CONSTRUCTOR UNIVERSITY BREMEN

Bachelor of Science
Computer Science

Ivan Klimov

Mechanized verification of pretty-printing library implemented in C

Bachelor's Thesis

Scientific supervisor:
Prof. Anton Podkopaev

Reviewer:

Bremen
2023

Contents

Introduction

Pretty-printing is a technique used to format code or data in a human-readable way. It involves adding indentation, line breaks, and other formatting elements to make code or data more visually organized and easier to understand. Particulary, we will be interested in pretty-printers, which transform code from some abstract representation of a program(for example, AST) into comprehensible text that satisfies certain properties.

When code is formatted in a way that is easy to read, developers can quickly identify errors or inconsistencies, making debugging and troubleshooting faster and more efficient. Additionally, well-formatted code is easier to modify, update, and maintain over time, as changes can be made with confidence that they will not inadvertently affect other parts of the codebase.

Since pretty-printing is an essential tool for any developer looking to create code, many algorithms, using different techniques and different asymptotics to achieve the desired result, have emerged [5, 1, 3, 7]. Particulary in the area of functional programming, algorithms based on pretty-printing combinators have appeared, allowing very natural way of working with text representation of AST. Wadler [8] and Hughes' [3] libraries stand out among them, as they are included in the standard libraries of Ocaml and Haskell. However, these algorithms are not powerful enough in terms of expressive power, and are less efficient.

A more flexible approach is pretty-printer combinators with choice, providing more freedom when working with combinators and thus allowing you to find the optimal solution. Pretty-printer combinators with choice (we will be calling them printers in further reading) were originally presented in the works of Pablo Azero and Doaitse Swierstra [1, 7], however, their proposed algorithm has exponential complexity in the worst case. Significant impovements have been made in libraries of Anton Podkopaev and Dmitri Boulytchev [6] and Jean-Philippe Bernardy [2], algorithms in which are polynomial.

Bugs in algorithm implementations can lead to many problems, such as loss of information during formatting, mismatch between the resulting text and the given parameters, etc. Therefore, it is important to verify the code that is used to format data. Among recent results: a Vladimir Korolikhins' verified library of printers [4] on Coq, an interactive proof assistant, providing a formal language to write mathematical definitions, executable algorithms and theorems together with an environment for semi-interactive development of machine-checked proofs. The library includes different functional pretty-printing algorithms, starting from the simple but less efficient algorithm by Pablo Azero and ending with more advanced polynomial algorithms by Anton Podkopaev and Dmitri Boulytchev [6] and Jean-Philippe Bernardy [2].

However, all verified versions of pretty-printing libraries are functional. Implementing a verified pretty-printer library in C and further expanding it with a polynomial algorithm will allow us to achieve significant performance improvements. This will be the main goal of our work.

1 Goals and objectives

The goal of this work is to provide a verified implementation of pretty-printer combinators library in C. The algorithm that we are going to implement and verify is described in article of Swierstra et al. [7]. It has exponential complexity in the worst case, nevertheless, it was chosen because it supports the extension of functionality, as the algorithm is based on the same approach (pretty-printer combinators with choice) as more advanced polynomial algorithms [6, 2]. Thus, the main objectives of this work are:

1. Implement the pretty-printer library [7] in C.
2. Prove that the C implementation of the core operations on formats satisfy its functional specification [4].
3. Extend the specification to sets of formats implemented as linked lists.

2 Related work

2.1 Pretty-printing library

This sections aims to give detailed explanation of pretty-printing library [7] and all the approaches that are used in it, including detailed description of pretty-printer combinators.

2.1.1 Brief overview

The algorithm takes as input a representation of the text in the form of an AST, and for each node, it constructs a list of possible formatted text pieces. Each formatted text piece is stored in a structure called *format*. In addition to the text itself, the format contains some properties of the formatted data for ease of further manipulation with the structure. The combination of formats is done using *pretty-printing combinators*, which are essentially functions that take two formats as arguments and return a new format that represents formatted concatenation of the two original text pieces. The algorithm uses printer combinators to iterate through all possible formatting options for the data. In other words, the node of the AST is associated with not a single format, but a list of formats that are all possible ways to format the data corresponding to that node. In each of the AST leaves, there is list of a single element (the format, which is essentially a string), and during recursive runs, all possible formats of the left and right children are iterated through, and all possible combinations of combinators are applied to them. At the end of the algorithm, we choose the most suitable format for us from the root node.

2.1.2 Description of the `format` structure

The atomic structure when working with printer combinators is the `format` structure. This structure represents pre-formatted text that includes parameters of its height and width, as well as `to_text` function, which allows obtaining its text representation. The `to_text` function takes two arguments: `shift`, which specifies the desired shift of the text represen-

tation, and `line`, which will be concatenated with our text representation at the end.

```
Record t : Type := T {
  height           : nat;
  first_line_width : nat;
  middle_width     : nat;
  last_line_width  : nat;
  to_text          : nat -> string -> string
}.
```

Figure 1: `Format` definition in the reference library [4]

Let's give an example of a specific format and the information it contains. Let's say we want to format the following piece of `C++` code:

```
int s;
if(condition1) { s = 1; } else if(condition2) { s = 2; }
```

Figure 2: Sample of `C++` code

This code can be formatted in many different ways. Let's consider the format `G` and say that it contains the following representation of our code:

```
int s;
if(condition1) {
  s = 1;
} else if(condition2) {
  s = 2;
}
```

Figure 3: Formatted `C++` code

Then our format will have the following values for its fields:

```
G =
{|
  height := 6;
  first_line_width := 6;
  middle_width := 23;
  last_line_width := 1;
  to_text := ...
|}
```

Figure 4: Format `G`

The implementation of the `to_text` function in this case is too complicated to include it in the text of this thesis. However, the example of using this function should give readers an intuition about how it works:

```
// G.to_text 0 '' else {\n  assert(false);\n}'' ==>
int s;
if(condition1) {
  s = 1;
} else if(condition2) {
  s = 2;
} else {
  assert(false);
}
```

Figure 5: Example of using the `to_text` function

2.1.3 Description of pretty-printer combinators

Pretty-printer combinator is a function that take a format as input and return a new format that contains data from the original formats and, obviously, has a strictly larger size than each of them. There are three different combinators, let's take a closer look at each of them:

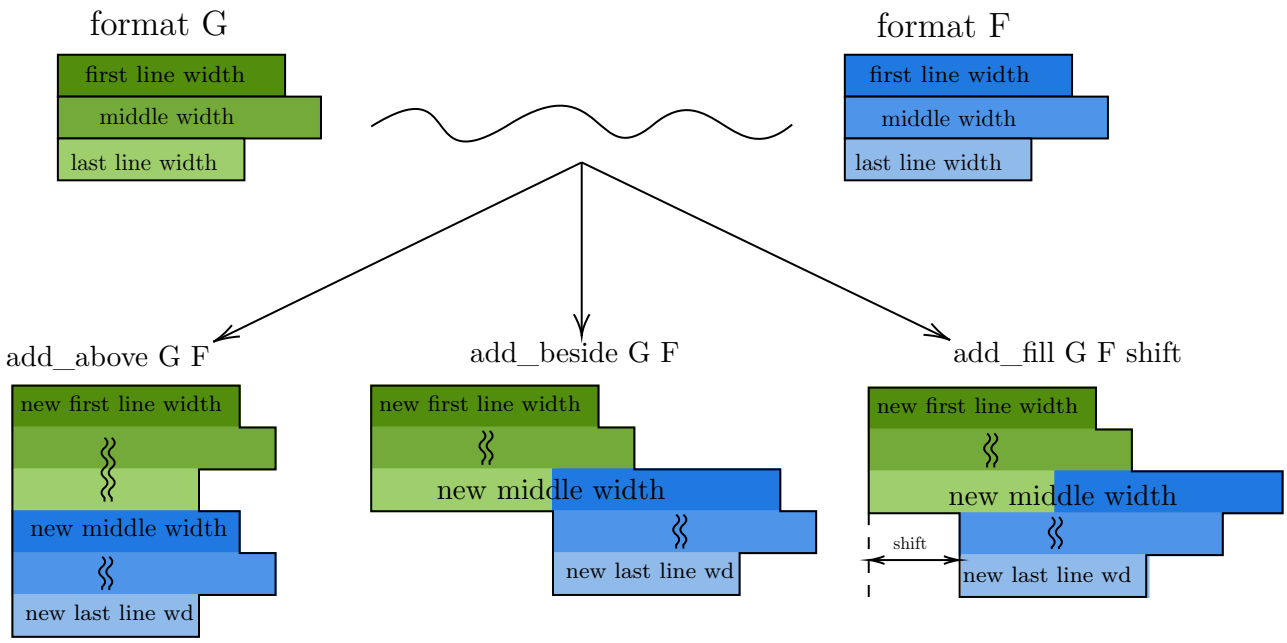

Figure 6: Pretty-printer combinators

In the picture, you can see various examples of applying combinators to formats and the corresponding characteristics of the resulting format (the height, width of the first line, width of the last line, and width of the data in the middle). In general, the fill combinator is not present in article [7], however, it was decided to include it in this work because it was introduced in article [6] and will be needed in case of expanding the library with a polynomial algorithm.

2.1.4 Detailed description of the algorithm

Previously we gave a brief description of the algorithm used, now let's describe each step of the algorithm in more detail.

The algorithm is given a recursive structure called `Doc`, which is a tree, with each node containing one of several combinators.

```
Inductive Doc : Type :=
| Text (s: string)
| Indent (t: nat) (d: Doc)
| Beside (d: Doc) (d: Doc)
| Above (d: Doc) (d: Doc)
| Choice (d: Doc) (d: Doc)
| Fill (d: Doc) (d: Doc) (s: nat).
```

Figure 7: `Doc` definition in the reference library [4]

Our algorithm will recursively descend through this structure, and depending on the type of node, it will process the data received from the child nodes in different ways:

- `Text` — a leaf node of the tree that contains a single format. Text takes the string from which the format that will represent the data in the leaf is constructed. The string is simply broken down into different lines, from which format characteristics such as height and width are calculated. As we mentioned earlier in the brief overview, each node of the tree corresponds to a list of possible formats. Therefore, the Text node will correspond to a list containing a single element — the resulting format.

- `Indent` — a node that takes the list of formats obtained from the child node and shifts all the formats to the right by `t`. In other words, at the beginning of each line, it adds `t` spaces and increases all parameters related to width by `t`. This is the only type of node that does not change the size of the list, but simply modifies the existing formats slightly.

- `Choice` — a node that gives us flexibility in using combinators. It does not modify the formats obtained from the child nodes, but takes two lists, concatenates them, and passes them up. Thanks to the `Choice` node, we can choose for ourselves where and which types of combinators to iterate over, instead of trying to use formats that are unsuitable for us by default.

- `Beside, Above, Fill` — nodes that take two lists of formats received from the child nodes and apply the corresponding combinator to them pairwise. The detailed operation of each combinator is described in details in section 2.1.2.

The algorithm recursively descends through the tree, applying corresponding operations for each node. As a result, we obtain a list of suitable formats that are located in the root node. We remove from this list all formats that do not fit the constraints of width or height and take any of the remaining formats. This format will be the final formatted version of our initial text.

2.2 Mechanized verification

3 Implementation in C

4 Correctness of printer combinators

5 Correctness of the algorithm

Conclusion and future work

References

[1] Pablo Azero and Doaitse Swierstra. "Optimal Pretty-Printing Combinators". In: (1998). URL: http://www.cs.ruu.nl/groups/ST/Software/PP.

[2] Jean-Philippe Bernardy. "A pretty but not greedy printer (functional pearl)". In: *Proceedings of the ACM on Programming Languages* 1 (2017), pp. 1–21. DOI: https://doi.org/10.1145/3110250.

[3] John Hughes. "The design of a pretty-printing library". In: *International School on Advanced Functional Programming* 925 (1995).

[4] Vladimir Korolihin. *Proof of correctness of the functional implementation of the pretty-printer combinator libraries with choice.* 2020. URL: https://se.math.spbu.ru/thesis/texts/Korolihin_Vladimir_Igorevich_Bachelor_Report_2020_text.pdf.

[5] Derek C. Oppen. "Pretty-printing". In: *ACM Transact. Program. Lang. Syst.* 2 (1980), pp. 465–483.

[6] Anton Podkopaev and Dmitri Boulytchev. "Polynomial-Time Optimal Pretty-Printing Combinators with Choice". In: *International Andrei Ershov Memorial Conference on Perspectives of System Informatics* (2014), pp. 257–265. DOI: http://dx.doi.org/10.1007/978-3-662-46823-4_21.

[7] Pablo R Azero Alcocer S Doaitse Swierstra and Joao Saraiva. "Designing and implementing combinator languages". In: *International School on Advanced Functional Programming* (1998), pp. 150–206.

[8] Philip Wadler. "A Prettier Printer". In: *The Fun of Programming, Cornerstones of Computing* (2004), pp. 223–243.