

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

<b>Introduction</b>	<b>3</b>
<b>1. Goals and objectives</b>	<b>5</b>
<b>2. Related work</b>	<b>6</b>
2.1. Pretty-printing library . . . . .	6
2.1.1. Description of the <code>format</code> structure . . . . .	6
2.1.2. Description of pretty-printer combinators . . . . .	9
2.1.3. Detailed description of the algorithm . . . . .	10
2.2. Mechanized verification . . . . .	12
2.2.1. Separation Logic . . . . .	12
2.2.2. Verified Software Toolchain(VST) . . . . .	13
<b>3. Implementation in C</b>	<b>14</b>
3.1. <code>Format</code> representation . . . . .	14
3.2. Standard library . . . . .	16
3.3. Combinators representation . . . . .	19
<b>4. Correctness of printer combinators</b>	<b>25</b>
<b>5. Correctness of the algorithm</b>	<b>26</b>
<b>Conclusion and future work</b>	<b>27</b>
<b>References</b>	<b>28</b>

# 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. Particularly, 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 accidentally affect other parts of the codebase.

Since pretty-printing became 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 [6, 1, 4, 9, 7, 2]. Particularly in the area of functional programming, algorithms based on pretty-printing combinators have appeared, allowing a very natural way of working with text representation of AST. Wadler [10] and Hughes' [4] 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 to put more constraints on the solutions obtained as a result of the algorithm. 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, 9], however, their proposed algorithm has exponential complexity in the worst case. Significant improvements have been made in libraries of Anton Podkopaev and Dmitri Boulytchev [7] 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 [5] 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 [7] and Jean-Philippe Bernardy [2].

However, all verified versions of pretty-printing libraries are functional. Implementing a verified pretty-printer library in C will allow us to optimize the written code more efficiently, and in the future expand the library with a polynomial algorithm, which will result in hopefully more performant implementation than its existing functional counterparts. 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. [9]. 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 [7, 2].

After implementing the library in C, we will need to verify it. To do this, we will prove the correspondence between our library and Korolikhin’s verified library [5] on Coq (which was chosen due to the absence of analogs, as well as the presence in this library of the algorithm we are going to implement). We will divide this part into two subtasks.

Firstly, we will need to verify the basic structures for working with printer combinators, as well as the printer combinators themselves. This will allow us to already at that point be able to extend the library with any algorithm based on pretty-printer combinators.

Secondly, we will need to verify the algorithm itself, in which at that point the printer combinators will already be verified and it will only remain to prove statements about the highest-level functions that manipulate printer combinators. Thus, the main objectives of this work are:

1. Implement the pretty-printer library [9] in C.
2. Prove that the C implementation of the core operations on formats and combinators satisfy its functional specification [5].
3. Extend the proof to verify the correctness of the chosen algorithm [9].

## 2 Related work

### 2.1 Pretty-printing library

This sections aims to give detailed explanation of pretty-printing library, described in article by Swierstra et al. [9] and all the approaches that are used in it, including detailed description of pretty-printer combinators.

#### 2.1.1 Description of the format structure

The atomic structure when working with printer combinators is the `format` structure.

```
Record format : Type := Format {  
  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 [5]

This structure represents pre-formatted text that includes the following parameters:

1. **Height** of the format. This parameter is used to determine the number of lines in the formatted text.
2. **First line width** of the format. This parameter is used to determine the width of the first line of the formatted text.
3. **Middle line width** of the format. This parameter is used to determine the maximum width of all the lines between first and last lines. If there's only two lines in the formatted text, then this parameter is equal to the first line width. If there's only one line in the formatted text, then this parameter is equal to the width of this line.

4. **Last line width** of the format. This parameter is used to determine the width of the last line of the formatted text.
5. **to\_text** function allows obtaining text representation of the format. The **to\_text** function takes two arguments: **shift**, which specifies the desired shift of the text representation, and **line**, which will be concatenated with our text representation at the end. In particular, by taking a shift equal to zero and an empty string as a line, we obtain an exact textual representation of the formatted data.

As an illustration, we'll give an example of a specific format and the information it contains. Suppose 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:

```
1 int s;  
2 if(condition1) {  
3     s = 1;  
4 } else if(condition2) {  
5     s = 2;  
6 }
```

Figure 3: Formatted C++ code

We will go through all the format's fields and calculate values for them:

1. **G.height**. As shown in the Figure 3, the amount of lines in the formatted text is equal to 6. Therefore, the height of our format is equal to 6.

2. `G.first_line_width`. As shown in the Figure 3, the first line of the text is `int s;`. It's width is equal to 6, so the first line width of our format is also equal to 6.
3. `G.middle_width`. As shown in the Figure 3, there are 4 lines that are between the first one and the last one:

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

The third one obviously has the maximum width among them, and it's width equal to 23. So the middle line width of our format is also equal to 23.

4. `G.last_line_width`. As shown in the Figure 3, the last line of the text is `}`. It's width is equal to 1, so the last line width of our format is also equal to 1.
5. `G.to_text`. As we discussed earlier, the `to_text` function with default arguments returns the formatted text, shown in Figure 3. Changing the first parameter affects the indentation of the entire returned text, and changing the second parameter allows modifying the entire text by adding an arbitrary string to the end of it. Later, we will show that these parameters are not necessary for a work with formats, nevertheless, it is important to understand their semantics as they are used in the reference library.

<pre>int s; if(condition1) {     s = 1; } else if(condition2) {     s = 2; }</pre>	<pre>int s; if(condition1) {     s = 1; } else if(condition2) {     s = 2; } else {</pre>
--	---

Figure 4: `G.to_text(0, EmptyString)` and `G.to_text(5, " else {")`



Internal representation of our concrete format is shown in Figure 5. We didn't include the specific implementation of `to_text` function as it can be implemented in many different ways and it's more important to understand its semantics.

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

Figure 5: Format G

### 2.1.2 Description of pretty-printer combinators

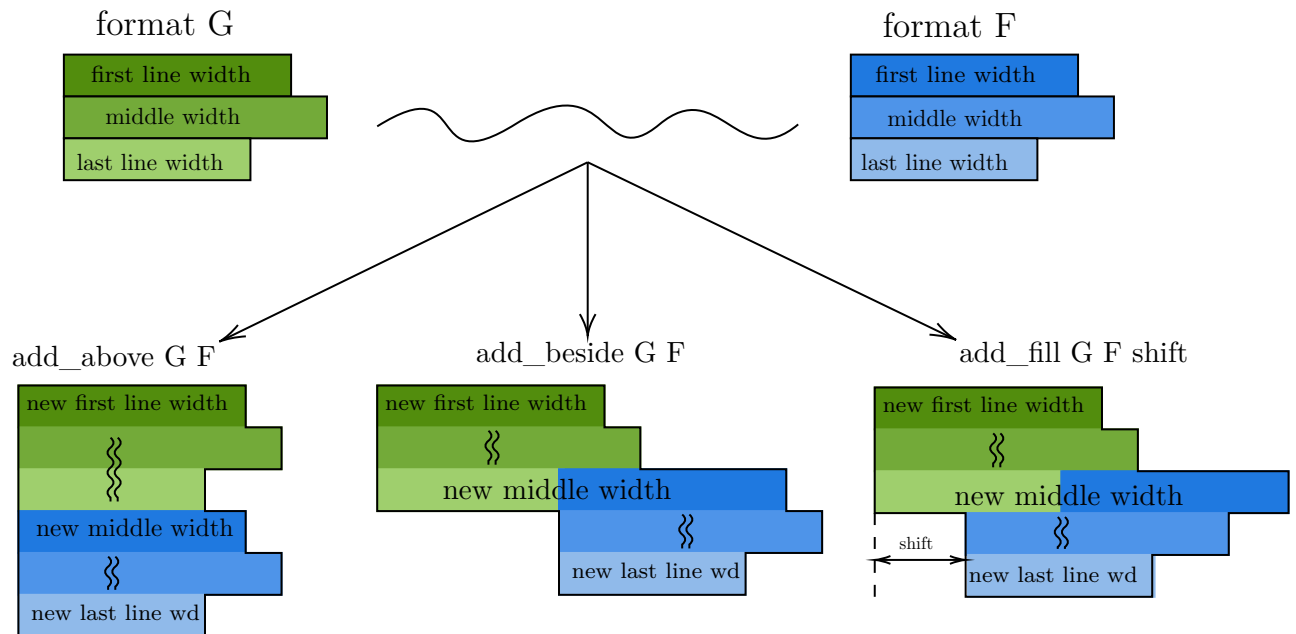


Figure 6: Pretty-printer combinators

Pretty-printer combinator is a function that takes a format as input and returns 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: `add_above`, `add_beside` and `add_fill`.

In Figure 6, you can see 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 [9], however, it was decided to include it in this work because it was introduced in article [7] and will be needed in case of expanding the library with a polynomial algorithm.

### 2.1.3 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 actions.

```
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 [5]

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  $\tau$ . In other words, at the beginning of each line, it adds  $\tau$  spaces and increases all parameters related to width by  $\tau$ . 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 2.1.1.

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

This section is dedicated to tools for automated verification of programs in C.

### 2.2.1 Separation Logic

The standard tool for proving partial correctness of programs is Hoare Logic [3]. It is a formal system that involves a collection of logical rules for meticulous reasoning about computer programs. The key component of this system is famous *Hoare triple*, which consist of two assertions(precondition and postcondition) and a command.

$$\begin{array}{ccccc} \{x = 239\} & y := x + 1 & \{y := 240\} \\ \text{precondition} & \text{command} & \text{postcondition} \end{array}$$

Basically, the Hoare triple is statement, which denotes that when the precondition is met, executing the command establishes the postcondition. To construct new triples and assertions, Hoare Logic introduces *inference rules*, which take the form as shown in Figure 8.

$$\frac{P_1, P_2, \dots, P_n}{Q}$$

Figure 8: Hoare rule notation

The inference rules assert the following: if all the *premises*  $P_1, P_2, \dots, P_n$  are satisfied, then the *conclusion*  $Q$  is also satisfied. In addition to the ability to create custom rules (provided that their correctness is proved), Hoare Logic includes a set of standard rules — *axioms*.

$$\frac{}{\{P\} \text{ skip } \{P\}} \qquad \frac{\{P\}S\{Q\} \quad , \quad \{Q\}T\{R\}}{\{P\}S;T\{R\}}$$

Figure 9: Hoare axiom rules examples

However, despite the fact that Hoare Logic is a highly potent tool for working with imperative programming languages, it has a number of limitations. The biggest one is the lack of support for pointers, which is especially relevant for reasoning about imperative programs. To address this problem, Hoare Logic was extended to Separation Logic by John C. Reynolds and Peter O’Hearn in 2002 [8]. It allows for working with statements of the form  $s, h \models P$ , where  $s$  is a store,  $h$  is a heap and  $P$  is an assertion over the given store and heap. Separation Logic also introduces several new constants and operators that allow for constructing statements related to memory. We will consider those that are relevant for our work:

- Constant **emp** — constant, which asserts that heap is empty. For instance  $s, h \models \mathbf{emp}$  when  $h$  is undefined for all addresses.
- Operator  $e \mapsto e'$  denotes that heap is defined at address  $e$  and contains value  $e'$ .
- Operator  $s, h \models P * Q$  denotes that there exist  $h_1, h_2$  such and  $h = h_1 \sqcup h_2$  and  $s, h_1 \models P$  and  $s, h_2 \models Q$ .

### 2.2.2 Verified Software Toolchain(VST)

## 3 Implementation in C

In this section, we will provide a brief overview of the implementation of our C library, examine examples of the most important functions, and also discuss the solutions we have adopted due to the use of an imperative programming language (as well as to simplify verification).

### 3.1 Format representation

As shown in the Figure 1, the `Format` structure consists of 5 fields. `height`, `first_line_width`, `middle_width`, `last_line_width` are natural numbers and thus can be represented as `unsigned int` (which is done in our implementation). However, the `to_text` field is a function and in general can't be conveniently represented in C, so we had to turn to the semantics of this function.

We noticed that the `shift` and `line` parameters in the original functional library exist solely for the sake of code simplicity and are not an essential part of the algorithm. It is easy to see that given a format `G`, a natural number `shift`, and a string `line`, we can recover `G.to_text(shift, line)` from just `G.to_text(0, EmptyString)`. Therefore, we made the decision to store a linked list of strings instead of the `to_text` function.

<pre>struct format {     unsigned int height;     unsigned int first_line_width;     unsigned int middle_width;     unsigned int last_line_width;     list *to_text; } typedef format;</pre>	<pre>struct list {     size_t shift;     char *line;     struct list *tail; } typedef list;</pre>
--	---

Figure 10: Format in C

Additionally, the string indentation and concatenation functions were made independent of the `format` structure. Implementations of `format` structure and linked list are shown in Figure 10.

In our implementation, each combinator returns a completely new format that is not a reference to any other format, as each format can be reused many times during the algorithm. And since sometimes the result of a combinator's work is equal to one of the formats passed to it, it is necessary to have function, that creates a copy of a specific format. Thus, the main functions for manipulating formats, in addition to printer-combinator functions, are `format_copy` and `list_copy`, which turned out to be one of the most difficult function to verify in the first part of the work.

```
list* list_copy(list *l) {
    if (l == NULL) return NULL;
    list *new = malloc(sizeof(list));
    if(!new) exit(1);

    list *cur = new;
    list *l_cur = l;
    while(true) {
        cur->shift = l_cur->shift;
        cur->line =
            malloc(strlen(l_cur->line) + 1);
        if(!cur->line) exit(1);
        strcpy(cur->line, l_cur->line);
        cur->tail = NULL;

        if (l_cur->tail == NULL) {
            cur = NULL;
            break;
        }
        cur->tail =
            malloc(sizeof(list));
        if(!cur->tail) exit(1);
        cur = cur->tail;
        l_cur = l_cur->tail;
    }
    return new;
}
```

```
format* format_copy(format *G) {
    format* result =
        malloc(sizeof(format));
    if(!result) exit(1);
    result->height = G->height;
    result->first_line_width =
        G->first_line_width;
    result->middle_width =
        G->middle_width;
    result->last_line_width =
        G->last_line_width;
    result->to_text =
        list_copy(G->to_text);
    return result;
}
```

Figure 11: `list_copy` and `format_copy` functions

Our implementations of `list_copy` and `format_copy` function are shown in Figure 11. Note that the importance of implementing and verifying these functions will be particularly noticeable in the future sections when discussing the internal structure of the functions that implement printer combinators.

## 3.2 Standard library

Since our library is heavily focused on working with text, basic functions from the standard library, in particular from `string.h`, are required for manipulating our formats. However, in VST-Floyd, only the functions `malloc`, `free`, and `exit` were verified. Therefore, all the functions we wanted had to be implemented by ourselves. In total, we used 3 functions from standard library, let's take a closer look at each one of them:

1. `strlen` — function that returns the length of the string. The implementation of this function is shown in Figure 12.

```
size_t strlen(const char *str) {  
    size_t i;  
    for (i=0; ; i++)  
        if (str[i]==0)  
            return i;  
}
```

Figure 12: `strlen` implementation

The function takes a pointer to the string and iterates over it until it finds a null character. Then it returns the number of characters that have been iterated over. In our library it mostly used to calculate number of bits to then put it to the `malloc` function.

2. `strcpy` — function that copies the string pointed to by `src` to the buffer pointed to by `dest`. The implementation of this function is shown in Figure 13.



```

char *strcpy(char *dest, const char *src) {
    size_t i;
    for(i = 0;; i++){
        char d = src[i];
        dest[i] = d;
        if(d == 0) return dest;
    }
}

```

Figure 13: `strcpy` implementation

The function takes two pointers to the strings and iterates over the first one until it finds a null character. Then it copies the characters from the first string to the second one. In our library it is used only once — in the function that copies the `format` structure.

3. `strcat` — function that concatenates the string pointed to by `src` to the end of the string pointed to by `dest`. The implementation of this function is shown in Figure 14.

```

char *strcat(char *dest, const char *src) {
    size_t i,j;
    for(i = 0;;i++){
        char d = dest[i];
        if(d == 0) break;
    }
    for(j = 0;;j++){
        char d = src[j];
        dest[i + j] = d;
        if(d == 0) return dest;
    }
}

```

Figure 14: `strcat` implementation

The function takes two pointers to the strings and iterates over the first one until it finds a null character. Then it iterates over the second string and copies the characters to the first one. In our library

it is used in `add_beside` and `add_fill` combinators, as they take two formats as argumants and concatenate the last string of the first format and the first string of the second format.

Another function that is not included in the standard C library, but which we deemed appropriate to add to this section, is the `max` function.

```
unsigned int max(unsigned int a, unsigned int b) {  
    if (a <= b)  
        return b;  
    return a;  
}
```

Figure 15: `max` implementation

The `max` function is used 17 times in our library and is used to calculate new dimensions in all combinators, so its implementation and verification were necessary for the readability of our library code.

### 3.3 Combinators representation

Let's examine in detail the internal structure of printer combinators in our reference library.

```
Definition add_beside (G:t) (G':t):t :=
match G.(height), G'.(height) with
| 0, _ => G'
| _, 0 => G
| _, _ =>
  let middle_width_new :=
    match G.(height), G'.(height) with
    | 1,(1|2) => G.(first_line_width) + G'.(first_line_width)
    | _,1     => G.(middle_width)
    | 1,_     => G.(last_line_width) + G'.(middle_width)
    | 2,_     => max (G.(last_line_width) + G'.(first_line_width))
                     (G.(last_line_width) + G'.(middle_width))
    | _,_     => max G.(middle_width)
                     (max (G.(last_line_width) + G'.(first_line_width))
                        (G.(last_line_width) + G'.(middle_width)))
  end
in
  let first_line_width_new :=
    if (G.(height) == 1) then
      G.(first_line_width) + G'.(first_line_width)
    else
      G.(first_line_width)
  in
    T
    (G.(height) + G'.(height) - 1)
    first_line_width_new
    middle_width_new
    (G.(last_line_width) + G'.(last_line_width))
    (fun s t =>
      G.(to_text) s (G'.(to_text) (s + G.(last_line_width)) t))
end.
```

Figure 16: add\_beside combinator in functional library

As you can see in Figure 16, the add\_beside combinator (as well as any

other combinator, in general) consists of several more or less independent parts. In our C implementation of the library, we tried to move as many calculations as possible out of the main combinator function to make the code more concise and easier to verify. Let's go through each part of the combinator in more detail:

1. **Beginning of the combinator.** It's easy to see (and it's actually verified in our library), that height of a format equals to zero if and only if the format doesn't contain any text (i.e. our C list is empty). In that case, whenever one of the arguments of combinator has height equal to zero, the second argument will be result of our combinator.

```
format *add_beside(format *G, format *F) {  
    if (G->height == 0) {  
        return format_copy(F);  
    }  
    if (F->height == 0) {  
        return format_copy(G);  
    }  
  
    t* result = malloc(sizeof(format));  
    if(!result) exit(1);  
}
```

Figure 17: Beginning of `add_beside` function in C

In our library, this logic is supported by relatively simple conditions as shown in Figure 17. Moreover, because this reasoning apply to all combinators at once, this piece of code appears in all functions: `add_above`, `add_beside`, and `add_fill`. It is also worth noting that it is exactly at this point that we need `format_copy`, because without it, we would have to return a pointer to one of the arguments, which obviously does not suit us due to the multiple reuse of the same format in different combinators.

2. **Format height calculation.** As shown in Figure 16, the height of the resulting format is calculated in a pretty straightforward way

without any case distinction. In fact, a similar formula is applicable to almost all combinators (except that in `add_above` the height is equal to `G.height + G'.height`).

```
result->height = G->height + F->height - 1;
```

Figure 18: Height calculation in `add_beside` function in C

Thus, in all combinators, the height calculation always corresponds to only one line, and in the case of `add_beside`, this line is shown in Figure 18.

3. **Format first line width calculation.** Here is where the differences between the combinators begin. Obviously, in `add_above`, the first line width is always equal to the first line width of the first argument (if it is not empty). However, in `add_beside` and `add_fill`, the situation may be different if the height of the first argument is one. In this case, the first line widths of both arguments are added together.

```
unsigned int flw_add_beside(format *G, format *F) {
    if(G->height == 0)
        return F->first_line_width;
    if(F->height == 0)
        return G->first_line_width;

    unsigned int first_line_width_new;
    if (G->height == 1)
        first_line_width_new = G->first_line_width + F->first_line_width;
    else
        first_line_width_new = G->first_line_width;
    return first_line_width_new;
}
```

Figure 19: `add_beside` first line width calculation in C

Now, there is a case distinction and the logic is no longer primitive. For ease of verification we put this piece of code into a separate function, as shown in Figure 19.

4. **Middle width calculation.** Now the formulas become slightly more complicated, because the value of middle line width itself has a rather complex semantics. In the case when the format height is equal to one, middle line width is equal to the width of the only line in the format. In the case when the format height is equal to two, middle line width is equal to the first line width of this format. In all other cases, middle line width is equal to the maximum width of all the lines located between the first and last lines in the format.

```
unsigned int mdw_add_beside(format *G, format *F) {
    if(G->height == 0)
        return F->middle_width;
    if(F->height == 0)
        return G->middle_width;

    unsigned int middle_width_new;
    if (G->height == 1 && F->height == 1) {
        middle_width_new = G->first_line_width + F->first_line_width;
    } else if(G->height == 1 && F->height == 2) {
        middle_width_new = G->first_line_width + F->first_line_width;
    } else if (F->height == 1) {
        middle_width_new = G->middle_width;
    } else if (G->height == 1) {
        middle_width_new = G->last_line_width + F->middle_width;
    } else if (G->height == 2) {
        middle_width_new = max(G->last_line_width + F->first_line_width,
                               G->last_line_width + F->middle_width);
    } else {
        middle_width_new = max(G->middle_width,
                               max(G->last_line_width + F->first_line_width,
                                   G->last_line_width + F->middle_width));
    }
    return middle_width_new;
}
```

Figure 20: add\_beside middle line width calculation in C

As shown in Figure 20, the middle line width for the add\_beside

combinator is calculated in a rather cumbersome way. Due to the reasons described above, for other formats the size of the formula remains more or less the same (although its content may change).

5. **Format last line width calculation.** The situation is very similar to the calculation of height. In two out of three combinators (`add_above` and `add_beside`), the calculation is done in one line of code.

```
result->last_line_width = G->last_line_width + F->last_line_width;
```

Figure 21: `add_beside` last line width calculation in C

6. **Format `to_text` calculating.** This part is the most non-trivial of all. There are no formulas here, however, we need to build a complex structure, shown in Figure 6.

```
list *to_text_add_beside(format *G, format *F) {
    if(G->height == 0)
        return list_copy(F->to_text);
    if (F->height == 0)
        return list_copy(G->to_text);
    list *head = list_copy(G->to_text);           /* a */
    list *tail = get_list_tail(head);             /* b */
    list *copy_F = list_copy(F->to_text);         /* a */
    tail->line =
        line_concats(tail->line, copy_F->shift, copy_F->line); /* c */
    shift_list(copy_F->tail, G->last_line_width); /* d */
    tail->tail = copy_F->tail;                     /* e */
    free(copy_F);                                 /* f */
    return head;
}
```

Figure 22: `add_beside to_text` calculation in C

Let's break down the steps of the algorithm that might work for us:

- (a) Copy formats G and F.

- (b) Find the tail of format G. *We explicitly write this as a separate step, as it is a non-trivial task in a singly linked list.*
- (c) Concatenate a string from the tail of format G with the first line of format F.
- (d) Shift all lines of format F, starting from the second, by `G.last_line_width`.
- (e) Connect the modified G and the modified F starting from the second line.
- (f) Free the remaining unused data.

And exactly this algorithm we implemented in our library. Its correctness was also proven, although we will talk about it in the next chapter.

In fact, after writing a function for constructing `to_text` in `add_beside`, the other combinators will not cause any particular problems. It is easy to see that by replacing `G.last_line_width` with an arbitrary shift in the algorithm, we get `add_fill`. Moreover, to construct the list for `add_above`, it is only necessary to concatenate `G.to_text` and `F.to_text`, which is done trivially, having the `get_list_tail` function.



## 4 Correctness of printer combinators

## 5 Correctness of the algorithm

# Conclusion and future work

We have provided a verified implementation of a pretty-printing library in C, described in article by Swierstra et al. [9]. Let's go through each of our initial subgoals:

1. We implemented the pretty-printer library [9] in C. In particular, all the logic for working with formats and printer combinators has been implemented. This serves as the foundation for further extension of the library with polynomial algorithms [7, 2]. Additionally, we implemented a pretty-printing algorithm, making our library a comprehensive tool for working with data that requires formatting.
2. We have proved that our C implementation of the core operations on formats and combinators satisfy its functional specification [5].
3. We have proved that our C implementation of the algorithm by Swierstra et al. [9] satisfies its functional specification [5].

Continuing our work, we plan to provide a verified version of Anton Podkopaev and Dmitri Boulytchevs' algorithm [7] based on the printer combinators we have implemented and verified. This way, we will obtain a verified polynomial algorithm that can be conveniently used as a working tool for a code pretty-printing.

The source code for the thesis is available at:

<https://github.com/klimoza/verified-kisa>

# 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] C. A. R. Hoare. “An Axiomatic Basis for Computer Programming”. In: *Communications of the ACM* 12 (1969), pp. 576–580. DOI: 10.1145/363235.363259.
- [4] John Hughes. “The design of a pretty-printing library”. In: *International School on Advanced Functional Programming* 925 (1995).
- [5] 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](https://se.math.spbu.ru/thesis/texts/Korolihin_Vladimir_Igorevich_Bachelor_Report_2020_text.pdf).
- [6] Derek C. Oppen. “Pretty-printing”. In: *ACM Transact. Program. Lang. Syst.* 2 (1980), pp. 465–483.
- [7] 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](http://dx.doi.org/10.1007/978-3-662-46823-4_21).
- [8] John C. Reynolds. “Separation Logic: A Logic for Shared Mutable Data Structures”. In: *LICS* (2002). URL: <https://www.cs.cmu.edu/~jcr/seplogic.pdf>.
- [9] 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. URL: <https://www.cs.tufts.edu/~nr/cs257/archive/doaitse-swierstra/AFP3.pdf>.

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