that unavoidable overflows do occur in practice, we introduce the concept of a *cost factory*. The factory allows developers to choose a desired objective permitted by its interface, including an objective that tolerates overflow gracefully.

*Performance.* Printing proceeds in two phases: resolving choices and rendering the optimal choice to text (although many printers fuse these two phases together). Time complexity of printers is best measured against the resolving phase<sup>3</sup>, and it is usually specified with two parameters: the size of the document  $n$  and the width limit  $W$ , with the preference that the time complexity be polynomial in  $W$  and linear in  $n$ . Most printers in the literature leave their time complexity unanalyzed, instead opting to show experimental results that their implementations are efficient in practice. We analyze these printers and demonstrate documents that trigger worse than linear time behavior (in  $n$ ) on some printers. Further complication arises in printers with the arbitrary choice feature, which gives rise to documents that are structured as DAGs as opposed to trees. We show that many printers that treat the input document as a tree suffer from a combinatorial explosion as the DAG structure is unfolded, resulting in exponential time complexity. With a combination of proof and experimental results, we show that the time complexity of  $\Pi_e$  is linear in the DAG size of the document and that it runs fast in practice.

In summary, this paper makes the following contributions:

- A new PPL called  $\Sigma_e$  that is strictly more expressive than all published PPLs. Constructs in  $\Sigma_e$  are not new, but packaging them all in a single PPL has never been done before.
- A printer  $\Pi_e$  targeting  $\Sigma_e$  that utilizes a *cost factory* to allow a variety of optimality objectives.

<sup>1</sup>Languages such as Python require an extra pair of parentheses around an expression that spans multiple lines [The Python Language Reference 2010]. Similarly, some styles prefer adding an extra comma (also known as trailing comma) when a function call spans multiple lines [ESLint 2014]. Hence, the ability to express layouts with differing content is desirable.

<sup>2</sup>Different programming language styles prefer different concatenation operators. C-like languages heavily use unaligned concatenation, while aligned concatenation has been used for Haskell, Lisp, R, and Julia. However, there are instances where C-like languages would benefit from aligned concatenation, and Haskell would benefit from unaligned concatenation.

<sup>3</sup>This formulation allows us to talk about “linear-time” printers, even though there are, e.g., documents whose size is  $O(n)$ , but the rendering would result in  $O(n^2)$  characters.

Table 1. A comparison of existing printers.  $n$  and  $\hat{n}$  are the DAG size and tree size of the input document (where  $\hat{n}$  in the worst case is exponential in  $n$ ).  $W$  is the width limit.

Printer	Expressiveness		Optimality	Performance
	Choice	Concatenation	Minimization objective	Time complexity
Oppen [1980]	Group	Unaligned	Lexicographic overflow	$O(n)$
Hughes [1995]	Group	Aligned	Lexicographic overflow	$O(n^2)$
Wadler [2003]	Group	Unaligned	Lexicographic overflow	$O(n^2)$
Leijen [2000]	Group	Both	Lexicographic overflow	$O(n^2)$
Chitil [2005]	Group	Unaligned	Lexicographic overflow	$O(n)$
Kiselyov et al. [2012]	Group	Unaligned	Lexicographic overflow	$O(n)$
Swierstra et al. [1999]	Arbitrary	Aligned	Height <sup>†</sup>	Exp. in $n$
Podkopaev and Boulytchev [2015]	Arbitrary	Aligned	Height <sup>†</sup>	$O(\hat{n}W^4)$
Yelland [2016]	Arbitrary	Aligned	Linear cost	$O(\hat{n}^{3/2})$
Bernardy [2017c]	Arbitrary	Aligned	Height <sup>†</sup>	$O(nW^6)$
$\Pi_e$	Both	Both	Cost (from the cost factory)	$O(nW^4)$
$\Pi_e$ (aligned only)	Both	Aligned	Cost (from the cost factory)	$O(nW^3)$

<sup>†</sup> only consider layouts without an overflow over  $W$ .

- A proof of correctness for  $\Pi_e$ , formalized in the Lean theorem prover. To our knowledge, this is the first time that a printer has been formally verified.
- A framework to formally reason about the expressiveness of PPLs.
- A survey of printers and an analysis that dispels common misunderstandings about them.
- An implementation of  $\Pi_e$ , SNOWWHITE, and an evaluation that shows its effectiveness.

The rest of this paper is structured as follows. [Section 2](#) surveys the related work. [Section 3](#) presents the semantics of  $\Sigma_e$ . [Section 4](#) introduces a framework to reason about the expressiveness of PPLs. [Section 5](#) presents  $\Pi_e$  and its analysis. [Section 6](#) discusses SNOWWHITE, an implementation of  $\Pi_e$ . [Section 7](#) presents an evaluation of SNOWWHITE that demonstrates its effectiveness. Lastly, [Section 8](#) concludes the paper.

## 2 RELATED WORK

To understand the trade-off space of printer designs, we conduct a comprehensive analysis of related work in the literature. This section provides our analysis of the printers, grouped by the expressiveness of their public interface.<sup>4</sup> The summary is presented in [Table 1](#). We then compare and contrast our printer  $\Pi_e$  against them.

### 2.1 Traditional printers

Pretty printing has a long history. [Oppen \[1980\]](#) first introduced a general-purpose printer, written in the imperative style. Oppen pioneered the PPL that we call the traditional PPL, shown in [Figure 2a](#). Instead of representing an input document as a tree, as commonly done in subsequent work, Oppen represents the document as a stream of “instruction tokens.” The algorithm’s time complexity is  $O(n)$ , where  $n$  is the length of the stream. Furthermore, the algorithm is *bounded*, requiring a limited look-ahead into the stream. As with other printers in the family, the printer greedily minimizes overflow over the page width limit, which neither avoids overflow whenever possible nor minimizes number of lines, as discussed in the paper.

[Wadler \[2003\]](#) designed a printer that targets the traditional PPL. It is used in many real world applications, such as an industrial code formatter [[Prettier 2016](#)], and as a basis for much pretty printing research [[Chitil 2005](#); [Kiselyov et al. 2012](#)]. The printer aims to be a rewrite of Oppen’s

<sup>4</sup>In practice, printers include extensions that increase their expressiveness. A printer may even have different expressiveness across different versions. This section focuses on the core features of these printers.

$d \in \mathcal{D} ::=$  **text**  $s$     text  
 | **nl**    newline  
 |  $d \triangleleft d$     unaligned concatenation  
 | **nest**  $n$   $d$     increase indentation level  
 | **group**  $d$     create two choices where one flattens layouts

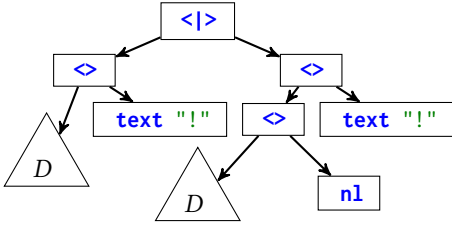
$d \in \mathcal{D} ::=$  **text**  $s$     text  
 |  $d_a \triangleleft+ d_b$     aligned concatenation  
 |  $d_a \triangleleft\$ d_b$     vertical concatenation  
 |  $d_a \triangleleft| d_b$     create two arbitrary choices

(a) A variant of traditional PPL from Wadler [2003].

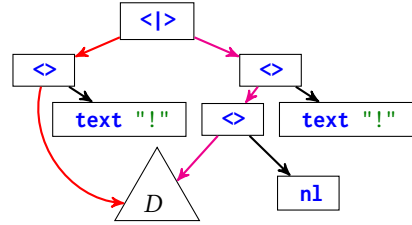
(b) A variant of arbitrary-choice PPL from Podkopaev and Boulytchev [2015].

Fig. 2. A comparison between the traditional and arbitrary-choice PPLs.  $s$  denotes a string without newline, and  $n$  denotes a natural number.

**let** shared  $:= D$  **in** (shared  $\triangleleft$  **text** "!" )  $\triangleleft|$  ((shared  $\triangleleft$  **nl**)  $\triangleleft$  **text** "!" )

(a) A document that encodes (at least) two possible layouts.  $D$  is an arbitrary sub-document.

(b) A tree representation of Figure 3a.  $D$  contributes to the size twice.



(c) A DAG representation of Figure 3a.  $D$  contributes to the size only once.

Fig. 3. An example document that shows the importance of treating document as a DAG rather than a tree. The red and pink paths illustrate that the DAG is properly shared, as will be discussed in Section 5.5.

printer using the functional style employed by Hughes (described later). The printer is claimed [Chitil 2005; Wadler 2003] to produce an output layout that does not exceed the width limit whenever possible, and minimizes the number of lines. However, this is not the case, as shown in Figure 14 (in Appendix A). The time complexity of the printer is claimed to be  $O(n)$  where  $n$  is the size of document [Wadler 2003], but it is in fact  $O(n^2)$  in the worst case, as demonstrated in Figure 15, although this worst case behavior is unlikely to occur in practice.

Chitil [2005] improved Wadler's printer so that it is as efficient as Oppen's,  $O(n)$ , by using lazy dequeues. Kiselyov et al. [2012] similarly improved Wadler's printer via their generator framework.

Compared to traditional printers,  $\Pi_e$  is more expressive as it allows arbitrary choices and aligned concatenation. Furthermore,  $\Pi_e$  can produce an output layout that minimizes number of lines when the output layout does not exceed the page width limit, and does not exceed the page width limit whenever possible. The tradeoff is that  $\Pi_e$  is less space efficient and slower than traditional printers. The space complexity of traditional printers is sub-linear in the size of document, which was especially important decades ago when memory is scarce. The space complexity of  $\Pi_e$  is  $O(nW^3)$  in the worst case (or  $O(nW^2)$  when targeting some PPLs). We find that on modern machines, the added memory consumption and performance overhead are rarely an issue in practice (Section 7).

## 2.2 Arbitrary-choice printers

Azero Alcocer and Swierstra [1998] introduced a printer that supports arbitrary choices with aligned concatenation, starting the line of work that targets the arbitrary-choice PPL, shown in Figure 2b. The printer's optimality objective is to avoid overflow whenever possible and produce a minimal number of lines. However, it does not have the ability to cope with unavoidable overflow. This printer was soon superseded by Swierstra et al. [1999], which improves its performance via



heuristics and adds the capability to *share* a sub-document across choices by deeply embedding the (equivalent of the) `let` construct in the PPL. As a result, the later printer can process documents that are structured as DAGs rather than trees, as shown in Figure 3. Nonetheless, the time complexity of both printers is exponential in  $n$  [Podkopaev and Boulytchev 2015].

Podkopaev and Boulytchev [2015] improved upon Swierstra et al.’s work by formulating the problem as dynamic programming. The formulation fixes the exponential blowup in the prior work, but treats the document as a tree, making its time complexity  $O(\hat{n}W^4)$ , where  $\hat{n}$  is the tree size of the document, which could be exponentially larger than its DAG size. The paper acknowledges the problem and surmises that memoization may be able to address it.

The paper by Bernardy [2017c] is the main inspiration for our work. The printer uses Pareto frontiers to find an optimal layout. By shallowly embedding the PPL (in Haskell), computations on sub-documents are effectively shared for free. However, as presented in the paper, the printer requires the page width limit to be hard-coded. In the actual implementation [Bernardy 2017b], the page width limit is customizable, accomplished by threading the value through functions. But this change destroys the shared computations, leading to exponential running time. Compared to Podkopaev and Boulytchev [2015]’s work, Bernardy [2017c]’s approach can exploit sparseness to improve practical efficiency, but the use of an inefficient algorithm makes the time complexity of the printer  $O(nW^6)$  in the worst case. While the paper does not handle unavoidable overflow, the implementation does by automatically scaling up the page width limit (or equivalently, minimizing the maximum overflow). This, however, allows avoidable overflow elsewhere, as shown in Figure 16 (in Appendix A), which is undesirable. Later on, Bernardy abandoned the arbitrary-choice operator, noting that it could trigger the exponential behavior [Bernardy 2017a].

Yelland [2016] similarly targeted the arbitrary-choice PPL. However, the paper took a very different approach. The core printer restricts the use of aligned concatenation by requiring the left sub-document to be a `text` syntactically. This restriction allows the core printer to utilize the concept of “piecewise linear cost function” to seemingly boost the performance. To achieve the expressiveness of the arbitrary-choice PPL, the printer employs rewriting rules to transform the original document into the restricted document. While the printer is careful to avoid exponential blowup by sharing documents in the resulting restricted document, it does not necessarily preserve the sharing structure of the original document, as demonstrated in Figure 17 (in Appendix A). Compound this with the lack of computation width limit, the number of piecewise linear cost functions under consideration could be as large as  $O(\hat{n}^{1/2})$ , making the time complexity  $O(\hat{n}^{3/2})$  in total, as shown in Figure 18. Another aspect to consider is the printer’s optimality objective, which is restricted to minimizing a linear combination of quantities like the number of lines and overflow. Hence, the printer will not technically avoid overflow whenever possible (although the overflow coefficient can be made arbitrarily large to arbitrarily discourage overflow). On the other hand, this optimality objective can support unique features, such as incorporating the costs due to multiple soft page width limits.

Compared to arbitrary-choice printers,  $\Pi_e$  is more expressive as it allows unaligned concatenation.  $\Pi_e$  is also asymptotically faster than most arbitrary-choice printers, as it treats a document as a DAG rather than a tree. Like Yelland’s printer, for each layout under consideration,  $\Pi_e$  keeps track of two quantities: cost and last line length. This is different from most printers in the family which keep track of three quantities: height, width, and last width. The dimension reduction further makes  $\Pi_e$  more efficient. The concept of cost also allows  $\Pi_e$  to decouple the page width limit and computation width limit, which allows graceful overflow handling.  $\Pi_e$ , unlike Yelland’s printer, is parameterized by a cost factory, which supports a variety of optimality objectives without requiring a modification to the core printer. This includes not only the linear optimality objectives that Yelland’s printer supports, but also non-linear optimality objectives.

Document  $d \in \mathcal{D}_e ::= \text{text } s \mid \text{nl} \mid d \triangleleft d \mid \text{nest } n \, d \mid$  String without newline  $s, t, \dots \in \text{Str}$   
 $\text{align } d \mid \text{flatten } d \mid d \triangleleft \triangleright d$  Natural number  $n \in \mathbb{N}$

Fig. 4. Syntax for  $\Sigma_e$ 

## 2.3 Other printers

Hughes [1995] brought a general-purpose printer to the functional world. The printer pioneers using combinators to construct a document, which is now a standard practice. The printer targets a PPL that is neither the traditional nor arbitrary-choice PPL, though the printer is more similar to the traditional printers in how it makes choices greedily, which minimizes neither overflow nor number of lines. The combination of greedy choice making and aligned concatenation makes some documents print very poorly [Bernardy 2017c]. Furthermore, Peyton-Jones [1997] identified quadratic time complexity in the printer.

Leijen [2000] implemented Wadler's printer in Haskell and added support for aligned concatenation via the inclusion of `align`, becoming the first printer that supports both aligned and unaligned concatenation. However, similar to Hughes' printer, the printer can produce very poor output [Bernardy 2017c].

## 3 THE SYNTAX AND SEMANTICS OF $\Sigma_e$

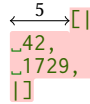
This section presents  $\Sigma_e$ , an expressive PPL. We begin this section by describing *layouts*, which are the textual outputs. Then, we describe the syntax of  $\Sigma_e$  and its informal semantics, which is determined by the evaluation of a document in  $\Sigma_e$  to layouts. Lastly, we formally describe the semantics of  $\Sigma_e$ .

### 3.1 Layout

A layout  $l \in \mathcal{L}$  is a textual output. In our work, we represent a layout as a non-empty, finite list of lines (implicitly joined by newlines), where each line is a string without the newline character. This allows us to easily reason about the number of lines and the length of each line.

A layout can be *placed* at a column position  $c$ . This placing puts the first line of the layout at the column position  $c$ , and puts the rest of the lines at the column position 0. While the final layout to present to users is placed at the column position 0, a (sub)layout may be placed at a non-zeroth column position during the rendering process.

*Example 3.1.* The following illustration displays a layout `["[", "_42,", "_1729,", "|"]` placed at the column position 5. The pink area shows the general shape of layout "boundaries", where the first line could be "indented" due to the placing on a non-zeroth column position, and the last line covers its length exactly. The shape will be useful to understand how layouts are composed together.



### 3.2 The syntax and the informal semantics of $\Sigma_e$

Figure 4 shows the syntax of document in  $\Sigma_e$ . Each construct is from either the traditional or the arbitrary-choice PPLs, except the flatten construct `flatten` (which is internally used in Wadler [2003]'s printer) and the align construct `align` (which is from Leijen [2000]'s printer). Similar to layouts, documents can be placed at a column position. As traditionally done, we for now ignore the choice operator. A document without the arbitrary-choice operator is called a *choiceless document*, which can be *rendered* to a single layout. We denote a choiceless document with  $\bar{d} \in \bar{\mathcal{D}}_e$ . The informal semantics of choiceless document are as follows:

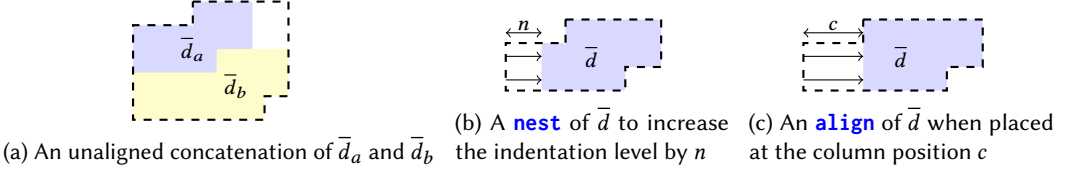


Fig. 5. Illustrations of constructs in  $\Sigma_e$ . The area with dashed borders is the resulting boundaries.

<b>text</b> $s$	renders to a layout with a single line $s$ .
<b>nl</b>	normally renders to a layout with two lines. The first line is empty, and the second line consists of $i$ spaces where $i$ is the <i>indentation level</i> . <b>nl</b> interacts with <i>flattening</i> , which reduces it to just a single space.
$\bar{d}_a \ltimes \bar{d}_b$	renders to a layout that concatenates the layout of $\bar{d}_a$ and the layout of $\bar{d}_b$ without alignment. The rendering of $\bar{d}_b$ is dependent on the rendering of $\bar{d}_a$ , because $\bar{d}_b$ will be placed at a column position after the last character of $\bar{d}_a$ . Figure 5a illustrates this.
<b>nest</b> $n \bar{d}$	renders to a layout like $\bar{d}$ , but with the indentation level <i>increased</i> by $n$ relative to the current indentation level. Figure 5b roughly illustrates this.
<b>align</b> $\bar{d}$	renders to a layout like $\bar{d}$ , but with the indentation level <i>set</i> (not relatively increased) to the column position that <b>align</b> $\bar{d}$ is being placed in. Figure 5c roughly illustrates this.
<b>flatten</b> $\bar{d}$	renders to a layout just like $\bar{d}$ , but with all newlines and indentation spaces due to <b>nl</b> s flattened to single spaces.

While Figure 5 provides a rough illustration that should be helpful to understand the semantics of choiceless document, it could be misleading.

*Example 3.2.* The document **text** "a"  $\ltimes$  (**nest** 42 (**align** (**text** "b"  $\ltimes$  **nl**  $\ltimes$  **text** "c")))) is rendered to ["ab", "\_c"]. The nesting doesn't visibly increase the indentation level by 42, because the alignment on the inner document overrides the indentation level. This example shows the importance of the indentation level, and why it must be specifically tracked.

This concludes our informal description of how a choiceless document renders to a layout. General documents, by contrast, can contain the arbitrary-choice operator  $\ltimes$ , which encodes layouts from two sub-documents into one document. Thus, unlike choiceless documents, which render to a single layout, general documents will *evaluate* to a non-empty, finite set of layouts. Our approach is to first *widen* a document into a set of choiceless documents, then render each choiceless document in the set to produce a set of layouts.

### 3.3 The formal semantics of $\Sigma_e$

The formal semantics of  $\Sigma_e$ , consisting of two relations, is given in Figure 6. The judgment  $\langle \bar{d}, c, i, f \rangle \Downarrow_{\mathcal{R}} l$  states that the choiceless document  $\bar{d} \in \mathcal{D}_e$  placed at column position  $c \in \mathbb{N}$  with indentation level  $i \in \mathbb{N}$  and flattening mode  $f \in \mathbb{B}$ , will render to the layout  $l \in \mathcal{L}$ . The flattening mode  $f$ , which indicates whether newlines should be replaced with spaces, can be either on ( $\top$ ) or off ( $\perp$ ). Another judgment  $d \Downarrow_{\mathcal{W}} \bar{D}$  states that a document  $d \in \mathcal{D}_e$  is widened to a finite, non-empty set of choiceless documents  $\bar{D} \in 2^{\mathcal{D}_e}$ . We sometimes call a combination of  $c$  and  $i$  (and possibly  $f$ ) a *printing context*. Now, we elaborate some interesting rules in the figure.



$$\begin{array}{c}
\text{TEXT} \frac{}{\langle \text{text } s, c, i, f \rangle \Downarrow_{\mathcal{R}} [s]} \quad \text{FLATTEN} \frac{\langle \bar{d}, c, i, \top \rangle \Downarrow_{\mathcal{R}} [s]}{\langle \text{flatten } \bar{d}, c, i, f \rangle \Downarrow_{\mathcal{R}} [s]} \\
\text{LINENOFLATTEN} \frac{}{\langle \text{nl}, c, i, \perp \rangle \Downarrow_{\mathcal{R}} [\epsilon, " " \times i]} \quad \text{LINEFLATTEN} \frac{}{\langle \text{nl}, c, i, \top \rangle \Downarrow_{\mathcal{R}} [" "]} \\
\text{CONCATONE} \frac{\langle \bar{d}_a, c, i, f \rangle \Downarrow_{\mathcal{R}} [s] \quad \langle \bar{d}_b, c + |s|, i, f \rangle \Downarrow_{\mathcal{R}} [t, t_1, \dots, t_n]}{\langle \bar{d}_a \Diamond \bar{d}_b, c, i, f \rangle \Downarrow_{\mathcal{R}} [s \mathbin{\text{++}} t, t_1, \dots, t_n]} \quad \text{CONCATMULT} \frac{\langle \bar{d}_a, c, i, f \rangle \Downarrow_{\mathcal{R}} [s_1, \dots, s_n, s] \quad \langle \bar{d}_b, |s|, i, f \rangle \Downarrow_{\mathcal{R}} [t, t_1, \dots, t_m]}{\langle \bar{d}_a \Diamond \bar{d}_b, c, i, f \rangle \Downarrow_{\mathcal{R}} [s_1, \dots, s_n, s \mathbin{\text{++}} t, t_1, \dots, t_m]} \\
\text{NEST} \frac{\langle \bar{d}, c, i + n, f \rangle \Downarrow_{\mathcal{R}} [s_1, \dots, s_m]}{\langle \text{nest } n \bar{d}, c, i, f \rangle \Downarrow_{\mathcal{R}} [s_1, \dots, s_m]} \quad \text{ALIGN} \frac{\langle \bar{d}, c, c, f \rangle \Downarrow_{\mathcal{R}} [s_1, \dots, s_n]}{\langle \text{align } \bar{d}, c, i, f \rangle \Downarrow_{\mathcal{R}} [s_1, \dots, s_n]} \\
\text{TEXTWIDEN} \frac{}{\text{text } s \Downarrow_{\mathcal{W}} \{\text{text } s\}} \quad \text{LINEWIDEN} \frac{}{\text{nl} \Downarrow_{\mathcal{W}} \{\text{nl}\}} \\
\text{CONCATWIDEN} \frac{d_a \Downarrow_{\mathcal{W}} \bar{D}_a \quad d_b \Downarrow_{\mathcal{W}} \bar{D}_b}{d_a \Diamond d_b \Downarrow_{\mathcal{W}} \{\bar{d}_a \Diamond \bar{d}_b \mid \bar{d}_a \in \bar{D}_a, \bar{d}_b \in \bar{D}_b\}} \\
\text{NESTWIDEN} \frac{d \Downarrow_{\mathcal{W}} \bar{D}}{\text{nest } n d \Downarrow_{\mathcal{W}} \{\text{nest } n \bar{d} \mid \bar{d} \in \bar{D}\}} \quad \text{ALIGNWIDEN} \frac{d \Downarrow_{\mathcal{W}} \bar{D}}{\text{align } d \Downarrow_{\mathcal{W}} \{\text{align } \bar{d} \mid \bar{d} \in \bar{D}\}} \\
\text{FLATTENWIDEN} \frac{d \Downarrow_{\mathcal{W}} \bar{D}}{\text{flatten } d \Downarrow_{\mathcal{W}} \{\text{flatten } \bar{d} \mid \bar{d} \in \bar{D}\}} \quad \text{UNIONWIDEN} \frac{d_a \Downarrow_{\mathcal{W}} \bar{D}_\alpha \quad d_b \Downarrow_{\mathcal{W}} \bar{D}_\beta}{d_a \Diamond d_b \Downarrow_{\mathcal{W}} \bar{D}_\alpha \cup \bar{D}_\beta}
\end{array}$$

Fig. 6. Semantics for  $\Sigma_e$ . “ $\epsilon$ ” is the empty string. “ $s \times i$ ” is the notation for replicating the string  $s$  for  $i$  times. “ $s \mathbin{\text{++}} t$ ” is a string concatenation of  $s$  and  $t$ . Lastly, “ $s_1, \dots, s_n$ ” and “ $s_1, \dots, s_n$ ” indicate  $n$  lines, where  $n \geq 0$  and  $n \geq 1$  respectively.

*Rendering text.* The TEXT rule states that the rendering of a text placement **text**  $s$  contains a layout with a single line of the text  $s$ . The printing context is completely ignored: indentation level and flattening mode do not affect the rendering of text, and we have already assumed that  $s$  will be placed at the column position.

*Rendering newline.* When the flattening mode is off, the LINENOFLATTEN rule states that the rendering of **nl** results in a layout with two lines. The first line is empty, while the second line is indented by  $i$  spaces. On the other hand, when the flattening mode is on, the LINEFLATTEN rule states that the rendering of the newline results in a layout with a single line of a single space.

*Rendering unaligned concatenation.* In the rendering of  $\bar{d}_a \Diamond \bar{d}_b$ , we recursively render  $\bar{d}_a$  and  $\bar{d}_b$ , but the rendering of  $\bar{d}_b$  is dependent on the rendering of  $\bar{d}_a$ . Let  $l_a$  be the rendering result of  $\bar{d}_a$ . The CONCATONE rule handles the case where  $l_a$  has a single line, and the CONCATMULT rule handles the case where  $l_a$  has multiple lines.

- If  $l_a$  has only a single line  $s$ , the column position of  $\bar{d}_b$ ’s rendering needs to be after the string  $s$  is placed, i.e. at  $c + |s|$ . In such case, let  $l_b$  be the rendering result of  $\bar{d}_b$ . The first line of the resulting layout is the concatenation of  $s$  and the first line of  $l_b$ . The rest of the lines are from the rest of  $l_b$ .
- On the other hand, if  $l_a$  has multiple lines, the column position of  $\bar{d}_b$ ’s rendering is simply the column position after the last line is placed. In such case, let  $l_b$  be the rendering result of  $\bar{d}_b$ , the resulting layout contains all but the last line of  $l_a$ , a concatenation of the last line of  $l_a$  and the first line of  $l_b$ , and the rest of  $l_b$ .

*Widening choice.* The UNIONWIDEN rule states that the widening of  $d_a <|> d_b$  is the union of widen  $d_a$  and widen  $d_b$

Both  $\Downarrow_{\mathcal{R}}$  and  $\Downarrow_{\mathcal{W}}$  are deterministic and total. Thus, we can define  $\text{eval}_e(d) = \{l : \langle \bar{d}, 0, 0, \perp \rangle \Downarrow_{\mathcal{R}} l, \bar{d} \in \bar{D}, d \Downarrow_{\mathcal{W}} \bar{D}\}$  as the evaluation function for  $\Sigma_e$ , which consumes a document, widens it, and produces a set of layouts.

## 4 A FRAMEWORK TO REASON ABOUT EXPRESSIVENESS

In previous sections, we informally made claims about expressiveness of PPLs. This section presents a framework to formally reason about it, based on two notions: *functional completeness* and *definability*. We first define the semantics of the traditional and arbitrary-choice PPLs. Then, we define our framework, and show that  $\Sigma_e$  is strictly more expressive than both the traditional and arbitrary-choice PPLs. Proof sketches of theorems in this section are provided in [Appendix B](#).

### 4.1 The extended semantics

To reason about the traditional and arbitrary-choice PPLs, we need to precisely define their semantics. To do so, we construct a PPL  $\Sigma_{\text{all}}$  that contains all constructs from  $\Sigma_e$ , traditional, and arbitrary-choice PPLs by extending [Figure 6](#) with the rules below (with the straightforward widening rules). Note that we follow [Wadler \[2003\]](#)'s approach by treating **group**  $d$  as a syntactic sugar for  $d <|> \text{flatten } d$ . As  $<|>$  and **flatten** are already in  $\Sigma_{\text{all}}$ , we do not need to adjust anything further.

$$\begin{aligned}
 &\text{VERTCONCATNOFLATTEN} \frac{\langle \bar{d}_a, c, i, \perp \rangle \Downarrow_{\mathcal{R}} [s_1, \dots, s_n] \quad \langle \bar{d}_b, i, i, \perp \rangle \Downarrow_{\mathcal{R}} [t_1, \dots, t_m]}{\langle \bar{d}_a <\$> \bar{d}_b, c, i, \perp \rangle \Downarrow_{\mathcal{R}} [s_1, \dots, s_n, t_1, \dots, t_m]} \\
 &\text{VERTCONCATFLATTEN} \frac{\langle \bar{d}_a, c, i, \top \rangle \Downarrow_{\mathcal{R}} [s] \quad \langle \bar{d}_b, c + 1 + |s|, i, \top \rangle \Downarrow_{\mathcal{R}} [t]}{\langle \bar{d}_a <\$> \bar{d}_b, c, i, \top \rangle \Downarrow_{\mathcal{R}} [s ++ "\_\_" ++ t]} \\
 &\text{ALIGNEDCONCATONE} \frac{\langle \bar{d}_a, c, i, f \rangle \Downarrow_{\mathcal{R}} [s] \quad \langle \bar{d}_b, c + |s|, c + |s|, f \rangle \Downarrow_{\mathcal{R}} [t, t_1, \dots, t_n]}{\langle \bar{d}_a <+> \bar{d}_b, c, i, f \rangle \Downarrow_{\mathcal{R}} [s ++ t, t_1, \dots, t_n]} \\
 &\text{ALIGNEDCONCATMULT} \frac{\langle \bar{d}_a, c, i, f \rangle \Downarrow_{\mathcal{R}} [s_1, \dots, s_n, s] \quad \langle \bar{d}_b, |s|, |s|, f \rangle \Downarrow_{\mathcal{R}} [t, t_1, \dots, t_m]}{\langle \bar{d}_a <+> \bar{d}_b, c, i, f \rangle \Downarrow_{\mathcal{R}} [s_1, \dots, s_n, s ++ t, t_1, \dots, t_m]}
 \end{aligned}$$

The semantics of the traditional and arbitrary-choice PPLs are then the restricted semantics of  $\Sigma_{\text{all}}$  that only allows constructs from the traditional and the arbitrary-choice PPLs, respectively. Throughout this section, we will assume that any PPL is similarly a sublanguage of  $\Sigma_{\text{all}}$ , whose semantics is well-defined and consistent with  $\Sigma_{\text{all}}$ .

The extended semantics are still deterministic and total. However, it is worth noting that there are many different ways to specify rules in a way that is consistent with the intended semantics of the arbitrary-choice PPL. For instance, an invariant in the arbitrary-choice PPL is that  $c = i$  throughout the rendering process. As a result, we could substitute the VERTCONCATNOFLATTEN rule with  $\text{VERTCONCATNOFLATTEN}^*$ , which is like VERTCONCATNOFLATTEN but with a modification to use  $\langle \bar{d}_b, c, c, \perp \rangle \Downarrow_{\mathcal{R}} [t_1, \dots, t_m]$  as a premise instead of  $\langle \bar{d}_b, i, i, \perp \rangle \Downarrow_{\mathcal{R}} [t_1, \dots, t_m]$ , without changing the semantics of the arbitrary-choice PPL. This change could affect the semantics of  $\Sigma_{\text{all}}$  and subsequent theorems in this section. We pick VERTCONCATNOFLATTEN over  $\text{VERTCONCATNOFLATTEN}^*$  because it seemingly integrates better with other features in  $\Sigma_{\text{all}}$ .

## 4.2 Functional completeness

In [Section 1](#), we claimed that the traditional PPL cannot express the two layouts in [Figure 1b](#), as one layout has an extra pair of parentheses. The question that we may want to ask in general then is, given a PPL  $\Sigma$  and a non-empty set of layouts  $L$ , is it possible to construct a document in  $\Sigma$  that evaluates to  $L$ ? This motivates us to define the notion of functional completeness for PPLs.

*Definition 4.1.* A PPL  $\Sigma$  with an evaluation function  $\text{eval}(\cdot)$  is *functionally complete* if for any non-empty set of layouts  $L$ , there exists a document  $d$  in  $\Sigma$  such that  $\text{eval}(d) = L$ .

With this definition, we can formally reason about some PPLs that we have previously seen.

LEMMA 4.2. *The arbitrary-choice and  $\Sigma_e$  PPLs are functionally complete.*

LEMMA 4.3. *The traditional PPL is not functionally complete.*

LEMMA 4.4. *For each construct  $F$  in  $\{\text{text}, \langle \$ \rangle, \text{nl}, \langle | \rangle\}$ ,  $\Sigma_e$  without  $F$  is not functionally complete.*

If we limit the notion of expressiveness to only functional completeness, then all functionally complete PPLs would be equally expressive. However, intuitively this is clearly not the case. The proof of [Lemma 4.2](#) (in [Appendix B](#)) shows that it suffices for a PPL to only have `text`, `<$>`, and `<|>` for functional completeness, yet such a PPL would not be pleasant to use compared to  $\Sigma_e$ , because of the lack of features to, e.g., adjust indentation level. In a sense, functional completeness for PPLs is similar to Turing completeness for programming languages, which similarly does not fully capture expressiveness for programming languages. The next subsection presents a more fine-grained notion of expressiveness, based on the ability to define features.

## 4.3 Definability

The proof of [Lemma 4.2](#) shows that while  $\Sigma_e$  doesn't have `<$>`, we can simply expand  $d_a \langle \$ \rangle d_b$  to  $d_a \langle \$ \rangle \text{nl} \langle \$ \rangle d_b$ , which are in  $\Sigma_e$ , to perform the same functionality. In other words, the construct `<$>` is already *definable* by `<$>` and `nl`. Thus, adding `<$>` to  $\Sigma_e$  doesn't increase its expressiveness. In contrast, `<$>` is not definable by any combination of features in the arbitrary-choice PPL. To achieve the same functionality of `<$>`, it would require a non-local restructuring of the document, making it difficult to construct the document in natural way. In this sense, the inability to define a construct in a PPL means that adding the construct to the PPL increases its expressiveness.

More concretely, consider the arbitrary-choice document in [Figure 1b](#). The document is awkwardly constructed. The `return` keyword must be distributed to combine with a first line of the returned expression, due to the undefinability of unaligned concatenation. This creates a disconnection between the document structure and the underlying AST structure, making it more tedious and error-prone to construct documents. In contrast, the following document is a rewrite of [Figure 1b](#) to utilize the full expressiveness of  $\Sigma_e$  in a natural way. The sub-document colored blue fully corresponds to the “returned expression,” allowing users to recursively construct documents naturally.

```
text "function_append(first,second,third){\" <$> nest 4 (
  let f = text \"first_+\" in let s = text \"second_+\" in let t = text \"third\" in
  nl <$> text \"return_\" <$>
  ((text \"(\" <$> (nest 4 (nl <$> (f <$> nl <$> s <$> nl <$> t))) <$> nl <$> text \")\") <|>
  let sp = text \"_\" in (f <$> sp <$> s <$> sp <$> t))
) <$> nl <$> text \"}\"
```

The notion of definability (also known as expressibility) for programming languages was first developed by [Felleisen \[1991\]](#), and we adapt it for PPLs through a series of definitions as follows:

*Definition 4.5.* A PPL  $\Sigma$  consists of:

- a set of constructs  $\Sigma = \{F_1, F_2, \dots\}$ . There could potentially be infinite constructs. Each construct may have different arity and sort, whose arguments may also have different sort.
- a non-empty set of documents  $\mathcal{D}$ , which are of sort Doc, generated from  $\Sigma$ .
- an evaluation function  $\text{eval} : \mathcal{D} \rightarrow 2^{\mathcal{L}}$ .

*Example 4.6.*  $\Sigma_e$  contains `nest`  $\square_{\mathbb{N}} \square_{\text{Doc}}$ , which is a construct with arity 2 of sort Doc. The first argument to `nest` has sort  $\mathbb{N}$  and the second argument has sort Doc.  $\Sigma_e$  also contains all natural numbers and strings with no newline, which are constructs with arity 0 of sort  $\mathbb{N}$  and Str respectively. The evaluation function for  $\Sigma_e$  is  $\text{eval}_e$  from Section 3.3.

Henceforth, unless indicated otherwise,  $\mathcal{D}_X$  and  $\text{eval}_X$  are the set of documents and the evaluation function for the PPL  $\Sigma_X$ .

*Definition 4.7.* A *syntactic abstraction*  $M(\alpha_1, \dots, \alpha_n)$  of arity  $n$  for a PPL  $\Sigma$  is a document in  $\Sigma \cup \{\alpha_1, \dots, \alpha_n\}$  where  $\alpha_1, \dots, \alpha_n$  are metavariables (nullary constructors) of some sorts. An *instance*  $M(e_1, \dots, e_n)$  is a document in  $\Sigma$  that substitutes  $\alpha_i$  with  $e_i$  in  $M(\alpha_1, \dots, \alpha_n)$  for all  $1 \leq i \leq n$ , where  $e_i$  and  $\alpha_i$  must have a compatible sort.

*Example 4.8.*  $M(\alpha_1, \alpha_2) = \alpha_1 \triangleleft \text{nl} \triangleleft \alpha_2$  is a syntactic abstraction for  $\Sigma_e$ , where  $\alpha_1$  and  $\alpha_2$  have sort Doc. On the other hand,  $M'(\alpha_1) = \text{nest } \alpha_1 \text{ nl} \triangleleft \alpha_1$  is **not** a syntactic abstraction because the first occurrence of  $\alpha_1$  requires it to have sort  $\mathbb{N}$ , but the second occurrence requires it to have sort Doc. An instance  $M(\text{text "a"}, \text{text "b"})$  is the document `text "a"  $\triangleleft$  nl  $\triangleleft$  text "b"`, but  $M(\text{text "a"}, 1)$  is not an instance due to the incompatible sort.

*Definition 4.9.* Let  $\Sigma_{\text{base}}$  be a PPL and  $\Sigma_{\text{extended}} = \Sigma_{\text{base}} \cup \{F\}$  where  $F$  has sort Doc. A *syntactic expansion*  $\text{expand}_F^M(d)$  from  $\Sigma_{\text{extended}}$  to  $\Sigma_{\text{base}}$  is a function from  $\mathcal{D}_{\text{extended}}$  to  $\mathcal{D}_{\text{base}}$  that replaces every occurrence of  $F(e_1, \dots, e_n)$  with an instance  $M(e_1, \dots, e_n)$  in  $d$ , where  $F$  and  $M$  must have compatible arity and sort arguments.

*Example 4.10.*  $\text{expand}_{\triangleleft}^M(\cdot)$  is a syntactic expansion from  $\Sigma_e \cup \{\triangleleft\}$  to  $\Sigma_e$ , where  $M$  is from Example 4.8. Hence,  $\text{expand}_{\triangleleft}^M(\text{text "a"} \triangleleft \text{text "b"}) = \text{text "a"} \triangleleft \text{nl} \triangleleft \text{text "b"}$ .

We are now ready to define definability.<sup>5</sup>

*Definition 4.11.* Let  $\Sigma_{\text{base}}$  be a PPL and  $\Sigma_{\text{extended}} = \Sigma_{\text{base}} \cup \{F\}$ . We say that  $\Sigma_{\text{base}}$  can *define*  $F$  (alternatively,  $F$  is *definable* by  $\Sigma_{\text{base}}$ ) if there exists a syntactic abstraction  $M$  from  $\Sigma_{\text{extended}}$  to  $\Sigma_{\text{base}}$  such that for every document  $d \in \mathcal{D}_{\text{extended}}$ ,  $\text{eval}_{\text{extended}}(d) = \text{eval}_{\text{base}}(\text{expand}_F^M(d))$ .

We can now present one of our main results:

**THEOREM 4.12.**  $\Sigma_e$  can define every construct in the traditional and arbitrary-choice PPLs.

Despite the result, we might wonder if  $\Sigma_e$  is actually needed. Could it be that the arbitrary-choice PPL can already define every construct in the traditional PPL? As we forshadowed, the answer to this question is negative. However, we must first develop tools that allow us to answer the question, again following the development in Felleisen's work.

*Definition 4.13.* A context  $C(\alpha)$  for  $\Sigma$  is a unary syntactic abstraction for  $\Sigma$  where  $\alpha$  has sort Doc.

*Definition 4.14.* Given a PPL  $\Sigma$  and a relation  $R \subseteq 2^{\mathcal{L}} \times 2^{\mathcal{L}}$ ,  $E_R^\Sigma(d_1, d_2)$  is a relation that holds if and only if for all contexts  $C$  for  $\Sigma$ ,  $R(\text{eval}(C(d_1)), \text{eval}(C(d_2)))$  holds<sup>6</sup>.

<sup>5</sup>One important distinction of this definition and Felleisen's counterpart is that PPLs are *total*. Hence, observing the termination behavior, as done in Felleisen's work, is not feasible in our formulation.

<sup>6</sup>The relation  $E_R^\Sigma$  is a generalization of the operational equivalence relation in Felleisen's work.

*Example 4.15.* Let  $\text{width} : \mathcal{L} \rightarrow \mathbb{N}$  be a function that computes the maximum length across all lines in the input layout, and we lift  $\text{width}$  to work on any set of layouts. Furthermore, let  $R = \{(L_1, L_2) : \text{width}(L_1) = \text{width}(L_2)\}$ .

- $E_R^{\Sigma_e}(\text{text "a"}, \text{text "b"})$  holds by induction. Intuitively, this is because (1) if we only observe the width, the textual content doesn't matter, and (2) there is no construct in  $\Sigma_e$  that allows us to layout differently in a way that would affect the width based on the textual content.
- On the other hand,  $\neg E_R^{\Sigma_e}(\text{text "a"}, \text{text "aa"})$ . For example, with  $C(\alpha) = \alpha$ , we have that  $\text{width}(\text{eval}_e(C(\text{text "a"}))) = \{1\}$ , but  $\text{width}(\text{eval}_e(C(\text{text "aa"})))) = \{2\}$ .

The following theorem provides a tool to prove that a construct is not definable in a PPL.

**THEOREM 4.16.** *Given a PPL  $\Sigma$  and a construct  $F$ , if there exists two documents  $d_1$  and  $d_2$  in  $\Sigma$  and a relation  $R$  such that  $E_R^{\Sigma}(d_1, d_2)$ , but  $\neg E_R^{\Sigma \cup \{F\}}(d_1, d_2)$ , then  $F$  is not definable in  $\Sigma$ .*

With this tool, we are now able to prove the following:

**THEOREM 4.17.** *The following is true:*

- $\langle \rangle$  is not definable in the arbitrary-choice PPL.
- **nest** is not definable in the arbitrary-choice PPL.
- **group** is not definable in the arbitrary-choice PPL.
- $\langle + \rangle$  is not definable in the traditional PPL.

Next, we show a relationship between functional completeness and definability.

**LEMMA 4.18.** *If  $\Sigma$  is not functionally complete, but  $\Sigma \cup \{C\}$  is, then  $C$  is not definable in  $\Sigma$ .*

We now present our last result for this section:  $\Sigma_e$  is *minimal* in a sense that each of its constructs is not definable by  $\Sigma_e$  without it. This is what the design of  $\Sigma_e$  strives to achieve, so that the size of primitive constructs is small.

**THEOREM 4.19.** *For any construct  $F$  of  $\Sigma_e$ ,  $F$  is not definable in  $\Sigma_e \setminus \{F\}$ .*

## 5 OUR PRINTER, $\Pi_e$

In this section, we describe our printer,  $\Pi_e$ , which targets the PPL  $\Sigma_e$  presented in Section 3.  $\Pi_e$  is parameterized by a *cost factory*, allowing users to customize the optimality objective within the resource budget that they find acceptable. We start with an overview of  $\Pi_e$ . Then, we describe the cost factory interface. Next, we define *measure*, which is an output from the core printer that allows us to record a cost and at the same time avoid a full-blown, expensive rendering. After that, we describe the requirements of the input document structure, which will become important when we analyze the complexity of the printer. Then, we present the  $\Pi_e$ 's printing algorithm, which utilizes the cost factory to achieve an optimal and efficient printing. Finally, we show our analysis of  $\Pi_e$ .

### 5.1 Overview

So far we have defined the *evaluation* of a document, which produces the set of possible layouts. But when we *print* a document, we wish to output only a single, most optimal layout.

A naïve approach is to evaluate the input document, via widening and rendering, to all possible layouts, determine costs of these layouts according to a given optimality objective, and then pick one with the lowest cost. However, this approach is not practical for two reasons. First, widening could produce exponentially many choiceless documents. Second, rendering non-optimal choiceless documents is unnecessary and wasteful.

A better approach would avoid rendering until an optimal choiceless document is identified, and utilize early pruning to reduce the search space. Since we wish to avoid full-blown rendering, we will

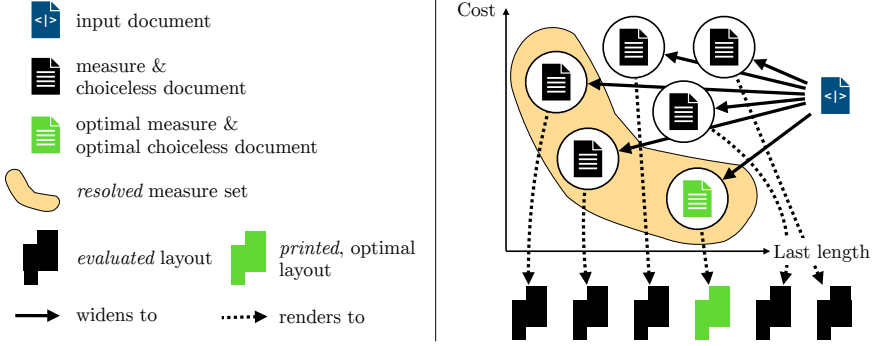


Fig. 7.  $\Pi_e$  resolves the document to a set of measures, and chooses an optimal measure from it to render.

Cost type $\tau$	
$\leq_{\mathcal{F}} : \tau \rightarrow \tau \rightarrow \mathbb{B}$	$\leq_{\mathcal{F}}$ must be a total ordering (transitive, antisymmetric, and total)
$+\mathcal{F} : \tau \rightarrow \tau \rightarrow \tau$	$\forall C_1, C_2, C_3, C_4 \in \tau. [C_1 \leq_{\mathcal{F}} C_2 \rightarrow C_3 \leq_{\mathcal{F}} C_4 \rightarrow C_1 +_{\mathcal{F}} C_3 \leq_{\mathcal{F}} C_2 +_{\mathcal{F}} C_4]$
$\text{text}_{\mathcal{F}} : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \tau$	$\forall c, c', l \in \mathbb{N}. [c \leq c' \rightarrow \text{text}_{\mathcal{F}}(c, l) \leq_{\mathcal{F}} \text{text}_{\mathcal{F}}(c', l)]$
$\text{newline}_{\mathcal{F}} : \mathbb{N} \rightarrow \tau$	$\forall i, i' \in \mathbb{N}. [i \leq i' \rightarrow \text{newline}_{\mathcal{F}}(i) \leq_{\mathcal{F}} \text{newline}_{\mathcal{F}}(i')]$
$W_{\mathcal{F}} : \mathbb{N}$	$W_{\mathcal{F}}$ has no contract

Fig. 8. The cost factory interface. Users need to supply the cost type  $\tau$  and implement the operations satisfying the contracts indicated in the interface.

instead operate on *measures* [Bernardy 2017c], which are limited views of layouts. This allows us to record the cost of a layout without expensive rendering. The need to prune early motivates us to devise the notion of *cost factory*, which allows us to incrementally compute costs to be used for pruning decision. At the same time, we design the cost factory to support a variety of optimality objectives.

The workflow of  $\Pi_e$  is shown in Figure 7. The printer first resolves choices, with early pruning, to produce a small set of measures that contain the optimal measure. We could then pick the optimal measure from the set and render its choiceless document to produce an optimal layout.

## 5.2 The cost factory

The cost factory interface is presented in Figure 8. It allows users to define a cost type  $\tau$  and implement operations on the cost type.  $C_1 \leq_{\mathcal{F}} C_2$  tests if the cost  $C_1$  is less than or equal to  $C_2$ .  $C_1 +_{\mathcal{F}} C_2$  adds the cost  $C_1$  and  $C_2$  together.  $\text{text}_{\mathcal{F}}(c, l)$  computes a cost due to a placement of a string without a newline of length  $l$  at the column position  $c$ .  $\text{newline}_{\mathcal{F}}(i)$  is the cost due to a newline with  $i$  indentation spaces. The implemented operations must satisfy the indicated contracts in the interface, which are required for the core printer to function correctly. However, in practice, more constraints are recommended so that the printer is not brittle in presence of rewriting rules. We discuss more about this topic in Appendix C. Lastly,  $W_{\mathcal{F}}$  is the computation width limit. When printing a document  $d$ ,  $\Pi_e$  only provides the optimality guarantee among layouts in  $\text{eval}_e(d)$  whose column position or indentation level during the printing does not exceeds  $W_{\mathcal{F}}$ . To illustrate how a cost factory is used in  $\Pi_e$ , we show a printing of the document in Figure 9 with a concrete cost factory.

*Example 5.1.* Consider an optimality objective that minimizes the sum of *overflows*, which are the numbers of characters that exceed a given *page width limit*  $w$  in each line, and *then* minimizes the *height*, which is the total number of newline characters (or equivalently, the number of lines minus one). This objective is thus able to avoid the excessive overflow problem in Bernardy’s printer described in Section 2.



```

687 let args = nl <> text "arg1," <> nl <> text "arg2"
688 in
689 text "func(" <>
690 (group (nest 2 args <> nl)) <>
691 text ")")

```

```

1 func( arg1, arg2 )
1 func(
2   arg1,
3   arg2
4 )

```

Fig. 9. An example document to illustrate how the cost factory computes a cost. The dotted lines show the width limit of 8 and 10.

More formally, given a layout, a cost is a pair of the overflow sum and the height, where the lexicographic order determines which cost is less. With  $w = 8$ , the first layout in Figure 9 has the cost  $(10, 0)$ , whereas the second layout has the cost  $(0, 3)$ . Thus, the second layout is the optimal layout that  $\Pi_e$  should pick.

We implement the optimality objective with the following cost factory  $\mathcal{F}$ .

$$\tau = \mathbb{N} \times \mathbb{N} \quad \leq_{\mathcal{F}} = \leq_{\text{lex}} \quad (o_a, h_a) +_{\mathcal{F}} (o_b, h_b) = (o_a + o_b, h_a + h_b)$$

$$\text{text}_{\mathcal{F}}(c, l) = (\max(c + l - \max(w, c), 0), 0) \quad \text{newline}_{\mathcal{F}}(i) = (\max(i - w, 0), 1)$$

Each text and newline placement would query the cost factory to compute its cost. For example, "func(" of length 5 is placed at the column position 0 in both layouts, so the cost is  $\text{text}_{\mathcal{F}}(0, 5) = (0, 0)$ . In the first layout, "arg1," of length 5 is placed at the column position 6, so the cost is  $\text{text}_{\mathcal{F}}(6, 5) = (3, 0)$ . On the other hand, in the second layout, "arg1," is placed at the column position 2, so the cost is  $\text{text}_{\mathcal{F}}(2, 5) = (0, 0)$ . Spaces due to newlines when the flattening mode is on are considered text.  $\text{newline}_{\mathcal{F}}$  is never used for the first layout, but the second layout would query  $\text{newline}_{\mathcal{F}}$  twice with indentation level 2 and once with indentation level 0. When we sum the costs up using  $+_{\mathcal{F}}$ , we obtain that the first layout has the cost  $(10, 0)$ , whereas the second layout has the cost  $(0, 3)$  as expected.  $\leq_{\mathcal{F}}$  could then be used to conclude that the second layout is optimal.

We have been ignoring the computation width limit  $W_{\mathcal{F}}$  until now. When a column position or indentation level exceeds the computation width limit during a printing, the result is *tainted*. For example, with  $W_{\mathcal{F}} = 10$ , the first layout would be tainted, while the second layout would not. Tainted layouts can usually be discarded right away, except when every possible layout is tainted. In such case,  $\Pi_e$  keeps one tainted layout so that it can still output a layout, but provide no guarantee that the layout will be optimal. The tainting system allows us to bound the computation so that the algorithm is efficient.

The cost factory interface is versatile. The above example shows that  $\Pi_e$  does not need to take a page width limit as an input, because the concept of page width limit can already be defined by users via  $\text{text}_{\mathcal{F}}$ . It is also possible to, for example, implement the concept of *soft* width limit, compute a linear combination of height and overflow in the style of Yelland [2016], compute a sum of squared overflow, and compute a maximum of overflow. Furthermore, the cost factory plays a central role that enables  $\Pi_e$  to be efficient. In the subsequent subsections, every definition is implicitly parameterized by a cost factory  $\mathcal{F}$ .

### 5.3 Measure

As presented earlier, the resolving phase computes *measures*. Each measure is a limited view of a choiceless document rendering that incorporates cost. Presented in Figure 10, a measure consists of five components: length of last line ( $l$ ), cost ( $C$ ), choiceless document ( $d$ ), max column position ( $x$ ), and max indentation ( $y$ ). We gray out the last two components because they are ghosted [Owicki and Gries 1976], only needed for the correctness theorem, and not required in the actual implementation.

$$\begin{aligned}
& \text{Measure } m \in \mathcal{M} = \langle l, C, \bar{d}, x, y \rangle_{\mathcal{M}} \\
& \text{last} : \mathcal{M} \rightarrow \mathbb{N} \quad \text{last}(\langle l, C, \bar{d}, x, y \rangle_{\mathcal{M}}) = l \quad \text{cost} : \mathcal{M} \rightarrow \tau \quad \text{cost}(\langle l, C, \bar{d}, x, y \rangle_{\mathcal{M}}) = c \\
& \text{doc} : \mathcal{M} \rightarrow \mathcal{D}_e \quad \text{doc}(\langle l, C, \bar{d}, x, y \rangle_{\mathcal{M}}) = \bar{d} \\
& \text{maxx} : \mathcal{M} \rightarrow \mathbb{N} \quad \text{maxx}(\langle l, C, \bar{d}, x, y \rangle_{\mathcal{M}}) = x \quad \text{maxy} : \mathcal{M} \rightarrow \mathbb{N} \quad \text{maxy}(\langle l, C, \bar{d}, x, y \rangle_{\mathcal{M}}) = y \\
& \circ : \mathcal{M} \rightarrow \mathcal{M} \rightarrow \mathcal{M} \quad \langle l_a, C_a, \bar{d}_a, x_a, y_a \rangle_{\mathcal{M}} \circ \langle l_b, C_b, \bar{d}_b, x_b, y_b \rangle_{\mathcal{M}} = \\
& \quad \langle l_b, C_a +_{\mathcal{F}} C_b, \bar{d}_a \diamond \bar{d}_b, \max(x_a, x_b), \max(y_a, y_b) \rangle_{\mathcal{M}} \\
& \text{adjustNest} : \mathbb{N} \rightarrow \mathcal{M} \rightarrow \mathcal{M} \quad \text{adjustNest}(n, \langle l, C, \bar{d}, x, y \rangle_{\mathcal{M}}) = \langle l, C, \text{nest } n \bar{d}, x, y \rangle_{\mathcal{M}} \\
& \text{adjustAlign} : \mathbb{N} \rightarrow \mathcal{M} \rightarrow \mathcal{M} \quad \text{adjustAlign}(i, \langle l, C, \bar{d}, x, y \rangle_{\mathcal{M}}) = \langle l, C, \text{align } \bar{d}, x, \max(y, i) \rangle_{\mathcal{M}} \\
& \leq : \mathcal{M} \rightarrow \mathcal{M} \rightarrow \mathbb{B} \quad \langle l_a, C_a, \bar{d}_a, x, y \rangle_{\mathcal{M}} \leq \langle l_b, C_b, \bar{d}_b, x, y \rangle_{\mathcal{M}} = l_a \leq l_b \wedge C_a \leq_{\mathcal{F}} C_b
\end{aligned}$$

Fig. 10. Measure and operations on measures

$$\begin{aligned}
& \text{TEXTM} \frac{}{\langle \text{text } s, c, i \rangle \Downarrow_{\mathcal{M}} \langle c + |s|, \text{text}_{\mathcal{F}}(c, |s|), \text{text } s, c + |s|, i \rangle_{\mathcal{M}}} \\
& \text{LINEM} \frac{}{\langle \text{nl}, c, i \rangle \Downarrow_{\mathcal{M}} \langle i, \text{newline}_{\mathcal{F}}(i), \text{nl}, \max(c, i), i \rangle_{\mathcal{M}}} \quad \text{CONCATM} \frac{\langle \bar{d}_a, c, i \rangle \Downarrow_{\mathcal{M}} m_a \quad \langle \bar{d}_b, \text{last}(m_a), i \rangle \Downarrow_{\mathcal{M}} m_b}{\langle \bar{d}_a \diamond \bar{d}_b, c, i \rangle \Downarrow_{\mathcal{M}} m_a \circ m_b} \\
& \text{NESTM} \frac{\langle \bar{d}, c, i + n \rangle \Downarrow_{\mathcal{M}} m}{\langle \text{nest } n \bar{d}, c, i \rangle \Downarrow_{\mathcal{M}} \text{adjustNest}(n, m)} \quad \text{ALIGNM} \frac{\langle \bar{d}, c, c \rangle \Downarrow_{\mathcal{M}} m}{\langle \text{align } \bar{d}, c, i \rangle \Downarrow_{\mathcal{M}} \text{adjustAlign}(i, m)}
\end{aligned}$$

Fig. 11. Measure computation from a choiceless document printing state

*Example 5.2.* Given  $\bar{d} = \text{text } "\_=\{|" \diamond \text{nest } 1 (\text{nl} \diamond \text{text } "1234") \diamond \text{nl} \diamond \text{text } "|}"$  being placed at the column position 0, with indentation level 0. The choiceless document would render to a layout  $["\_=\{|", "\_1234", "|"]$ . With the cost factory in [Example 5.1](#), the cost of the layout is 0. Thus, the measure is  $\langle 2, 0, \bar{d}, 5, 1 \rangle_{\mathcal{M}}$ .

[Figure 11](#) shows rules that define measure computation. The judgment  $\langle \bar{d}, c, i \rangle \Downarrow_{\mathcal{M}} m$  states that when we compute the measure of  $\bar{d} \in \mathcal{D}_e$  placed at the column position  $c \in \mathbb{N}$  with indentation level  $i \in \mathbb{N}$ , the resulting measure is  $m \in \mathcal{M}$ . To simplify the core printer, we (temporarily) remove **flatten** from  $\Sigma_e$ . This allows us to eliminate the flattening mode parameter, which implicitly defaults to  $\perp$ . Toward the end of this section, we will show how to add support for **flatten** back.

The rules are largely standard. They reflect the actual rendering defined by  $\Downarrow_{\mathcal{R}}$ , and utilize the cost factory in a straightforward way. The rules use a helper operator function  $\circ$  to concatenate two measures, and helper functions **adjustNest** and **adjustAlign** to construct a correct measure for **nest** and **align**. These functions are defined in [Figure 10](#). Notably, the **LINEM** rule creates a measure whose **maxc** is  $\max(c, i)$  because before placing the newline, the column position is  $c$ , and after placing the newline, the column position is  $i$ . The **ALIGNM** rule creates a measure whose **maxi** is  $\max(y, i)$  where  $y$  is obtained via the recursive computation. This is because the recursive computation discards the current indentation level, so we need to specifically record the information.

$\Downarrow_{\mathcal{M}}$  is deterministic and total. It is also correct with respect to  $\Downarrow_{\mathcal{R}}$ .

**THEOREM 5.3.** *For any  $\bar{d} \in \mathcal{D}_e$  and  $c, i \in \mathbb{N}$ , there exists a cost  $C$  and max indentation  $y$  such that*

- *if  $\langle \bar{d}, c, i, \perp \rangle \Downarrow_{\mathcal{R}} [s]$ , then  $\langle \bar{d}, c, i \rangle \Downarrow_{\mathcal{M}} \langle c + |s|, C, \bar{d}, c + |s|, y \rangle_{\mathcal{M}}$ .*
- *if  $\langle \bar{d}, c, i, \perp \rangle \Downarrow_{\mathcal{R}} [s, s_1, \dots, s_n, t]$ , then  $\langle \bar{d}, c, i \rangle \Downarrow_{\mathcal{M}} \langle |t|, C, \bar{d}, \max(c + |s|, |s_1|, \dots, |s_n|, |t|), y \rangle_{\mathcal{M}}$ .*

So far, we only consider the measure computation for a choiceless document. When we take the choice operator into account, there could be multiple measures under the same printing context.

Measure set $S \in \mathcal{S} ::=$	$\text{Tainted}(\hat{m})$	where $\hat{m}$ is a promise that can be forced to a measure
	$  \text{Set}([m_1, \dots, m_n])$	where $\text{last}(m_1) > \dots > \text{last}(m_n)$ and $\forall i \neq j, \neg(m_i \leq m_j \vee m_j \leq m_i)$
$\text{taint} : \mathcal{S} \rightarrow \mathcal{S}$	$\text{taint}(\text{Tainted}(m)) = \text{Tainted}(m)$	
	$\text{taint}(\text{Set}([m_0, m_1, \dots, m_n])) = \text{Tainted}(m_0)$	
$\text{lift} : \mathcal{S} \rightarrow (\mathcal{M} \rightarrow \mathcal{M}) \rightarrow \mathcal{S}$	$\text{lift}(\text{Tainted}(m), f) = \text{Tainted}(f(m))$	
	$\text{lift}(\text{Set}([m_1, \dots, m_n]), f) = \text{Set}([f(m_1), \dots, f(m_n)])$	
$\text{dedup} : \vec{\mathcal{M}} \rightarrow \vec{\mathcal{M}}$	$\text{dedup}([m, m', m_1, \dots, m_n]) = \text{dedup}([m', m_1, \dots, m_n])$	if $m' \leq m$
	$\text{dedup}([m, m', m_1, \dots, m_n]) = [m] @ \text{dedup}([m', m_1, \dots, m_n])$	if $m' \not\leq m$
	$\text{dedup}([m]) = [m]$	
$\uplus : \mathcal{S} \rightarrow \mathcal{S} \rightarrow \mathcal{S}$	$S \uplus \text{Tainted}(m) = S$	
	$\text{Tainted}(m) \uplus \text{Set}([m_1, \dots, m_n]) = \text{Set}([m_1, \dots, m_n])$	
	$\text{Set}([m_1, \dots, m_n]) \uplus \text{Set}([m'_1, \dots, m'_{n'}]) = \text{Set}([m_1, \dots, m_n] \uplus [m'_1, \dots, m'_{n'}])$	
$\uplus : \vec{\mathcal{M}} \rightarrow \vec{\mathcal{M}} \rightarrow \vec{\mathcal{M}}$	$[] \uplus [m_1, \dots, m_n] = [m_1, \dots, m_n]$	
	$[m_1, \dots, m_n] \uplus [] = [m_1, \dots, m_n]$	
$[m_0, m_1, \dots, m_n] \uplus [m'_0, m'_1, \dots, m'_{n'}] =$	$\begin{cases} [m_0, m_1, \dots, m_n] \uplus [m'_1, \dots, m'_{n'}] & \text{if } m_0 \leq m'_0 \\ [m_1, \dots, m_n] \uplus [m'_0, m'_1, \dots, m'_{n'}] & \text{if } m'_0 \leq m_0 \\ [m_0] @ ([m_1, \dots, m_n] \uplus [m'_0, m'_1, \dots, m'_{n'}]) & \text{if } \text{last}(m_0) > \text{last}(m'_0) \\ [m'_0] @ ([m_0, m_1, \dots, m_n] \uplus [m'_1, \dots, m'_{n'}]) & \text{otherwise} \end{cases}$	

Fig. 12. Measure set and the merge operation on measure sets. @ denotes a list concatenation. We treat a promise  $\hat{m}$  and a measure  $m$  interchangeably, as they can be straightforwardly casted to each other.

The main operation that we can perform on these measures is finding domination  $\leq$ , also presented in Figure 10.  $m_a \leq m_b$  when both the cost and the last length of  $m_a$  are no worse than those of  $m_b$ . The fact that  $m_a \leq m_b$  is useful because it allows us to prune  $m_b$  away immediately.

## 5.4 Measure set

Resolving a document (in a printing context) produces a set of measures. To accommodate taintedness mentioned in Section 5.2, Figure 12 defines a measure set to be either a non-empty Set of untainted measures where no measure dominates the other, or a Tainted singleton set of a promise  $\hat{m}$  that can be forced to a measure. The Set, by definition, forms a Pareto frontier. To aid computation, we represent the Set with a list ordered by the cost in the strict ascending order (and therefore the last length in the strict descending order). We are able to do so because in a Pareto frontier, all last and cost values must be distinct.

The main operation that we can perform on measure sets is merging two measure sets ( $\uplus$ ), shown in Figure 12, where we prefer a Set over a Tainted. The merge operation maintains the Pareto frontier invariant, by doing the merge in the style of the merge operation in merge sort, although the Pareto frontier merging can also prune measures away during the operation. One important “quirk” of this merge operation is that it is *left-biased* in presence of taintedness. If two tainted measure sets are merged, the result is always the left one. This means the order of arguments to the merge operation is important, as we will see in the next subsections.

Other operations on measure sets which are used in next subsections are taint, lift, and dedup. taint taints a measure set. When tainting a Set, we choose to pick the first measure from the Set because it has the lowest cost, which is a greedy heuristic. lift adjusts measures in a measure set. Lastly, dedup prunes measures that are sorted by last in the strict decreasing order and by cost in the non-strict increasing order, so that the result conforms the Pareto frontier invariant.

## 5.5 The document structure

Section 2.2 has shown that we need to handle document sharing by treating the input document as a DAG. However, documents cannot be arbitrarily shared, as the following example shows:

*Example 5.4.* The following document  $\text{mk}(n)$  has the DAG size of  $O(n)$ . However, resolving it necessitates  $O(2^n)$  units of computation, as the printing contexts are all different. This is bad news because it means resolving could take exponential time in the input size.

```
let rec mk (n : int): doc =
  if n = 0 then text "x" else let shared = mk (n - 1) in shared <> shared
```

However, we argue that the above document is not *properly shared*, because the sub-documents are not shared *across choices*, which is how sharing is employed in practice. The corresponding properly shared document should have  $O(2^n)$  DAG size, so  $O(2^n)$  units of computation are still linear in the input size. To make this precise, we provide the following definitions:

*Definition 5.5.* Given a document  $d \in \mathcal{D}_e$ ,  $G(d)$  is a DAG rooted at  $d$  whose edge in the graph connects a document to its direct subdocuments.

*Definition 5.6.* A document  $d \in \mathcal{D}_e$  is *properly shared* if for any two vertices  $d_a$  and  $d_b$  in  $G(d)$ , if  $p_1$  and  $p_2$  are two distinct paths from  $d_a$  to  $d_b$ , then there exists a common document  $d'$  such that (1)  $d'$  is a <> ; (2)  $d'$  occurs in both  $p_1$  and  $p_2$ ; and (3)  $d'$  is not  $d_b$ .

Figure 3c shows a properly shared document (assuming that  $D$  is properly shared). It illustrates two paths where  $d_a$  is the root node,  $d_b$  is  $D$ , and  $d'$  is  $d_a$ . In practice, non-properly shared documents can still be processed by  $\Pi_e$ , and in fact can even make resolving (but not rendering) faster when the documents are printed under the same printing context. However, they would be treated as different documents when they are shared in different contexts. For simplicity, we only consider properly shared documents as the input to  $\Pi_e$  in this paper.

## 5.6 The resolver

We now formally define the core of  $\Pi_e$ , which is the resolver. It is described in Figure 13, which is a fusion of widening in Figure 6 and measure computation in Figure 11, with early pruning inherent in the merge operation and extra bookkeeping for taintedness. The judgment  $\langle d, c, i \rangle \Downarrow_{\text{RS}} S$  states that a properly shared document  $d \in \mathcal{D}_e$  at a column position  $c \in \mathbb{N}$  with an indentation level  $i \in \mathbb{N}$  resolves to a measure set  $S$ .

*Resolving text.* If placing the text would exceed  $W_{\mathcal{F}}$  or the indentation level is beyond  $W_{\mathcal{F}}$ , the  $\text{TEXTSTNT}$  rule returns a Tainted. Otherwise, the  $\text{TEXTRS}$  rule returns a singleton Set.

*Resolving newline.* Resolving a **nl** is similar to resolving a **text**, but we only need to consider the current column position and indentation level, as resolving the newline does not change the column position. The  $\text{LINERSTNT}$  and  $\text{LINERS}$  rules cover these two cases.

*Resolving nest.* Resolving a **nest** is handled by the  $\text{NESTRS}$  rule, which recursively resolves its sub-document, with the indentation level changed. The recursive resolving would determine whether the measure set would be a Set or Tainted. In all cases, the result is adjusted to construct correct choiceless documents.

*Resolving alignment.* Resolving an **align** is similar to resolving **nest**. However, because the recursive resolving would discard the current indentation level, which could exceed  $W_{\mathcal{F}}$ , we need to taint the measure set when the indentation level is beyond  $W_{\mathcal{F}}$ . The  $\text{ALIGNRSTNT}$  rule handles such case, and the  $\text{ALIGNRS}$  rule handles other possibilities.

$$\begin{array}{c}
\text{883} \quad \text{TEXTRSSSET} \frac{c + |s| \leq W_{\mathcal{F}} \quad i \leq W_{\mathcal{F}} \quad \langle \text{text } s, c, i \rangle \Downarrow_{\mathbb{M}} m}{\langle \text{text } s, c, i \rangle \Downarrow_{\text{RS}} \text{Set}([m])} \quad \text{LINERSSET} \frac{c \leq W_{\mathcal{F}} \quad i \leq W_{\mathcal{F}} \quad \langle \text{nl}, c, i \rangle \Downarrow_{\mathbb{M}} m}{\langle \text{nl}, c, i \rangle \Downarrow_{\text{RS}} \text{Set}([m])} \\
\text{884} \\
\text{885} \\
\text{886} \quad \text{TEXTRSTNT} \frac{c + |s| > W_{\mathcal{F}} \vee i > W_{\mathcal{F}} \quad \langle \text{text } s, c, i \rangle \Downarrow_{\mathbb{M}} m}{\langle \text{text } s, c, i \rangle \Downarrow_{\text{RS}} \text{Tainted}(m)} \quad \text{LINERSTNT} \frac{c > W_{\mathcal{F}} \vee i > W_{\mathcal{F}} \quad \langle \text{nl}, c, i \rangle \Downarrow_{\mathbb{M}} m}{\langle \text{nl}, c, i \rangle \Downarrow_{\text{RS}} \text{Tainted}(m)} \\
\text{887} \\
\text{888} \\
\text{889} \quad \text{NESTRS} \frac{\langle d, c, i + n \rangle \Downarrow_{\text{RS}} S}{\langle \text{nest } n \, d, c, i \rangle \Downarrow_{\text{RS}} \text{lift}(S, \text{adjustNest}(n))} \quad \text{ALIGNRS} \frac{i \leq W_{\mathcal{F}} \quad \langle d, c, c \rangle \Downarrow_{\text{RS}} S}{\langle \text{align } d, c, i \rangle \Downarrow_{\text{RS}} \text{lift}(S, \text{adjustAlign}(i))} \\
\text{890} \\
\text{891} \quad \text{UNIONRS} \frac{\langle d_a, c, i \rangle \Downarrow_{\text{RS}} S_a \quad \langle d_b, c, i \rangle \Downarrow_{\text{RS}} S_b}{\langle d_a <|> d_b, c, i \rangle \Downarrow_{\text{RS}} S_a \uplus S_b} \quad \text{ALIGNRSTNT} \frac{i > W_{\mathcal{F}} \quad \langle d, c, c \rangle \Downarrow_{\text{RS}} S}{\langle \text{align } d, c, i \rangle \Downarrow_{\text{RS}} \text{lift}(\text{taint}(S), \text{adjustAlign}(i))} \\
\text{892} \\
\text{893} \\
\text{894} \quad \text{CONCATRS} \frac{\langle d_a, c, i \rangle \Downarrow_{\text{RS}} \text{Set}([m_1, \dots, m_n]) \quad \langle m_1, d_b, i \rangle \Downarrow_{\text{RSC}} S_1 \quad \dots \quad \langle m_n, d_b, i \rangle \Downarrow_{\text{RSC}} S_n}{\langle d_a \Diamond d_b, c, i \rangle \Downarrow_{\text{RS}} S_1 \uplus \dots \uplus S_n} \\
\text{895} \\
\text{896} \\
\text{897} \quad \text{CONCATRSTNT} \frac{\langle d_a, c, i \rangle \Downarrow_{\text{RS}} \text{Tainted}(m_a) \quad \langle d_b, \text{last}(m_a), i \rangle \Downarrow_{\text{RS}} S \quad \text{taint}(S) = \text{Tainted}(m_b)}{\langle d_a \Diamond d_b, c, i \rangle \Downarrow_{\text{RS}} \text{Tainted}(m_a \circ m_b)} \\
\text{898} \\
\text{899} \\
\text{900} \quad \text{RSCSET} \frac{\langle d_b, \text{last}(m_a), i \rangle \Downarrow_{\text{RS}} \text{Set}([m_1, \dots, m_n])}{\langle m_a, d_b, i \rangle \Downarrow_{\text{RSC}} \text{Set}(\text{dedup}([m_a \circ m_1, \dots, m_a \circ m_n]))} \quad \text{RSCTNT} \frac{\langle d_b, \text{last}(m_a), i \rangle \Downarrow_{\text{RS}} \text{Tainted}(m_b)}{\langle m_a, d_b, i \rangle \Downarrow_{\text{RSC}} \text{Tainted}(m_a \circ m_b)} \\
\text{901} \\
\text{902} \\
\text{903}
\end{array}$$

Fig. 13. The resolver

*Resolving choice.* The UNIONRS rule recursively resolves its two sub-documents, and then merges the measure sets. As mentioned in Section 5.4, the merge operation is left-biased. Therefore, the left sub-document will be preferred over the right sub-document if exceeding  $W_{\mathcal{F}}$  is unavoidable. It is possible to employ a heuristic to remove this bias, as discussed in Appendix C.

*Resolving unaligned concatenation.* Resolving a  $\Diamond$  is done through CONCATRSTNT and CONCATRS rules, which handle the two possibilities of measure set types obtained from the left sub-document's recursive resolving. Notably, the CONCATRS rule employs  $\Downarrow_{\text{RSC}}$  to help us concatenate a left measure from the left measure set with a right measure set. The series of merges should be done with the right fold ordering to avoid unnecessary list traversal.

$\Downarrow_{\text{RS}}$  is deterministic and total. This allows us to define the top-level printer as  $\Pi_e(d) = l$  where  $\langle \text{doc}(m_0), 0, 0, \perp \rangle \Downarrow_{\mathcal{R}} l$  and  $\langle d, 0, 0 \rangle \Downarrow_{\text{RS}} [m_0, m_1, \dots, m_n]$ , which consumes a properly shared document  $d$ , resolves it to a set of measures, picks the measure with the lowest cost, and simply renders the associated choiceless document to produce a layout (although our implementation further fuses resolving and rendering together, as described in Appendix C).

While the rules above are enough for correctness, implementing these rules requires further consideration. As we will see in Lemma 5.10, any resolving beyond  $W_{\mathcal{F}}$  would eventually result in a tainted measure set. Hence,  $\Pi_e$  should *immediately* delay the computation for any resolving beyond  $W_{\mathcal{F}}$ .  $\Pi_e$  should also *memoize* the computation, so that on identical document and printing context within  $W_{\mathcal{F}}$ , the result of the previous computation is reused.

We claim that  $\Pi_e(d)$  consumes a properly shared document  $d$  in  $\Sigma_e$  and produces an optimal layout among  $\text{eval}_e(d)$  within  $W_{\mathcal{F}}$ . We will prove the claim in the next subsection.

## 5.7 Correctness of $\Pi_e$

$\Downarrow_{\mathbb{P}}$  is correct with respect to  $\Downarrow_{\mathbb{M}}$ . Two theorems govern the correctness. The first theorem states that the core printer returns a measure set that contains a measure that is no worse than any measure within the computation width limit from all possible measures.

THEOREM 5.7 (OPTIMALITY). For any  $d \in \mathcal{D}_e$ ,  $c \in \mathbb{N}$ ,  $i \in \mathbb{N}$ , if the following conditions hold

- $\langle d, c, i \rangle \Downarrow_{\text{RS}} S$
- $\bar{d} \in \bar{D}$
- $\text{maxx}(m) \leq W_{\mathcal{F}}$
- $d \Downarrow_{\mathcal{W}} \bar{D}$
- $\langle \bar{d}, c, i \rangle \Downarrow_{\mathbb{M}} m$
- $\text{maxy}(m) \leq W_{\mathcal{F}}$

then  $S = \text{Set}([m_1, \dots, m_n])$ . Furthermore, there exists  $i$  such that  $m_i \leq m$ .

The second theorem states that measures in the resulting measure set are valid.

THEOREM 5.8 (VALIDITY). For any  $d \in \mathcal{D}_e$ ,  $c \in \mathbb{N}$ ,  $i \in \mathbb{N}$  with  $d \Downarrow_{\mathcal{W}} \bar{D}$ , if  $\langle d, c, i \rangle \Downarrow_{\text{RS}} \text{Set}([m_1, \dots, m_n])$ , then for each  $i$ , there exists  $\bar{d}$  such that  $\bar{d} \in \bar{D}$  and  $\langle \bar{d}, c, i \rangle \Downarrow_{\mathbb{M}} m_i$ . Likewise, if  $\langle d, c, i \rangle \Downarrow_{\text{RS}} \text{Tainted}(m_0)$ , then there exists  $\bar{d}$  such that  $\bar{d} \in \bar{D}$  and  $\langle \bar{d}, c, i \rangle \Downarrow_{\mathbb{M}} m_0$ .

The correctness of  $\Pi_e$  follows immediately.

While the above theorems guarantee the correctness of the result that the printer produces, it does not concern with efficiency. The following lemmas provide some properties of the printer that allow us to reason about the efficiency.

LEMMA 5.9. For any  $d \in \mathcal{D}_e$ ,  $c \leq W_{\mathcal{F}}$ ,  $i \leq W_{\mathcal{F}}$ , if  $\langle d, c, i \rangle \Downarrow_{\text{RS}} \text{Set}([m_1, \dots, m_n])$ , then  $n \leq W_{\mathcal{F}} + 1$ .

LEMMA 5.10. For any  $d \in \mathcal{D}_e$ , if  $c > W_{\mathcal{F}}$  or  $i > W_{\mathcal{F}}$  and  $\langle d, c, i \rangle \Downarrow_{\text{RS}} S$ , then  $S$  is a Tainted.

We now informally prove the efficiency of  $\Pi_e$  that we claimed in [Section 1](#). The proof sketches are provided in [Appendix B](#).

THEOREM 5.11. The time complexity of  $\Pi_e$  is  $O(nW_{\mathcal{F}}^4)$  where  $n$  is the DAG size of the document.

THEOREM 5.12. If a document  $d$  is in the arbitrary-choice PPL,  $\Pi_e$  can print  $d$  in  $O(nW_{\mathcal{F}}^3)$ .

## 5.8 Handling flattening

To support **flatten**, we make it a function that walks its sub-document and replaces all **nl** with **text** `"_"`. The walk is memoized and preserves the original identity of the document whenever possible (i.e. if nothing is flattened in sub-documents, then the document itself is returned unchanged without creating a new document). Thus, each document can be flattened at most once. This flattening creates at most  $O(n)$  new documents without destroying the shared structure in the original document. We therefore achieve the functionality of **flatten** without affecting the time complexity of the printer.

## 6 IMPLEMENTATION

We implement  $\Pi_e$  in OCaml and Racket. The printer, which we call SNOWWHITE, is further refined to be more efficient and practical. Due to the space limit, we describe these refinements in [Appendix C](#). The OCaml SNOWWHITE, as a reference implementation, is used for comparing against other printers in [Section 7](#). The Racket SNOWWHITE has even more features, and it has been used to implement the code formatter for the Racket programming language.

SNOWWHITE provides a pre-defined cost factory that minimizes the sum of squared overflows over the page width limit  $w$  (without considering indentation spaces) and then height. The text placement formula is derived from the identity  $(a+b)^2 - a^2 = b(2a+b)$  where in each text placement,  $a$  is the starting position count past the page width limit and  $b$  is the overflow length. With this cost factory and  $w = 8$ , the first layout in [Figure 9](#) has the cost  $(10^2, 0)$  whereas the second layout has the cost  $(0, 3)$ .

$$\text{text}_{\mathcal{F}}(c, l) = \begin{cases} (b(2a+b), 0) & \text{if } c+l > w \\ (0, 0) & \text{otherwise} \end{cases} \quad \text{where} \quad \begin{cases} a = \max(w, c) - w \\ b = c + l - \max(w, c) \end{cases}$$



## 7 EVALUATION

This evaluates the performance and optimality of SNOWWHITE. The evaluation consists of two parts. First, we compare SNOWWHITE against Wadler/Leijen [2000] and Bernardy [2017b]’s printers, which are popular practical printers with capabilities from the traditional and arbitrary-choice PPLs. Second, we evaluate the Racket code formatter, which uses SNOWWHITE as its foundation. The evaluation aims to answer the following questions:

- (1) Does SNOWWHITE run fast in practice?
- (2) Does SNOWWHITE produce pretty layouts in practice?

All experiments are performed on a (non-poisoned) Apple M1 MacBook Pro with 16GB of RAM. We describe the experiments and benchmarks in Section 7.1 and Section 7.2, and discuss the results in Section 7.3.

### 7.1 Comparison of printers

We compare OCaml SNOWWHITE against the latest version (1.2.1) of Wadler/Leijen’s printer, and the “camera ready version” of Bernardy’s printer<sup>7</sup>. This “camera ready version” consists of two printers: the “naïve” variant, which is presented in the paper, and the “practical” implementation, which has more features (such as unavoidable overflow handling) but suffers from the exponential time complexity when the DAG structure unfolds, as discussed in Section 2. We manually remove the capability to customize the width limit from the latter to avoid the issue. Both variants are used for the evaluation, since the naïve variant does not have necessary features for some benchmarks.

SNOWWHITE is instantiated with the cost factory in Section 6, with the page width limit of 80 (unless indicated otherwise). We run SNOWWHITE twice with different computation width limits (once with  $W_{\mathcal{F}} = 100$ , unless indicated otherwise, and once with  $W_{\mathcal{F}} = 1000$ ), in order to observe the effect of the tainting system and how it affects the performance.

The benchmarks (Table 2) are mostly taken from Bernardy [2017c], and we add a few more to test basic constructs. While Leijen’s printer is expressive enough to handle all benchmarks (due to the inclusion of `align` to support aligned concatenation in addition to constructs from the traditional PPL), Bernardy’s printers are not applicable to benchmarks that require constructs from the traditional PPL. Furthermore, Bernardy’s naïve printer is not applicable to benchmarks that require extra features like unavoidable overflow handling.

In more detail, the benchmarks test the following kinds of documents:

**Concat** benchmarks test a long chain of concatenations, which are identified by Peyton-Jones [1997] as a source of quadratic time complexity in Hughes’ printer.

**FillSep** benchmarks test the `fillSep` construct (also known as `fill`), which fills lines with words as long as they fit.

**Flatten** benchmarks test repeated flattening, as shown in Figure 15.

**SExpFull** benchmarks are the last two data points from the “full tree” benchmark in Bernardy [2017c]’s paper. They create complete binary trees and print them as S-expressions.

**RandFit** benchmarks [Bernardy 2017c] are similar to SExpFull, but use random Dyck paths to generate random trees and filter only those that can fit within the page width limit.

**RandOver** benchmarks are like RandFit with the opposite filtering.

**JSON** benchmarks are also from Bernardy [2017c]’s paper. They format large JSON files.

<sup>7</sup>We also tried other versions of Bernardy’s printer, such as the commit 006fa0e8, which is the version right before the `<|>` operator was removed, and supposedly more optimized than the camera ready version. Unfortunately, we find that it has a severe performance deficiency. When attempting to replicate the experiments in Bernardy [2017c], we find that formatting the 10k-line-JSON file takes about 80 seconds, which is much slower than the 145 milliseconds reported in the paper.

Table 2. Comparison between SNOWWHITE in different configurations and other printers. For each printer and configuration, the first column reports the running time, and the second column reports the line count of the output layout. SNOWWHITE has an additional third column, where ✓ indicates that the output layout fits  $W_{\mathcal{F}}$  and ✗ indicates that the output layout is tainted. “N/A” means the benchmark is not applicable. ⌚ indicates that running the benchmark exceeds the timeout of 60 seconds. “-” means the data is not collected. A grayed row indicates an output mismatch among the printers/configurations. The bolded line count signals that in our manual inspection, the associated layout is the prettiest.




Benchmark	SNOWWHITE						Wadler/Leijen		Bernardy			
	default $W_{\mathcal{F}}$ (usually 100)			$W_{\mathcal{F}} = 1000$					Naïve		Practical	
Concat10k	0.89 ms	1	✗	0.91 ms	1	✗	2.05 ms	1	N/A	-	548.14 ms	1
Concat50k	10.58 ms	1	✗	10.55 ms	1	✗	11.55 ms	1	N/A	-	17.18 s	1
FillSep5k	13.51 ms	668	✓	13.71 ms	668	✓	12.34 ms	668	3.32 s	668		-
FillSep50k	264.34 ms	6834	✓	262.99 ms	6834	✓	73.16 ms	6834				-
Flatten8k	42.10 ms	7986	✓	47.15 ms	7986	✓	3.33 s	7986	N/A	-	N/A	-
Flatten16k	94.86 ms	15986	✓	92.62 ms	15986	✓	22.14 s	15986	N/A	-	N/A	-
SExpFull15	3.28 s	4107	✓	8.55 s	4107	✓	60.44 ms	4107	678.92 ms	4107	929.26 ms	4107
SExpFull16	5.51 s	8246	✓	21.79 s	8246	✓	98.79 ms	8246	1.35 s	8246	1.83 s	8246
RandFit1k	130.97 ms	<b>629</b>	✓	243.59 ms	<b>629</b>	✓	15.16 ms	943	51.68 ms	<b>629</b>	80.17 ms	<b>629</b>
RandFit10k	1.13 s	<b>7861</b>	✓	4.66 s	<b>7861</b>	✓	42.65	10459	582.83 ms	<b>7861</b>	902.99 ms	<b>7861</b>
RandOver1k	88.06 ms	<b>1531</b>	✗	958.34 ms	<b>1531</b>	✓	7.27 ms	1635	N/A	-	73.95 ms	1105
RandOver10k	486.81 ms	<b>15027</b>	✗	13.98 s	<b>15027</b>	✓	135.75 ms	16015	N/A	-	1.18 s	7953
JSON1k	2.13 ms	564	✓	2.25 ms	564	✓	1.87 ms	564	N/A	-	5.02 ms	564
JSON10k	27.30 ms	5712	✓	25.86 ms	5712	✓	23.06 ms	5712	N/A	-	106.73 ms	5712
JSONW	2.15 ms	<b>721</b>	✗	2.20 ms	<b>721</b>	✓	8.52 ms	<b>721</b>	N/A	-	5.44 ms	709

Table 3. The code formatter benchmarks. The table is in the same format as the SNOWWHITE column in Table 2.

Benchmark	$W_{\mathcal{F}} = 100$		$W_{\mathcal{F}} = 1000$		Benchmark	$W_{\mathcal{F}} = 100$		$W_{\mathcal{F}} = 1000$	
class-internal	651 ms	5750	✗	665 ms	<b>5749</b>	✓	98 ms	993	✓
xform	796 ms	5154	✗	819 ms	5154	✓	8 ms	83	✓
list	98 ms	993	✓	90 ms	993	✓	9 ms	83	✓
hash	8 ms	83	✓	9 ms	83	✓			

JSONW benchmark is the same as JSON1k but with a page width limit of 50 instead of 80, and we further adjust SNOWWHITE’s default  $W_{\mathcal{F}}$  from 100 to 60 to test the tainting system.

## 7.2 Racket code formatter

We evaluate the effectiveness of a Racket code formatter that uses the Racket SNOWWHITE as its foundation. Racket [Felleisen et al. 2018] is a programmable programming language. Its main syntax is S-expression, but this can be customized via its #lang protocol to read an arbitrary syntax. Even in the S-expression syntax, users can define custom forms via the macro system. Our long-term plan for the code formatter is to make it extensible to support any syntax and custom forms. SNOWWHITE is thus a natural choice as a foundational printer, due to its expressiveness.

The code formatter currently supports only S-expression formatting. However, the task is already challenging. While the S-expression syntax may look simple and uniform, Racket users employ a variety of styles for different forms to make them look distinctive in order to improve readability. Each function application, for example, has three possible styles (while most languages have two function application styles). The search space of the code formatter is thus quite large.

The benchmarks (Table 3) consist of files of different sizes from the Racket language codebase<sup>8</sup>. class-internal and xform are the two largest files. We use the code formatter to format these files with the page width limit of 80. We run the code formatter twice, once with  $W_{\mathcal{F}} = 100$  and once with  $W_{\mathcal{F}} = 1000$ .

<sup>8</sup><https://github.com/racket/racket/tree/master/racket/collects> at commit 4f1a2bd4

### 7.3 Results

*Performance.* The benchmarking results in Table 2 and Table 3 show that overall, SNOWWHITE is sufficiently fast in practice. While not the fastest, it can process large, practical workloads class-internal and xform under a second. Furthermore, it provides a performance guarantee even on tricky inputs. The same is not true for other printers. The Flatten benchmarks work very poorly for Wadler’s printer, and the FillSep benchmarks work very poorly for Bernardy’s printer. Interestingly, Bernardy’s naïve printer is faster than its practical variant, even though the latter is more optimized; this is due to the extra features that the practical printer needs to support. SNOWWHITE, by contrast, is set to support these features from the start.

We note two interesting observations on SNOWWHITE. First, it performs poorly on SExpFull relative to other printers. This is due to the memory pressure from memoization. Better engineering effort may be able to alleviate this issue. Second, although the time complexity of  $\Pi_e$  is  $O(nW_{\mathcal{F}}^4)$ , this worst case behavior happens only if Pareto frontiers are always full. In practice, this is not the case<sup>9</sup>, as evidenced by the fact that increasing  $W_{\mathcal{F}}$  tenfold does not multiply the running time by  $10^4$ . On the contrary, increasing  $W_{\mathcal{F}}$  does not affect the running time at all on most benchmarks.

*Optimality.* We find that SNOWWHITE is the prettiest compared to others, offering high quality output when we use the cost factory described in Section 6. Table 2 shows (via line count) that the output layouts in many benchmarks agree in all printers. The exceptions are RandFit, RandOver, and JSONW benchmarks. Upon manual inspection, we find that the layouts produced by SNOWWHITE are better. JSONW and RandOver are cases where there is an unavoidable overflow, causing Bernardy’s printer to overflow more than necessary. Figure 16 (in Appendix A) shows a concrete example of this problem. RandFit and RandOver are cases where the greedy minimization and the `align` construct in Leijen’s printer interact poorly, as discussed and illustrated in Bernardy [2017c].

It should also be noted that neither Leijen’s nor Bernardy’s printers support custom optimality objectives, as their optimality objectives are integral to their algorithms. SNOWWHITE, in contrast, allows users to customize optimality objective via the cost factory.

Lastly, we evaluate the effectiveness of the tainting system. For almost every benchmark that gets a tainted layout ( $\mathbf{X}$ ) with the default  $W_{\mathcal{F}}$ , we find that using  $W_{\mathcal{F}} = 1000$  in an attempt to avoid taintedness<sup>10</sup> yields the same result, confirming the optimality of the output layout. The only exception is the class-internal benchmark in Table 3, for which the output layouts are different in one line and otherwise identical, because the greedy heuristic in the taint operation prunes the optimal choice away. This demonstrates that despite being tainted, and thus no longer guaranteed to be optimal, the output layout is still reasonable (at least with respect to the cost factory that we employ and the heuristic to avoid bias described in Appendix C).

## 8 CONCLUSION

We have described  $\Pi_e$ , an expressive printer that supports a variety of optimality objectives and is practically efficient. We developed a framework for reasoning about the expressiveness of PPLs, and we used this framework to guide the design of the PPL that  $\Pi_e$  targets. By surveying existing pretty printers, we have shown that  $\Pi_e$  is well-placed in the design space of printers.  $\Pi_e$  is proven correct in the Lean theorem prover and implemented as a practical printer SNOWWHITE, which powers a real-world code formatter for the Racket programming language. Our results show that  $\Pi_e$  is both pretty and fast.

<sup>9</sup>This observation also applies to Bernardy’s printers, which are also based on Pareto frontiers.

<sup>10</sup>Therefore, the Concat benchmarks do not count, since they are still tainted afterwards. The benchmarks are not interesting anyway, since there is no choice in the documents, so the output layouts are always optimal.

## REFERENCES

- Pablo R Azero Alcocer and S Doaitse Swierstra. 1998. Optimal pretty-printing combinators. <https://web.archive.org/web/20040911044443/http://www.cs.uu.nl/groups/ST/Software/PP/pabloicfp.ps>.
- Jean-Philippe Bernardy. 2017a. Disjunctionless. <https://github.com/jyp/prettiest/pull/10>.
- Jean-Philippe Bernardy. 2017b. prettiest. <https://github.com/jyp/prettiest/blob/5e7a12cf37bb01467485bbe1e9d8f272fa4f8cd5/Text/PrettyPrint/Compact/Core.hs>.
- Jean-Philippe Bernardy. 2017c. A Pretty but Not Greedy Printer (Functional Pearl). *Proc. ACM Program. Lang.* 1, ICFP, Article 6 (Aug. 2017), 21 pages. <https://doi.org/10.1145/3110250>
- Olaf Chitil. 2005. Pretty Printing with Lazy Dequeues. *ACM Trans. Program. Lang. Syst.* 27, 1 (jan 2005), 163–184. <https://doi.org/10.1145/1053468.1053473>
- Merijn De Jonge. 2002. Pretty-printing for software reengineering. In *International Conference on Software Maintenance, 2002. Proceedings.* IEEE, 550–559.
- ESLint. 2014. Change no-comma-dangle to comma-dangle. <https://github.com/eslint/eslint/issues/1350>.
- Matthias Felleisen. 1991. On the expressive power of programming languages. *Science of Computer Programming* 17, 1 (1991), 35–75. [https://doi.org/10.1016/0167-6423\(91\)90036-W](https://doi.org/10.1016/0167-6423(91)90036-W)
- Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2018. A Programmable Programming Language. *Commun. ACM* 61, 3 (March 2018), 62–71. <https://doi.org/10.1145/3127323>
- John Hughes. 1995. The design of a pretty-printing library. In *Advanced Functional Programming*, Johan Jeuring and Erik Meijer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 53–96.
- Oleg Kiselyov, Simon Peyton-Jones, and Amr Sabry. 2012. Lazy v. Yield: Incremental, Linear Pretty-Printing. In *Programming Languages and Systems*, Ranjit Jhala and Atsushi Igarashi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 190–206.
- Daan Leijen. 2000. wl-pprint: The Wadler/Leijen Pretty Printer. <https://hackage.haskell.org/package/wl-pprint>.
- Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28*, André Platzer and Geoff Sutcliffe (Eds.). Springer International Publishing, Cham, 625–635.
- Dereck C. Oppen. 1980. Prettyprinting. *ACM Trans. Program. Lang. Syst.* 2, 4 (Oct. 1980), 465–483. <https://doi.org/10.1145/357114.357115>
- Susan Owicki and David Gries. 1976. Verifying Properties of Parallel Programs: An Axiomatic Approach. *Commun. ACM* 19, 5 (May 1976), 279–285. <https://doi.org/10.1145/360051.360224>
- Simon Peyton-Jones. 1997. A pretty printer library in Haskell. <https://web.archive.org/web/20080221052958/http://research.microsoft.com/Users/simonpj/downloads/pretty-printer/pretty.html>. The identified mistakes are noted at <https://github.com/haskell/pretty/blob/50b70d1be6e17a644dc3b5c80592cf7c5b339fd9/Text/PrettyPrint/HughesPJ.hs>.
- Anton Podkopaev and Dmitri Boulytchev. 2015. Polynomial-Time Optimal Pretty-Printing Combinators with Choice. In *Perspectives of System Informatics*, Andrei Voronkov and Irina Virbitskaite (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 257–265.
- Prettier. 2016. Technical Details. <https://prettier.io/docs/en/technical-details.html>.
- S Doaitse Swierstra, Pablo R Azero Alcocer, and Joao Saraiva. 1999. Designing and Implementing Combinator Languages. In *Third Summer School on Advanced Functional Programming, volume 1608 of LNCS*. Springer-Verlag, 150–206.
- The Python Language Reference. 2010. Lexical analysis. [https://docs.python.org/2.7/reference/lexical\\_analysis.html](https://docs.python.org/2.7/reference/lexical_analysis.html).
- Emina Torlak and Rastislav Bodik. 2014. A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Edinburgh, United Kingdom, 530–541. <https://doi.org/10.1145/2666356.2594340>
- Philip Wadler. 2003. A prettier printer. *The Fun of Programming, Cornerstones of Computing* (2003), 223–243.
- Phillip Yelland. 2015. rfmt: A code formatter for R. <https://github.com/google/rfmt>.
- Phillip Yelland. 2016. A New Approach to Optimal Code Formatting. Technical note for open source project rfmt; <https://github.com/google/rfmt>.