LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

CHAIR OF THEORETICAL COMPUTER SCIENCE AND THEOREM PROVING

Optimizing the Isabelle Pretty Print in VS Code

Lukas Ali Prokoph

Bachelor's Thesis
in Computer Science

Supervisor: Prof. Dr. Jasmin Blanchette

Advisor: Balazs Toth

Submission Date: 16. Oct. 2025

Disclaimer

I confirm that this thesis type is my own work and I have documented all sources and material used.

Munich, October 15, 2025 Author

Acknowledgments

I want to thank my advisor Balazs Toth for giving me impulses when I had troubles to start, for encouraging me to dig deeper, and for letting me use his workplace. I want as well thank the other doctoral students in the chair, who lent me their workspaces and gave me hints and inspiration from time to time. And I want to thank all the good people, that again and again helped me to get it together and bore with my failing communication.

Abstract

Interactive theorem provers like Isabelle are quite important in a world of growing data, critical software, and an always evolving ecosystem of programming languages. Currently, the only way to use the full set of Isabelle's features in an IDE, is through the well-adapted, but antiquated jEdit text editor. To increase the variety of available IDEs, VSCode was adapted to Isabelle and is currently still under development. In this work, I improved the formatting of output code, known as pretty printing, by (1) improving the measurement of the available width of VSCode's state panel component and the way of initiating it, (2) introducing a new way to use exact character widths for special symbols, and (3) improving the quality of the output code by setting an additional parameter.

Keywords: Isabelle, Interactive Theorem Prover, Pretty Printing, jEdit, VSCode, Extension

Contents

1 Introduction

Mathematical proofs are the gold standard of scientific certainty - however, longer proofs can be quite cumbersome to verify and there have been instances of mathematical proofs turning out to be faulty years and even decades later. To conquer the problem, automatic theorem solving was developed - with the first computationally solved math problem being the four color theorem in 1976 by Appel and Haken (Appel & Haken (1976)) after over a century of failed proof attempts by famous mathematicians of their times (Gonthier (2023)). In the field of theoretical computer science, automatic theorem proving has become an important tool to cope with steadily increasing complexity and size of mathematical theorems and the need for formal verification of critical software, languages, protocols, or type systems.

One of the tools is the interactive theorem prover Isabelle, which was introduced by Paulson (1986) and developed and published by *Cambridge University*. Later on, it was further developed by *Technical University of Munich* (TUM) and *Cambridge University* as well as many other contributors from all over the world ((TUM 2024)). A lot of the Integrated Development Environment, a software that helps developers to write code (IDE) integration in jEdit and VSCode was done and supervised by Makarius Wenzel.

2 Background

Isabelle/HOL is the core theorem proving environment for higher-order logic, which can also be extended to other logical formalisms. It's main proof language is Isar/ML ("Intelligible semi-automated reasoning"), which allows writing human-readable proofs. The prover kernel and logic engine of Isabelle is implemented in standard ML, while the system integration and IDE part is mainly written in Scala. The two systems are linked by PIDE ("Prover IDE"), which orchestrates the information flow between the stateless Isabelle core and the stateful Graphical User Interface. That part of software, that the user interacts with by clicking, dragging, or typing (GUI) components of the IDE (Wenzel (2018)).

The core feature in the display of the human-readable proofs is proper formatting: we need linebreaks and indentations at the right places to quickly understand the connections and levels in longer code segments. To calculate the formatting, over time different algorithms were developed. These algorithms have to adhere to three basic principles: visibility, legibility, and frugality. Visibility means that everything has to fit the window width. Legibility means that the ability for a human to grasp the structure of the shown code should be supported by the layout. And by frugality, we understand the property of the algorithm to decide the layout as fast and resource effective as possible, leading to minimal lag and a real-time user experience in the best case (Bernardy (2017)).

One of the first prettyprint algorithms that is still widely used and which is the basis for Isabelle's prettyprinting, is the algorithm Oppen suggested in (Oppen (1980)). Oppen's algorithm works line by line and is agnostic of the whole code. It works with the basic parameter margin, which is the width of the available space in characters and gets updated at every step. The basic algorithm consists of two cooperating functions: scan() and print(). scan() looks ahead and computes the sizes for the tokens string, blank, and block (with it's delimiters] and [). The size of every token is stored in the size array:

- string: length of the string

- blank: length of the next block + 1
- `[`: the length of the following block
- `]`: 0

Whenever a block closes (with "`]`"), the scan() algorithm goes back through the stack until it reaches the opening bracket "`[`", counts the length of the block on the way, and stores it in the size array. If the size of the block is bigger than the remaining margin, size is set to ∞ and an instant break decision is made. Blocks can be nested - and they usually are. (Oppen (1980))

2.1 The pretty mechanism in Isabelle

Isabelle implements the Oppen Algorithm in a more refined way by working with XML markup trees as source and output. The XML trees in Isabelle generally have an XML.Body, that consists of XML.Elems that themselves contain an XML.Body and optional markup that stores information about highlighting or references which the child XML.Body should point to. The `pretty.ML` output XML trees consist of nodes of type Str, Break, and block and have the option to break at all possible break points within a block ("consistent block"). While the `pretty.ML` returns XML trees with all possible blocks and break points, the actual dynamic adaption of these trees to the GUI happens in `pretty.scala`: the XML is first processed by the `separate` function, which parts the single subtrees and puts a separator element between them. After that the tree gets formatted, which means that the algorithm traverses the tree and decides dependent on the variables `margin`, `breakgain`, and a fallback `emergencypos`, where the code should actually break. In order to decide, it uses a character width metric class to calculate the length of strings, which has the possibility to feed in variable character widths derived from varying font characteristics and special symbols. The character width metric classes used for the VSCode output by the Isabelle/Scala backend (later just called "backend"), are either codepoint.Metric (as a fallback) or Symbol.Metric (used by the central `pretty_text_panel.scala` as a parameter for `Pretty.formatted()`). While codepoint.Metrics works with the standard symbols used in jEdit, UTF-8-Isabelle (a custom UTF-8-extension), Symbol.Metric has a far simpler approach: it multiplies the Unicode character width with the amount of the prefix "long" in the symbol's Isabelle format (e.g. while the Isabelle symbol with the American Standard Code for Information Interchange, an older and smaller encoding standard compared to UTF-8, with only a subset of characters (ASCII) representation `\<Rightarrow>` gets assigned the relative width 1, the symbol represented by `\<Longrightarrow>` is assigned the value 2). Since the standard Isabelle font is monospaced, the character width is always 1.0 Units.

2.2 jEdit vs VSCode

The standard way of using Isabelle in an IDE is via jEdit, an early IDE from around 1998 (Pestov (2024)) that is still being maintained and developed. jEdit is very tightly coupled with the PIDE backend of Isabelle: edits in jEdit are directly sent into the PIDE document model and the returning XML markup tree from the backend is rendered directly within the editor, leading to minimal delay. Since the user changes are rendered directly in jEdit and only updated by refined versions from the backend, there is always a nice version of the formatted code visible to the user, while changes load in the background asynchronously. The Isabelle backend has a lot of useful functions like syntax

highlightling and referencing, (recursive) tooltips, completion and suggestion of syntax keywords and symbols in general and session specific, and integration of asynchronous tools like automated reasoning with Sledgehammer, or counter examples with Nitpick (Wenzel (2018)). Because jEdit and Isabelle/Scala run in the same JVM, the information flow via the Isabelle/PIDE bridge is very fast. All in all, these functions are the benchmark against which the VSCode integration of Isabelle is measured up.

In VSCode, Isabelle is implemented via an extension that serves as a leightweight client and transmits data via Language Server Protocol, a standard to transport information between the front and the backend of an IDE, based on JSON-RPC (LSP) (Language Server Protocol, based on JSON-RPC) to the language server of the Isabelle/PIDE backend, which in turn communicates with the Isabelle core. The different parts are much more encapsulated in the VSCode setup, since VSCode (or rather a version of VSCodium) runs in the Electron environment, which combines Node.js with the Chromium Browser (Electron (2025)). This makes the integration of all features known from jEdit more complicated and, more importantly, slower. This is especially true for features, that are not explicitly opened in a new window or tab, but should integrate seamlessly in the workflow, like completions, suggestions, hover texts etc. One of the most basic GUI features of Isabelle is the output of code in a human-readable, formatted form and the real-time adaption of said output during changes done by the user. Since the output rendering of Isabelle/VSCode state panel and output view panel still does not fully match the rendering in the respective Isabelle/jEdit components, the goal of this work is to find ways to improve the pretty printed VSCode output.

3 Related work

This work builds on other changes and improvements done to the Isabelle/VSCode part of the Isabelle project. At first, Wenzel introduced a VSCode extension in 2017, which can be installed in an existing VSCode (Wenzel (2017)). However, this approach had only little of the functionality known from jEdit and hasn't been continued since then. Many of the Isabelle/jEdit features need more information flow than what can be realized by an extension within the existing VSCode ecosystem. Then Denis Paluca improved the extension on various levels: He introduced a dark-mode compatible syntax highlighting and clickable items, that now could link to the corresponding places in the Isabelle theory or their definition. Notably, he also introduced a file system, which exchanges the Isabelle notations with actual Unicode symbols in the GUI, which was a significant improvement over the previous use of the third party extension "Prettify Symbols Mode", which made it inconvenient for the user to do standard operations like copying or pasting symbols and which used a lot of resources, making VSCode temporalily unresponsive on startup (Paluca (2021)). With this new approach he improved general performance on startup and with big theory files, while at the same time simplifying the setup process. Additionally, he introduced an autocomplete function for Isabelle notations, that works with the actual notation as well as with abbreviations. One year later, in 2022, Wenzel integrated VSCode into the official Isabelle repository as a bundle that can be started directly from the command line and uses Electron and Node.js (Wenzel (2022a)). In the beginning, there were still a lot of missing features compared to jEdit, like pretty printing with logical blocks, some markup bugs, a server with JSON Remote Procedure Call, a standard for sending notifications and data to a server, with different modi: with or without expecting a response (JSON-RPC) APIs, an adapted dark mode, and most of the more complex functions like

the Sledgehammer tool or a symbol panel. Some processes like the introduction of a viewport to avoid unnecessary rendering combined with more scalable messages for theory files were still not implemented at that time (Wenzel (2022b)) - so that the resulting application had still a lot of drawbacks compared with jEdit and could be considered as still being under development. Building on that integrated version, Thomas Lindae further improved `Isabelle/VSCode`: He unified the ID handling between VSCode and the backend, leaving the generation of IDs to `Isabelle/Scala`. The client sends an initialization request `PIDE/state_init` to the language server, whenever an instance of state panel is instantiated, and gets back an ID, which is associated with the state panel from this point on. He also added a more dynamic and unified handling of Isabelle options as a way to configure Isabelle beside the use of CLI commands or editing the project's preference files. For the language server he built prototypic setups with the IDEs Neovim and Sublime Text as frontends. Additionally, he fixed several bugs with autocompletion and persistence of decorations. However, most notably for this thesis, he introduced handling of the pretty printing breaks via the `pretty.scala` module, that is by default also used by jEdit. This way, a margin can be sent from the VSCode window to the backend, where the break decisions can now be computed by Isabelle's pretty printing instead of, until then, having the editor making a decision autonomously. This way, the XML tree output from `pretty.ML` can directly be used. In VSCode, this approach uses a newly introduced `Pretty_Text_Panel` component, that is used by both `state_panel` and `output_view`, and a "set margin" notification for the PIDE backend for each of the panels (`PIDE/state_set_margin` and `PIDE/output_set_margin`, respectively).

4 Error score and line count difference

Looking at the rendered code, there are several faulty indentations, that can and do happen. To have a somewhat quantitative measure on the quality of the rendered output, I categorized the missmatched indentations into four categories:

- 1. In the first category, the indentation happened almost correctly, but instead of increasing properly, the next line was indented one character **less** than the surrounding block.
- 2. The second category showed a clear indentation to the next higher block, but was several character less indented than it should logically be.
- 3. The third category lost indentation completely within the surrounding subgoal and started with the same indentation as that subgoal.
- 4. The fourth category of error lost all indentation and started at the left edge of the containing div element.

These error categories are exemplarily depicted in Figure 1. For each faulty line in the visible output frame (without scolling), the corresponding error type was noted. The visible frame was 18 lines in my setup. To account for the different severities of the categories, I weighted each category with an ascending factor (as shown in Table 1): All lines with faulty indentation were counted and the error score was then calculated by applying the weights:

$$\text{Error score} = \sum_{i=1}^{4} w_i \mathit{Category}_i$$

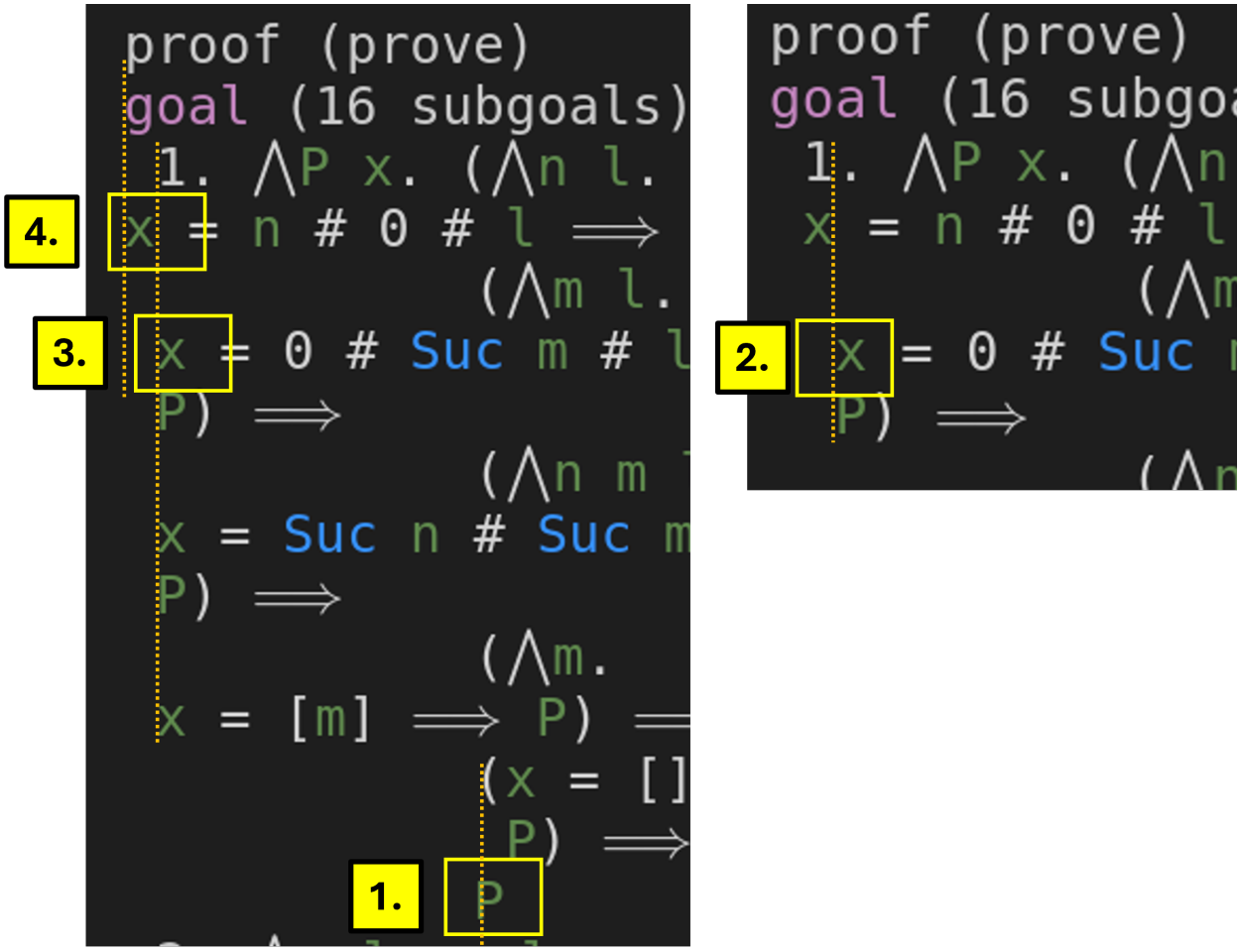


Figure 1: Exemplary depiction of the four indentation error categories with their associated indentation relative to the surrounding code.

Table 1: Weightings of the indentation error categories

Category	Short description	Weight w_i
1	Minor error	0.5
2	block scoped error	1.0
3	Subgoal scoped error	1.5
4	No indentation	2.0

Additionally, as another measure of good rendering, the difference between the lines in the output string from the `pretty_text_panel.scala` component was compared with the lines, that were measured in the frontend by dividing the text element's height by the lineheight for the specific font and font size.

5 Issue description

The general issue with the pretty printing in VSCode are often unaesthetic breaks at locations that do not fit the content of the code. In this work I concentrated on the state panel extension as the control view for output from Isabelle. In VSCode, it opens by default on the right side of the document view in an own tab group. When first opened, it shows some irregular breaks. To improve that, the window size was measured more exactly in the state panel. This exaggerated the fact, that the margin initially looking quite off - which in turn got a bit better, when the width of the container was changed, as seen in Figure 2.

(a) Before resize

```
proof (prove)
goal (16 subgoals):
 1. ⋀P x. (⋀n l. x = n # 0 # l ⟹
P) ⟹
           (⋀m l. x = 0 # Suc m # l
⟹ P) ⟹
           (⋀n m l. x = Suc n # Suc
m # l ⟹ P) ⟹
           (⋀m. x = [m] ⟹ P) ⟹ (x
= [] ⟹ P) ⟹ P
 2. ⋀n l na la.
       n # 0 # l = na # 0 # la ⟹
       Ackloop_sumC (Suc n # l) =
Ackloop_sumC (Suc na # la)
 3. ⋀n l m la.
       n # 0 # l = 0 # Suc m # la ⟹
       Ackloop_sumC (Suc n # l) =
Ackloop_sumC (1 # m # la)
 4. ⋀n l na m la.
```

(b) After resize

```
proof (prove)
goal (16 subgoals):
 1. ⋀P x. (⋀n l. x = n # 0 # l ⟹
                P) ⟹
          (⋀m l. x = 0 # Suc m # l
⟹
                P) ⟹
          (⋀n m l.
              x = Suc n # Suc m # l
⟹
              P) ⟹
          (⋀m. x = [m] ⟹ P) ⟹
          (x = [] ⟹ P) ⟹ P
 2. ⋀n l na la.
       n # 0 # l = na # 0 # la ⟹
       Ackloop_sumC (Suc n # l) =
       Ackloop_sumC (Suc na # la)
 3. ⋀n l m la.
       n # 0 # l = 0 # Suc m # la ⟹
```

Figure 2: The difference of the rendered output in the state panel on component startup and after a resize event triggered by the user changing the window size slightly.

5.1 Initial margin

That led to the conclusion, that the initial sending of the margin did not happen correctly and therefore the default margin was used by the pretty printing algorithm in the backend, leading to the frontend to break lines to fit in the too small panel - lines that were set by `pretty_text_panel.scala` with the initial default margin.

5.2 Breaks within spans

When looking at the individual spans in the rendered code, there were some container spans that had a break that led to the new line starting at the very left without any indentation. This could be explained either by the algorithm working with the wrong margin, or assuming the wrong width of the rendered characters.

5.3 Unrealistic metric

One possible reason for bad breaking behaviour in VSCode came up after comparing the relative sizes of rendered special symbols $\Longrightarrow$ and $\bigwedge$ with the output for the same symbols from the currently used `Symbol.Metric`: The rendered relative sizes were 2.987 and 1.498 respectively, while `Symbol.Metric` returned 2.0 and 1.0. This underestimation of the width of symbols by the prettyprint algorithm should naturally lead to lines that are to wide to fit the output window (in our case the state panel window). This was the explanatory approach most of the following work concentrates on.

6 Methods

In the following section I will describe the approaches I took to improve the rendering of the Isabelle output. Some approaches yielded only information and were not used in the final version of the code submitted along with this thesis.

6.1 Prototype extension

In a first approach to measure the width of a text including some symbols we first wanted to show, that it is possible to read the width of the rendered text as well as the width of the available space from the parent extension element. With the text width and the character count of the input string it was now also possible to calculate a mean character width, which seemed a possible approach to feeding the pretty printing algorithm with more exact data.

6.2 Mean character width of content

Following this direction we intercepted the output text, rendered it invisibly in one line, measured the width, and with that information calculated a mean character width of the actual output. The idea was to calculate the margin in the state panel (and later on in other extensions using the PIDE backend to render formatted code output). The approach would have enhanced the formerly used way of measuring by giving a real average for every output text. Currently, the character width is measured in `main.js` by rendering the string ”mix” and averaging the measured character width, which is pointless for a monospaced font and doesn't take symbols into account. However, also averaging the mean character width of text with symbols doesn't help the `pretty.scala` component in the Isabelle/Scala backend to make better decisions about line breaking, since the symbols are not evenly distributed in the output. And even then, measuring string length with an average character width would be far from exact.

6.3 Actively setting margin

So in the next step we established a route to actively calculate the margin from the mean character width and the width of the parent element in the state panel, send this margin via the language server to the Isabelle/Scala backend and trigger an update of the state panel. That way we could manipulate the sent margin and adapt it to the mean character width measured directly in VSCode. Also, the way the margin was measured, was improved: up until now, the margin was calculated in `main.js` with the `window.innerWidth`, which means that the whole component width is measured, even if the effectively usable text field inside the component without scrollbars or padding is smaller. In case of the state panel, the textfield is an element tagged with ”isabelle-content” nested within other html elements, has 20px padding on each side and optionally an approximately 10px scrollbar on the right.

In the new approach, the width is now calculated with this actual ”isabelle-content” element, that is set by `state_panel`. The difference between the measured `window.innerWidth` and the actual element containing the text is about 40-50 pixels because of aforementioned padding and scrollbar. The same level of inaccuracy was present in the measurement of the symbol widths: the script in `main.js` measured an up to 5% bigger character width compared to the actual rendered

characters, because it doesn't wait for the "Isabelle DejaVu Sans Mono" font to be loaded, which is later used for the output. These inaccuracies led to a margin which was up to 10% bigger than the actually available space. The method in `main.js` therefore substracted a fixed value 16 from this margin before sending it to the backend, leading to a significantly underestimated margin being sent to the backend - especially with narrow panels.

6.4 Initial margin

To fix the missing/wrong initial setting of the margin on startup and switch the flow to frontend measuring and active transmission of the margin to the backend, the previous way via "`resize`" was replaced by a specific "`state_panel_resize`" transmitting the margin calculated in the state panel with the right context parameters (as described above). The previous "`resize`" was sent by `main.js` and used `window.innerWidth`, while `state_panel_resize` used the actual textfield width (as described above in Subsection 6.3 in detail).

6.5 Custom metric

The default character width metric used in VSCode, `Symbol.Metric`, works with only four distinct sizes for any character: from one, which corresponds to one Unicode monospaced character, to four. Each size is the factor to one monospace character and derived from the count assigned to the substring "long" at the beginning of the symbol name. However, Isabelle's LaTeX style symbol notations of the original code were replaced with a Unicode symbol on the fly by Paluca's SymbolEncoder (Paluca (2021)) based on a symbols table, leading to the text shown in VSCode containing real 8-Bit Universal Coded Character Set Transformation Format, the most commonly used character encoding for the output of human-readable characters (UTF-8) symbols. This fact in combination with the used `Symbol.Metric` heuristic leads to the calculated string lengths differing from the reality of the rendered string in VSCode, whenever it contains non-ASCII symbols. Therefore the idea was to introduce a custom `VSCode_Metric` that holds a table with non-ASCII symbol lengths associated to their codepoint value and uses them in it's `apply()` method, where the length of a string is calculated. `VSCode_Metric.apply()`, loops through the string, looking up, wether a symbol exists in the width table, in which case it uses the associated relative width, and otherwise using the unicode default of 1.0. The widths in a string are added and the resulting float number is returned. The `apply()` method of the character width metric is called by the `pretty.scala` class to define the actual breaks based on the XML markup tree it gets from the `pretty.ML` algorithm of the Isabelle core.

6.5.1 Measuring Characters and Symbols in VSCode

To measure the width of a character, it is rendered in an invisible span element with the same font, font size, and font weight. The width of that span is then returned and the span removed, as shown in the source code fragment in Listing 1. FontFamily and FontSize are only used, after a asynchronous await function makes sure that the `document.fonts` are available. The method is called for all available symbol codepoints and the resulting widths are stored in an array together with their corresponding codepoints.

```
function measure_char_width_px(ch) {
  let span = document.createElement('span');
  span.textContent = ch;
  span.style.visibility = 'hidden';
  span.style.whiteSpace = 'pre';
  span.style.fontFamily = root.style.fontFamily
  span.style.fontSize = root.style.fontSize
  span.style.fontWeight = cs.fontWeight;
  root.appendChild(span);
  let width = span.getBoundingClientRect().width;
  span.remove();
  return width;
}
```

Listing 1: Measurement of character widths in `metric_agent_script.js`

This ”width table” array is in later implementations subsequently sent to the backend via LSP (detailed in Subsection 6.5.3).

6.5.2 For defined symbols

The next step was to establish a procedural flow from the VSCode editor to the Isabelle/Scala backend, in order to use the resulting width table as input for the `VSCode_Metric`. I started with a function using a fixed set of frequently used symbols, rendered them in the output view context, and measured them. With that setup I implemented the transmission of relative character widths into the backend, a first setup to feed that into the new `VSCode_Metric`, and use it in the state panel instead of the existing `Symbol.Metric`. In order to achieve that, the pretty text panel had to have a new parameter, which was a function returning either the new custom `VSCode_Metric`, if values from the frontend were already fed into it, or otherwise the `Symbol.Metric` which is the default character width metric used for prettyprinting with VSCode.

6.5.3 For all symbols from backend

However, the aim was to have a dynamically adjustable way to feed all relevant info into the `VSCode_Metric`. Therefore I improved the implementation, by making the state panel actively fetch the symbols from the backend, and using the symbols' codepoint data to measure them in the state panel/output view context as shown in Figure 3. In this flow, the symbol widths were measured once an instance of state panel was opened and transmitted together with the ID of the panel in order to couple the use of the `VSCode_metric` to a specific instance of the state panel.

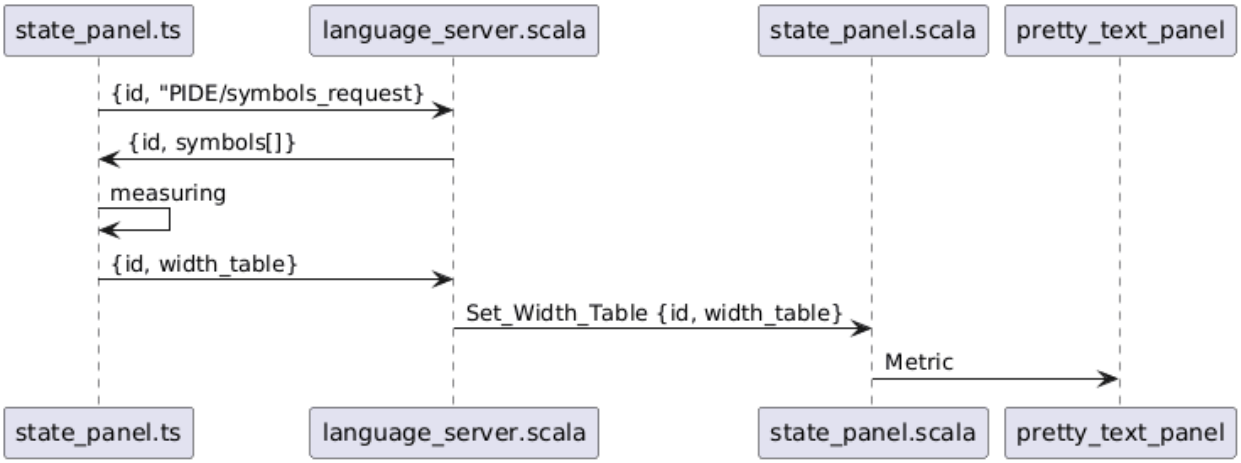


Figure 3: Sequence of fetching the symbols from the backend, measuring widths in the frontend by the `state_panel`, and sending it as metric to the `pretty_text_panel.scala`

This implementation now had the complete set of symbols, but the width table lacked persistency and because of the coupling to the specific instance of `pretty_text_panel.scala`, the measurement had to take place each time a state panel was opened, which imposed a considerable overhead.

6.6 Switching between metrics

To have a visual comparison between the use of the new `VSCode_metric` and the standard `Symbol.Metric`, I introduced a checkbox into the state panel which would activate a switching between the two metrics in the backend and reload the state panel. The user could also open different state panels with different states. This was possible, since each instance of the state panel would hold it's own metric. Later on, the character width metric was decoupled from individual instances of state panel and instead set globally (but still with the possibility to switch).

6.6.1 With persistent storage

To keep the data available in the backend and to avoid measuring it on each startup, the results of the measurement are stored in a JavaScript Object Notation, open standard file format to exchange data in a human-readable way (JSON) file by the small component `vscode_metric_store.scala`. The data is stored together with a fingerprint as an identifier - a string made from the font and the font size. We chose to include the font size to be as accurate as possible, even though the sizes differ only marginally between font sizes of the same font. We compared font sizes 18 and 20 of the standard "Isabelle Dejave Sans Mono" font and got deviations of +/- 0.2%, probably mostly due to rounding errors. However, storing several data sets only needs minimal storage resources (about 5kB/set of ratios) and no computational overhead when accessing.

6.6.2 In an own extension on startup

To detach the measurement from the state panel, it was moved to it's own extension `metric_agent`, which gets started from the general extension component as soon as the language server is up. As shown in Figure 4, on startup, the extension `metric_agent` collects the editor's font and font size, sends a fingerprint to the language server to get the information, wether there is already a width table stored for that fingerprint. The language server hands it down all the way to the `Metric_Store.scala` component, which decides, wether there is a stored width table matching the fingerprint. The answer contains a boolean to indicate if measurement is necessary - and if it is, the language server also appends an array with codepoints of all special symbols. Dependent on this, `metric_agents` renders all symbols in hidden spans with the same context as `output_view`, calculates the ratios and sends them to the backend as an array.

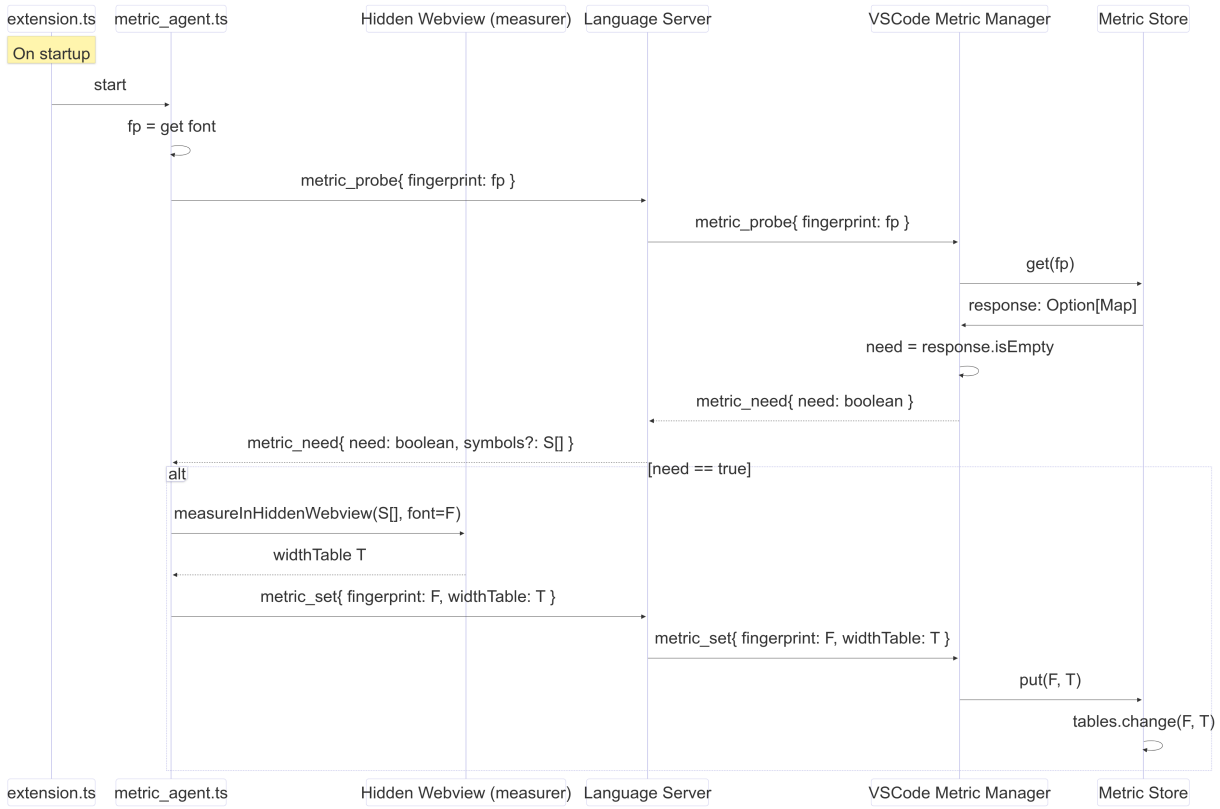


Figure 4: Sequence of measuring symbol widths in the frontend by the `metric_agent` extension and processing in the scala backend

To be sure, that the `metric_agent` uses the exact same settings as the state panel in regards to CSS font and text properties, I also used the same way, in which the state panel as well as the output view are implemented in VSCode. The state panel component puts it's content inside the content of `output_view` by calling `Output_View.get_webview_html` with it's own content as parameter, as shown in Figure 5. The same is now done by `metric_agent` to gain access to the settings and fonts and keep the conditions identical when measuring the symbol widths. Output view is the central component for showing rendered Isabelle output in state panel, as well as in it's

own ”Isabelle: output” view. Using it with `metric_agent` standardizes the context for the rendered symbols.

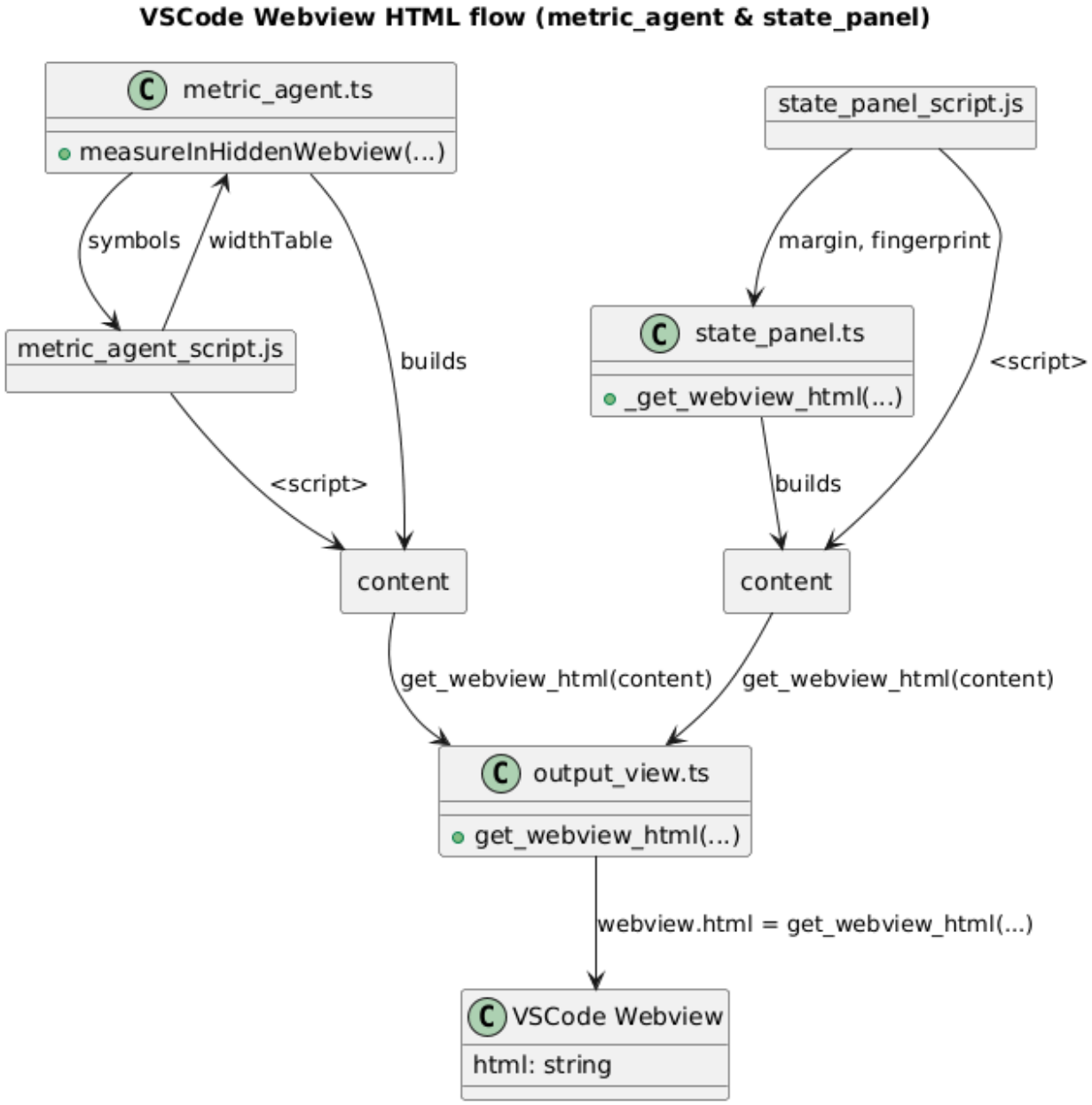


Figure 5: `metric_agent` reusing the same webview for state panel, metric agent, and output view.

6.6.3 Implementation of a helper class to log metric to file

In order to assess the differing relative sizes of `VSCode_Metric` and `Symbol.Metric`, a small helper was introduced that reads out the final values from the two metrics for all symbols and stores them in .csv format. That way we could compare the differences for every symbol that is included in the `component_vscodium.scala` symbols object (stored in `etc/symbols`).

6.7 Including dynamic *breakgain* parameter with Pretty.formatted()

One possible parameter of `pretty.scala`'s `formatted()` method is the breakgain. It gives the the pretty algorithm some tolerance to stay in the line, even if the block does not fit the line and overshoots a tiny bit. The default breakgain in `pretty.scala` is set at $default\_margin/20$, with `default_margin` being 76. The breakgain is one optional parameter in `Pretty.formatted()`, and if it's not set, the default breakgain is used. I discovered, that `pretty_text_panel.scala` doesn't set the breakgain, when calling `Pretty.(formatted)` on the separated output trees (see Section 2.1). This leads to the fixed default breakgain being used ($= 76/20 = 3.8$), which might be tolerable in wide windows or when the margin sent is significantly smaller than the real available space. The first condition is met, when the output is shown in the output view right under the document view (since it is wider than the state panel, which is opened in a split view beside the current document by default). The second condition is partially met in the current version of Isabelle/VSCode, since the margin is always reduced by 16 before being sent on resize events. However, with narrow panels or less tolerance (which happened, when I improved the accuracy of the measurement), the lines break too late and therefore get broken on the client side in VSCode - leading to the loss of indentation. I fixed this problem of a static, margin-independent breakgain by setting the parameter as $margin/20$, when `pretty_text_panel.scala` calls `Pretty.formatted()`. This way the breakgain is always set relative to the currently used margin.

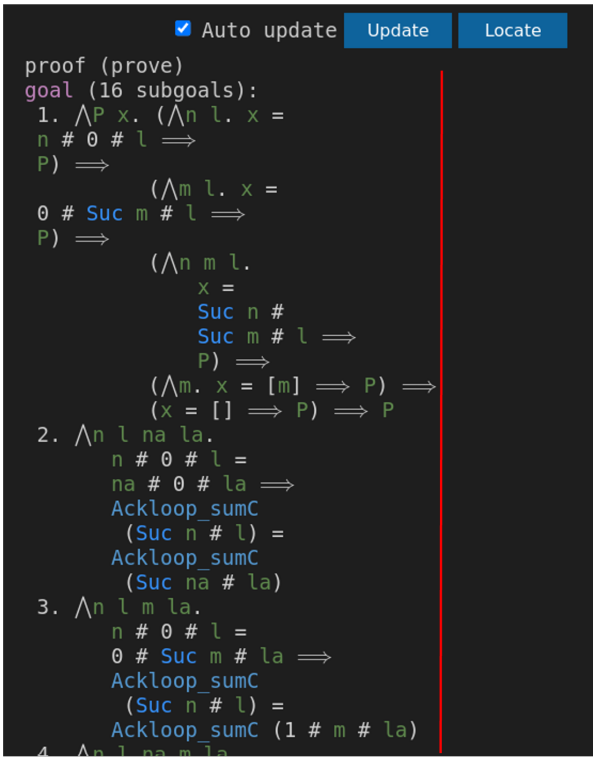
7 Results

The introduction of the new `VSCode_Metric` and the improved measurement of the state panel width lead to a visual improvement in a lot of the examined cases. In most cases, the output broke differently - which was in part a result of the more precise and overall wider measured panel width and therefore a higher margin. In some edge cases, the output looked worse with the new metric, breaking at illogical places. The superfluous breaks introduced at html level were removed by the introduction of the `VSCode_Metric` in all examined cases.

7.1 More exact width of parent elements and characters

The window width could be measured more exactly in the state panel. Before, the window width was only approximated and did not contain margins around the actual text box as well as space for scrollbar. To make up for that inaccuracy, a fixed margin of 16 was subtracted, which lead to a general underestimation of the available space. This underestimation was particularly large when the window was narrow - leading to unnecessary long, very narrow formatted code output. The more exact width measurement directly improved the appearance of the text at most widths

(a) Measurement in main.js



(b) Measurement in state panel

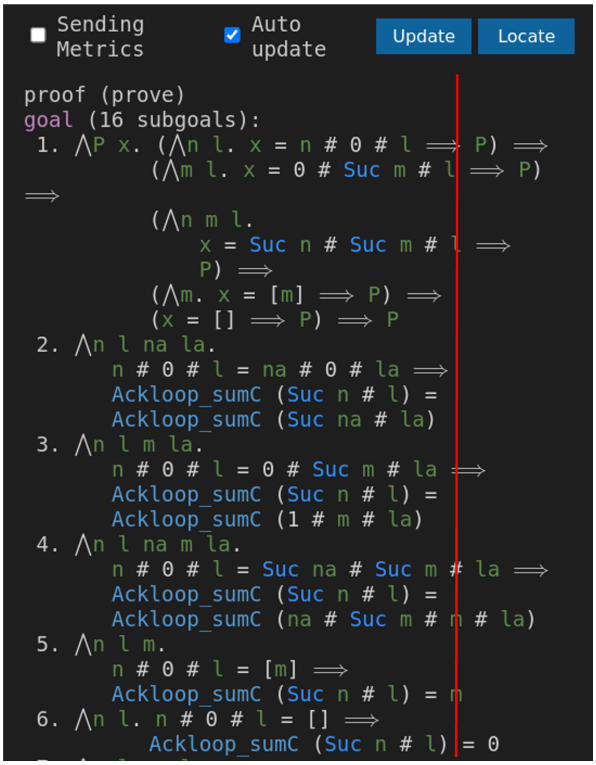


Figure 6: Differences in the formatted code between the original margin calculation (a) and the improved way of measuring in the `state_panel` itself (b).

Figure 6 shows, how the lines always end at about two thirds of the width in the left panel ((a) the original way of measuring in `main.js`) - marked with a red line. Compared to that, the variant

with the state panel measurement (b) uses all available space. That leads to a display in less lines of code, which enhances the visibility and legibility of the shown output.

7.2 Effect of initial margin

Sending an explicitly measured initial margin at the startup of the state panel lead to an improved breaking from the beginning (and before the window is changed by the user), as can be seen in Figure 2. However, it didn't abolish indentation errors completely.

7.3 Rendered symbols vs metric

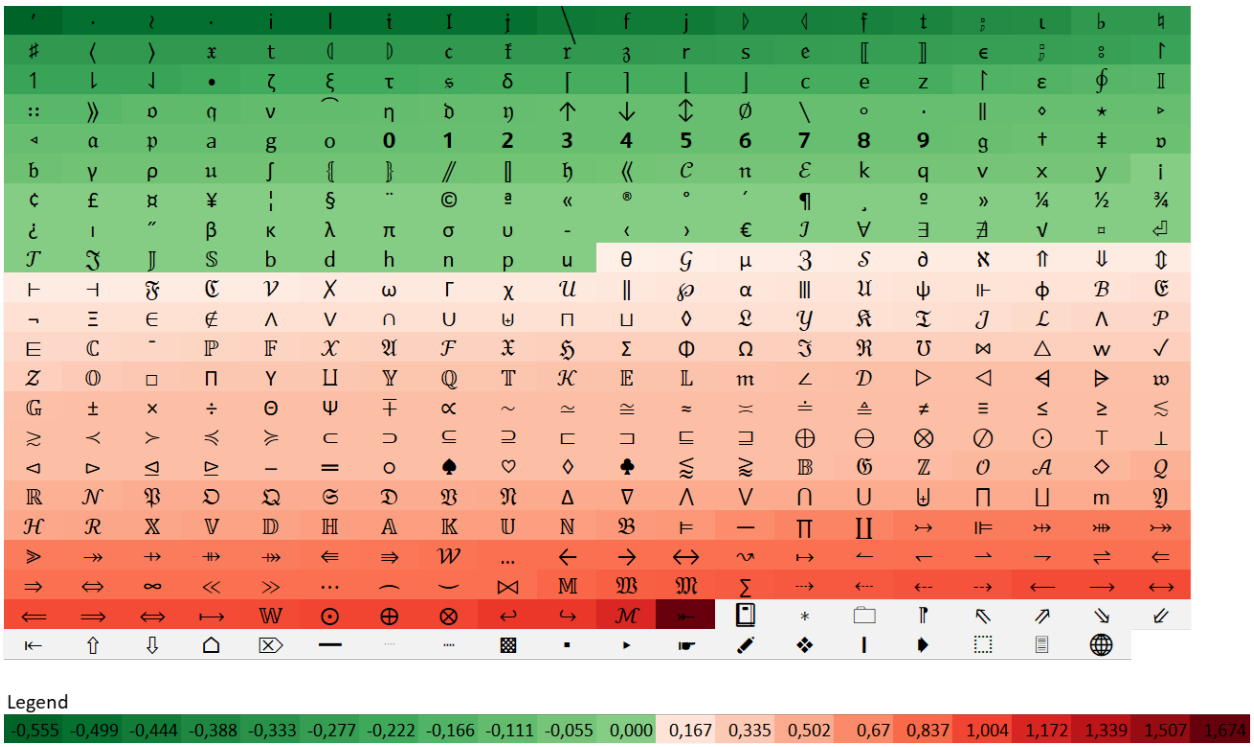


Figure 7: Heatmap of all symbols that get send from the backend. For each symbol, the relative unit from the Symbol.Metric is substracted from the one stored in `VSCode_metric`. Negative values indicate symbols that render smaller than assumed by Symbol.Metric and positive values indicate symbols that render bigger. Values were derived in `pretty_text_panel.scala` once at the first metric change.

When the relative widths of the symbols were exported via the `VSCode_Metric_Dump` helper and the current Symbol.Metric was compared to the improved custom `VSCode_Metric`, it became obvious that the differences are quite significant and affect most of the symbols - as can be seen in Figure 7. Since the influence of the font size on the relative character widths were minimal (+/- 0.2%), I decided to just use one configuration in order to not further complicate the figure. The figure shows that a big part (171 of 440 measured symbols) renders smaller than the `Symbol.Metric` assumed. These symbols do not cause problems in rendering the Isabelle output, since they can only

lead to underfull lines, which happens in most lines anyway. Then there is a part of the symbols that renders as expected, which obviously also should not cause problems. The interesting part for the rendering of Isabelle output are the symbols, that render bigger than expected: they can cause irregular line breaks in cases where the line sent from the Isabelle backend does not fit the width of the text container, since one or more symbols were underestimated and the output line uses the whole available margin. The characters rendering bigger than assumed by Symbol.Metric contain some widely used symbols like implication arrows $\Longrightarrow$ and logical operators like $\bigwedge$ or $\bigvee$. For the highest difference the measured symbol takes 1.67 times the space that the Symbol.Metric assumes. Apart from just the relative difference it is also important to note that with bigger symbols this relative change has a bigger effect on the ability to break a line that is not meant to break. The last class of symbols are the ones that are stored as zero width symbols in Symbol.Metric. These symbols are not meant to be rendered, but are used in combination with other characters to signify text formatting.

7.4 Effect of the new metric

Going through the example files provided in `HOL/Examples`, `HOL/Isar_Examples`, and `HOL/Proofs` at different places in the example code with different window widths of the state panel, we found that about 5% of the generated output showed differences between the lines counted from the output of `pretty_text_panel.scala` and the effective line count calculated by analyzing the rendered output in the state panel. This is exemplarily shown in Figure 8: With the `VSCode_Metric`, there is no deviation, while with the currently used `Symbol.Metric`, the rendered output is two lines longer than one would expect from the formatted and stringified `pretty_text_panel.scala` output (by counting the newline characters as detailed in Section 4). The percentage of affected configurations (width and caret position) differed greatly from file to file, depending of the amount and kind of non-ASCII symbols contained in the output. In all observed cases, the output line count from the scala backend matched the rendered output line count, when the new `VSCode_Metric` was used.

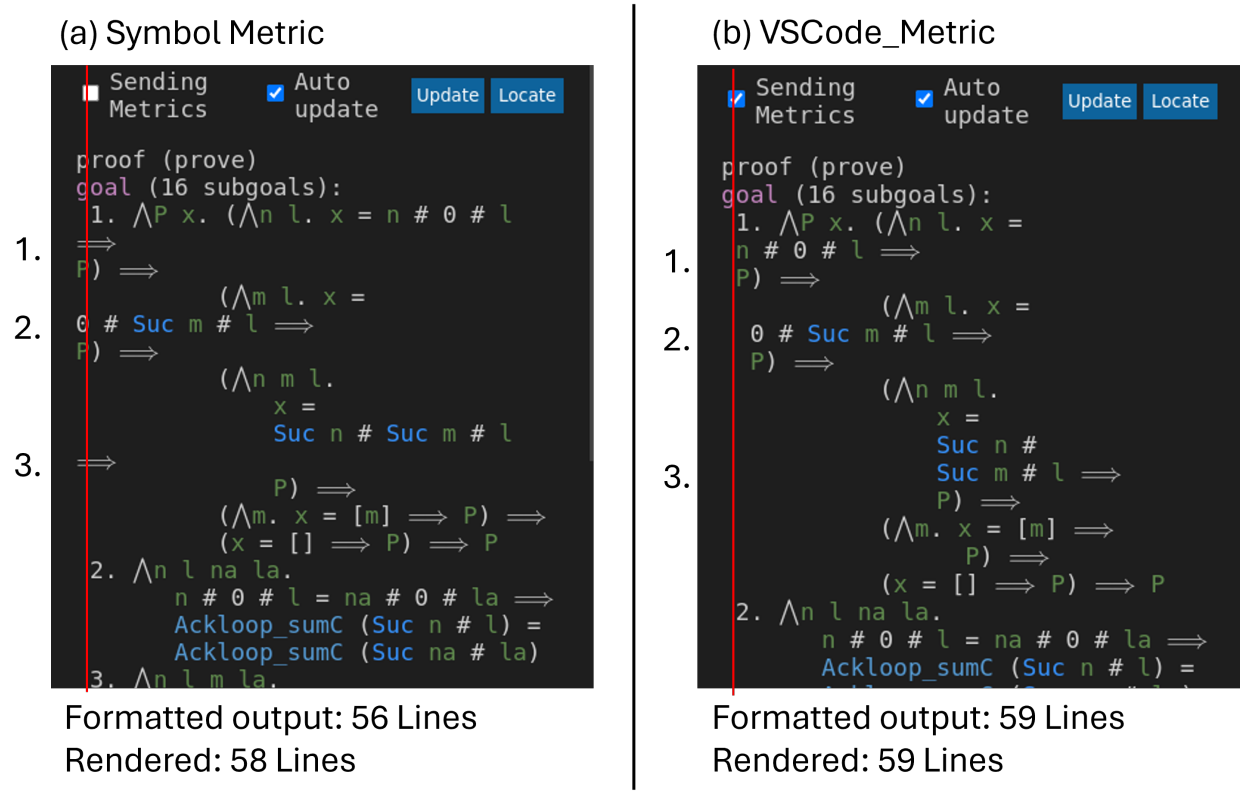


Figure 8: Different state panel outputs exemplarily shown at the same width with (a) the `Symbol.Metric` and (b) the new `VSCode_Metric`. The red line marks the indentation of the highest level in the output. The difference in the according line counts is shown below the screenshots.

I also noted, that the indentation got completely lost in some cases with the `Symbol.Metric`, leading to the subsidiary blocks being even less indented than the blocks of the highest level, as can be seen in Figure 8 at the marked places 1., 2., and 3. in the comparison between (a) and (b). Also striking is the difference in indentation in (b) between places 1. and 2., while the same places in (a) don't show any indentation and therefore no difference. However, also in (b), the indentation did not match the logical structure of the code, since the indentation within the first two blocks of the first subgoal got lost. Overall, I also observed that the output before rendering changed in many cases even though there were no superfluous line breaks in the rendered code with the original `Symbol.Metric`.

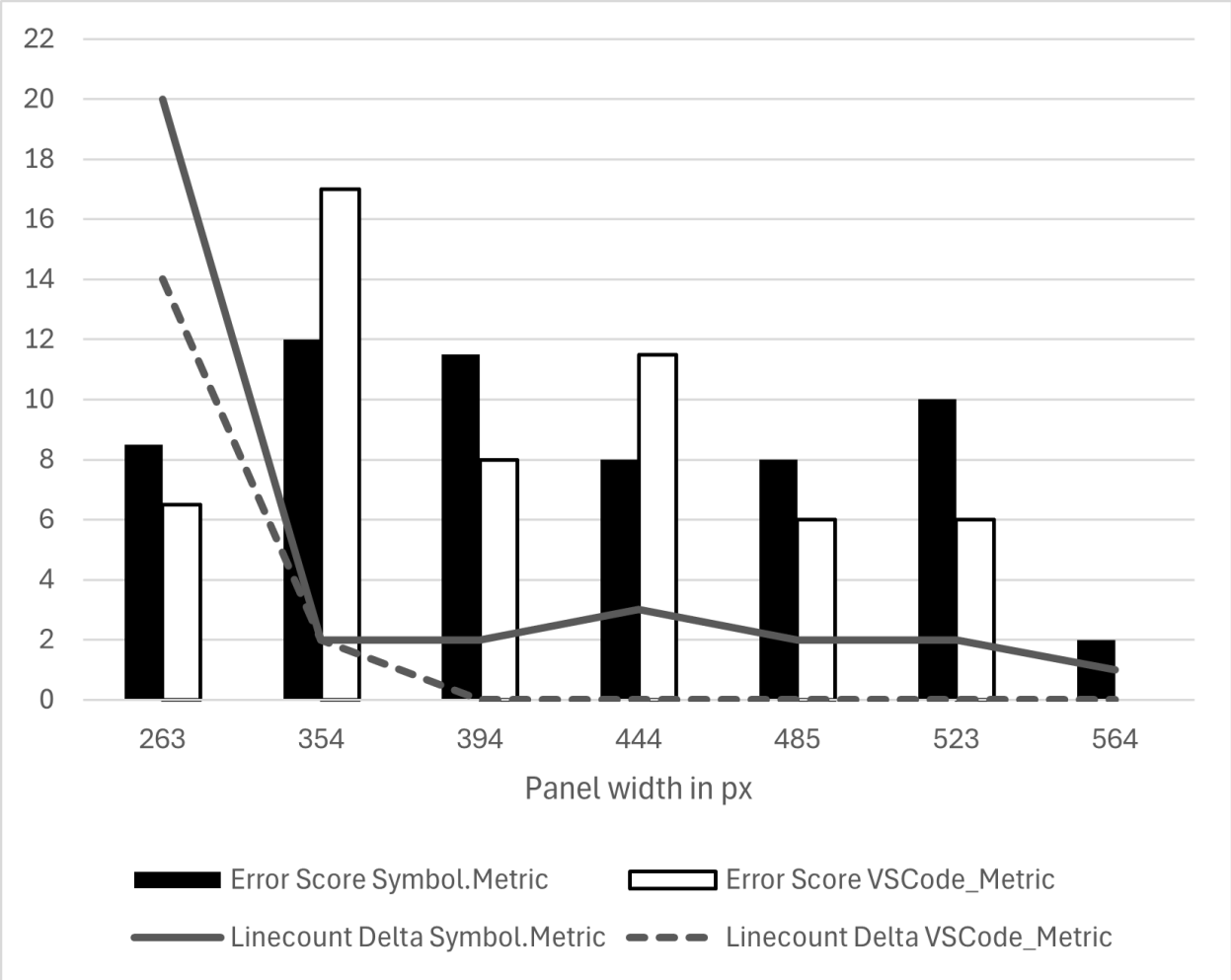


Figure 9: Code output in the state panel was compared between `VSCode_Metric` and `Symbol.Metric` at different component widths. The specified width is the width of the inner text field of the state panel. Error scores were calculated by using the procedure described in Section 4. The Linecount Delta is the difference between the lines in the output string (by counting `\n`) and the lines counted in the VSCode panel (by dividing the height of containing div by the measured line height). The data were collected using the example of Ackermann.thy

In Figure 9, there are differences visible between the usage of the `Symbol.Metric` and the `VSCode_Metric`, but the trend towards `VSCode_Metric` having less faulty indentations is

unambiguous. At some widths, as 354px and 444px in this example, there was a higher error score for `VSCode_Metric`. The difference in line count showed no superfluous line breaks, when `VSCode_Metric` was used above a threshold of about 360px available text width. However, below that threshold there were superfluous line breaks and at the narrowest measurement, the amount went up significantly with both used metrics.

7.5 Effect of the margin-adjusted breakgain

The active setting of the breakgain parameter for `Pretty.formatted()`, as described in Subsection 6.7, lead to a greatly improved output code quality, as quantified with error scores in Figure 10.

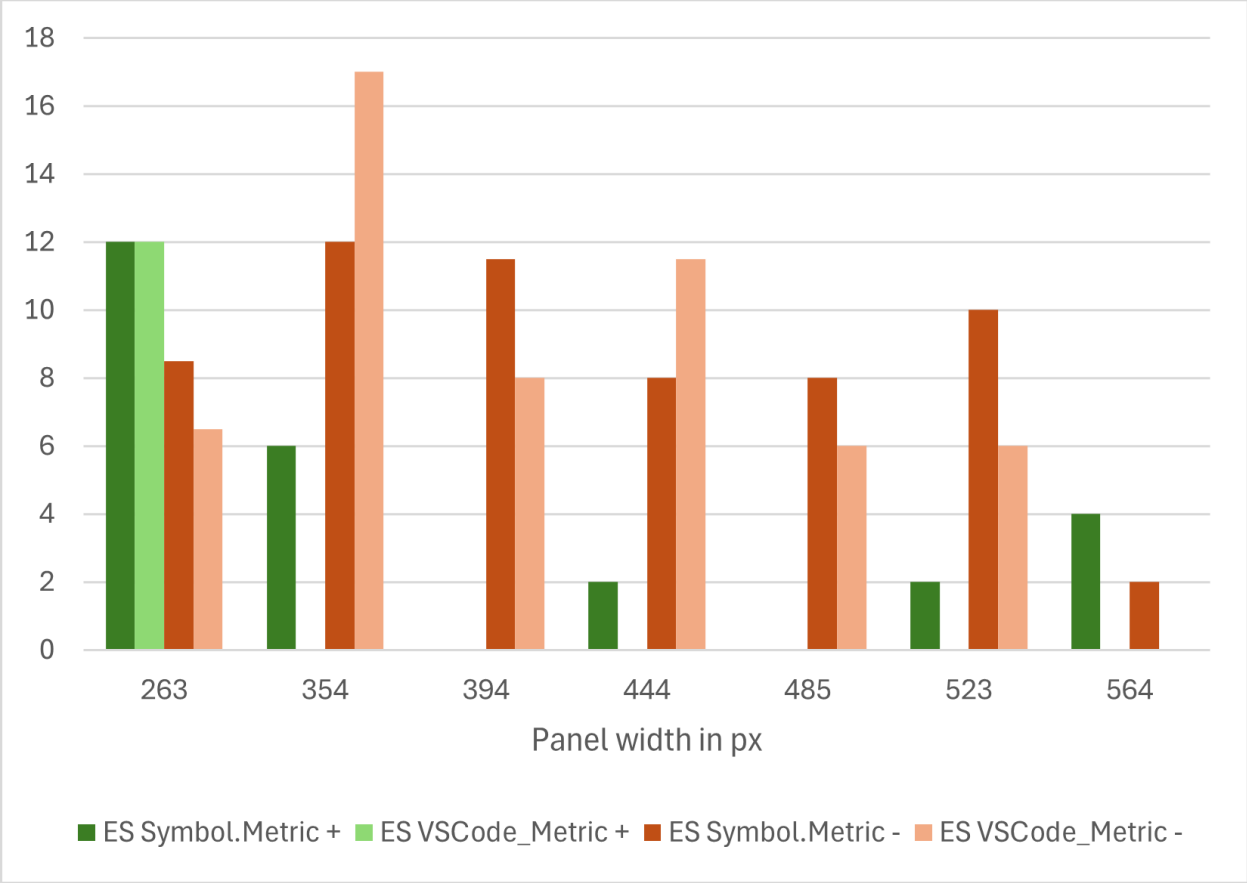
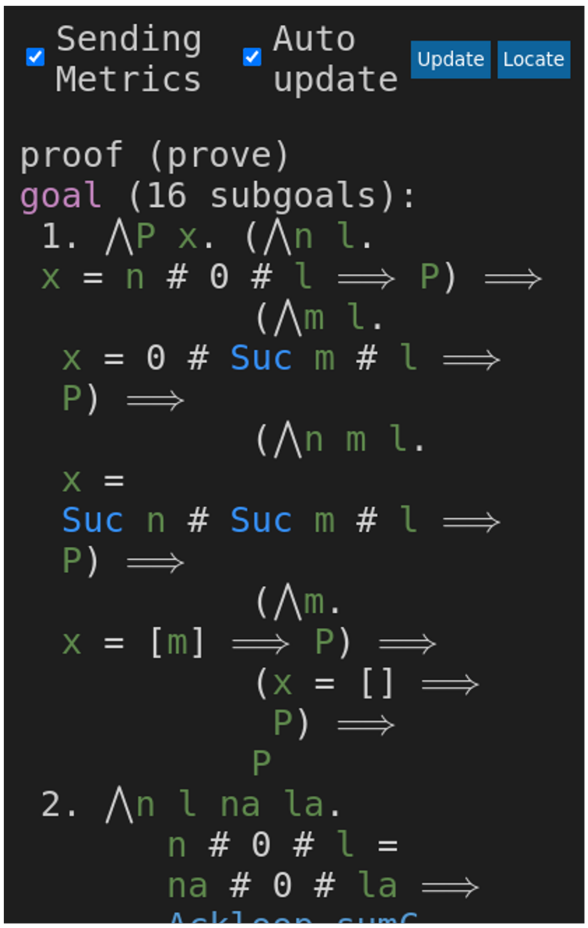


Figure 10: Error Score (ES) of the output in the state panel at different widths with the example file Ackermann.thy. The (-) marks the output without setting breakgain in `Pretty.formatted()` (red), while the data marked with (+) is the output, when `margin/20` is set as breakgain parameter (green).

With the newly introduced setting of the breakgain parameter, the error score dropped to 0 for all panel widths above 263, when combined with the usage of the new `VSCode_Metric`, while with the currently used `Symbol.Metric`, it dropped significantly - to only a fraction of the indentation error score. That is congruent with the qualitative evaluation of output prettiness, an example of which is shown in Figure 11.

(a) With default breakgain



(b) With actively set breakgain

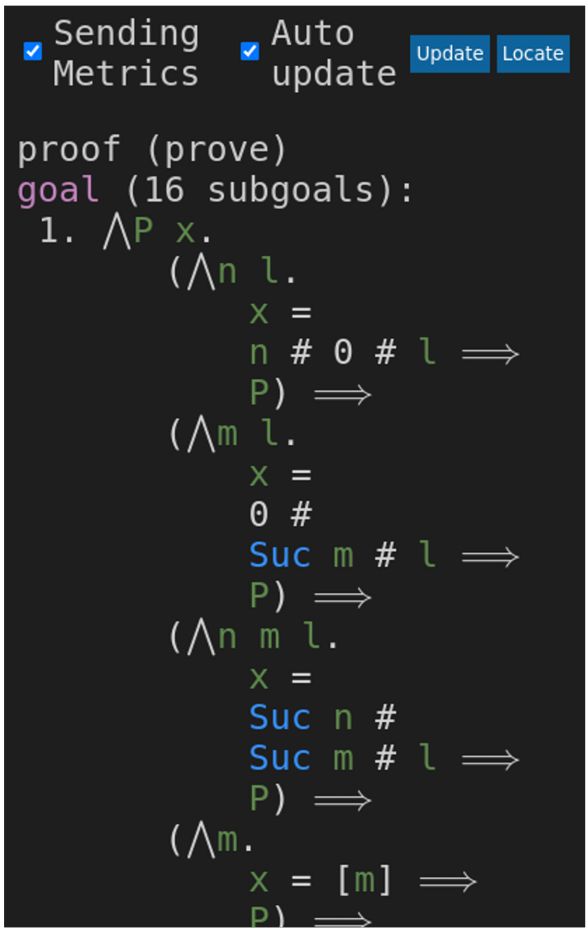


Figure 11: Qualitative difference between output in state panel, when breakgain parameter is not set (a) and when it is set to margin/20 (b).

8 Summary

Before discussing the results, I want to quickly summarize the findings and improvements made. The state panel width is now measured exactly and sent to the Isabelle/Scala backend as an margin calculated within the panel - replacing the more general, but also more inaccurate implementation via window width in `main.js`. We were able to measure the symbol widths in the frontend under the exact same conditions as in the `pretty_text_panel` component and use these measured widths as the basis for string length calculations in the `pretty_panel` component of the Isabelle/Scala backend. This was done by using a custom `VSCode_Metric`, that can take input and use it to calculate more exact string lengths, which then get used by `pretty.scala` for it's break decisions. The measurements for the new character width metric are persistently stored in JSON format in the backend. With a fingerprint of the font family and the font size and a handshake between frontend and Isabelle/Scala backend, new measurements only take place, when one or both of the font characteristics change. Depending on wether there are data for the fingerprint, these data are used or a new measurement is requested from the frontend. To allow for a comparison of the newly introduced character width metric and the one used to date, there is also the possibility for a client-side request for switching between `Symbol.Metric` and `VSCode_Metric` for the backend-generated output. If there is no data stored and new data cannot be acquired, `Symbol.Metric` is used as a fallback. To be able to compare the exact output for all symbols, we stored the widths for the symbols for both metrics. Comparing them side by side made clear, that the differences are significant and explain at least part of the erratic breaking in the rendered output in the state panel. This could be observed by comparing the line count of the output in the backend with the linecount of the actually rendered output. This means, that in the cases, where the line count of the rendered output was higher, the editor itself broke lines which were too long for the panel's text width. The achieved more exact output means a further approximation to the outcome of jEdit's pretty printing results. It can't be reproduced in the exact same way, since jEdit uses the UTF-8-Isabelle encoding, which cannot be used in VSCode. To bypass that problem, Denis Paluca had introduced a file system that replaces the Isabelle notations (like `\<Longrightarrow>` for $\Longrightarrow$) with symbols from the UTF-8 encoding that visually match the corresponding symbol (Paluca (2021)). Now we found a way to improve the application of these symbols to get again closer to a "pretty" pretty printed output and eliminate differences between the output from the pretty.scala component and the actual rendered output.

9 Discussion

9.1 Margin in VSCode

Since we now have a better measurement of the margin in the VSCode state panel component, one question is, how and if this can be generalized to the other places, where pretty printed code is displayed - namely the output view panel and other panels yet to follow, for example the Sledgehammer panel recently implemented by Diana Korchmar (Korchmar (2025)). Of course the measurement could be implemented in each panel separately - however, a way with less code duplication could be the placement of the measurement in the common `Output_View.get_webview_html()` function that returns the content for both, state panel and the output view panel.

9.2 Why the difference in output?

Another question is, why the output with the `VSCode_Metric` looks at some instances fundamentally different than when the `Symbol.Metric` is used. Especially important would be an explanation, why the indentation of a block in the output code is often lost, leading to hardly readable outputs. In this point, the two metrics also had a qualitative difference: In the standard `Symbol.Metric` rendering, the indentation was sometimes even less than the subgoal indentation, which I didn't witness with `VSCode_Metric`. This loss of indentation hints at a general problem in `pretty_text_panel.scala`, where margins are taken into account or in `pretty.scala` itself. The error score done exemplarily for `Ackermann.thy` at different widths (see Figure 9) showed, that there were no line count differences when using `VSCode_Metric` above a certain width threshold (about 360px). However, below that threshold, both metrics produced a line count difference. The difference accelerated towards very narrow panel width. This can be partly explained by narrow width results in more lines, which in turn results in more possibilities for a mismatch and therefore client-side breaking. Additionally, shorter lines mean that small inaccuracies in width can have a bigger impact. The algorithm generally seems to have problems with very narrow margins - this is not solely a VSCode problem, but also happens in jEdit. However, in jEdit, the state output is rendered below the document view instead besides and the possibility for the user to shrink it, is also limited. It's only possible to make it very narrow by downsizing the jEdit window itself. One reason for problems with small margins (at least in VSCode) is, that `pretty_text_panel.scala` doesn't send the parameter "breakgain", when calling `pretty.scala` in the `refresh()` method. This leads to breakgain being calculated by the default margin, which, with a value of 76, is relatively high. This in turn leads to a high tolerance for breaking and leads to breaking late, making client side breaking more likely. And indeed, when I changed the call of `Pretty.formatted()` to include a breakgain based on the current margin, the output quality improved dramatically. It brought the error score down to zero when using `VSCode_Metric` with a panel width above 300px and reduced it greatly when using `Symbol.Metrics`. It also led to more lines in the backend output and fewer instances of client-side breaking. Unfortunately, I only found out this causality in the very last days of writing this thesis, so there is no time to dive into the potential for further improvement.

One reason for the differences between the metrics being not as clearcut as expected may also be the "error score" I used to measure the quality of the output. The categories are based on what I deemed right - as are the weights. As all calculated scores that aim to integrate more complex observations into one numeric key figure, this error score also loses information for the sake of comparability. I tried to keep it as simple as possible, but may have introduced my own biases into it. Regarding the indentation errors, it would also be very interesting to investigate why indentation is sometimes lost completely, causing some lines to be less indented than even the surrounding subgoal. To do so, it will be necessary to focus more on the `pretty.scala` component.

10 Conclusion

It definitely makes sense to use the newly developed custom metric. It is very adaptable and doesn't use a lot of resources. It can improve the appearance especially in outputs with symbols whose rendered width ratio deviates strongly from the currently used symbol metric. Also, it performs better in very narrow windows like hover overs texts. Additionally, the improved measurement in the

state panel leads to more of the available space being used, which further enhances readability. The initial setting of the margin through ”resize” removes the broken view at startup. However, one of the biggest impacts on output prettiness had the setting of breakgain, when calling `pretty.scala`. With this small change, the output improved significantly, both in a qualitative way as well as with the quantitative measure of the defined ”error score”. This was true for both metrics, but especially in combination with `VSCode_Metric` it lead to perfect output in all viewed cases above the width threshold of about 300px.

11 Future work

Some additional features from jEdit were recently introduced into VSCode by Diana Korchmar: The symbol panel, which allows the user to select one symbol by a simple click on one of the buttons that are organized in different tabs within the panel. Then the sledgehammer tool, that can send automated requests to several external theorem provers, translate the responses into Isabelle proofs, and present them to the user. The third feature Diana added to Isabelle/VSCode, was the documentation panel, which shows all Isabelle related documentation and allows in-editor viewing of all necessary formats - of which PDF was especially complex to display. (Korchmar (2025)) These improvements are not yet introduced into the current official release. However, there might arise the necessity to adapt my work on the metrics to the possibility for the user to introduce and define new custom symbols or adapt Diana’s features to some of the changes introduced in my work.

The next steps towards a better pretty printing should probably focus on finding the reason, why indentations are lost within logical blocks. It becomes obvious, that the main cause of the faulty breaking lies within the `pretty.scala` or even the core `pretty.ML` components, when we compare jEdit state output with the VSCode state output at the same margin: after applying the new margin calculation and the `VSCode_Metric`, the state outputs in jEdit and VSCode break the same. This isn’t surprising, since both panels use the same components to calculate the actual line break and now both get accurate input. To achieve an improvement, the algorithms have to be thoroughly analyzed by comparing the XML output trees with the stringified output with a special attention to the number of spaces at the beginning of each line and if it fits the indentation of the logical block. To make sure that the `VSCode_Metric` always fits the currently used font and size, an update mechanism should be introduced, which triggers the `Metric_Need` handshake, whenever font or size are changed. Since the size only has a negligible effect on relative character widths of the symbols, the emphasis should be on the font. However, to really support different fonts, the process would have to be adapted. With a non-monospaced font, it would have to measure all different ligatures and text formatting, which would go beyond the possibility of what is achievable with the current setup - especially the simple storage structure. However, being able to use fonts, that are not the customized Isabelle font, would bring the Isabelle/VSCode extension closer to a real standalone VSCode extension - which, in turn, would radically change the ways, how users are able to reuse their existing VSCode setup. A first step could be to measure symbol width ratios with different text formats like superscripted, subscripted, fat, or italic. These formats could have an impact on the relative size of rendered symbols and therefore be necessary to correctly print formatted text. For storing many different varieties of the width tables, it may make sense to switch to another storage structure. One possibility would be SQLite, since it has way faster lookups and there are

scala libraries for this format (e.g. SQLite4S). A very small improvement to the `metric_store` without changing the underlying data structure could be a managed storage: instead of storing all tables for all fingerprints ever used, it could store the last couple of fingerprints with their associated tables. When looking into the usage of `Pretty.formatted()`, I found, that in most instances, breakgain is also not set by Isabelle/jEdit. Setting it there may also improve the output prettiness in narrow jEdit windows. That said, it may make sense to make `Pretty.formatted()` independent of it's input and to dynamically set the breakgain relative the margin within the method to avoid the fallback in the future. However, to do this, the implications on other parts, that use the scala pretty printing, should be thoroughly evaluated and tested. Additionally, besides setting breakgain, the ratio itself, which currently is `margin/20`, could perhaps also be optimized to avoid client-side breaking in edge cases and in very narrow windows. In the `pretty.scala` formatting algorithm, the indentation is set at 2, but later the start of the block is added to the indentation and the sum is rounded up to the next Integer. Since `VSCode_Metric` introduces float widths, that rounding error can increase indentation by almost one character width. In narrow windows the indentation of 2-3 can lead to a lot of space being wasted - especially with deep nesting. A fix for that increase and a dynamic adaption of the `indent` parameter to the margin could improve the legibility further.

List of Figures

List of Tables

Glossary

ASCII American Standard Code for Information Interchange, an older and smaller encoding standard compared to UTF-8, with only a subset of characters. 2, 9, 17

block Pretty-printing unit that groups items and can introduce indentation. 1, 2, 5, 14, 23, 25

break Potential line break whose realization depends on the available margin. 2, 24

character width metric Mapping from characters to measured display widths used for layout decisions. 2, 9–11, 22

Chromium free, open-source web browser project, primarily developed and maintained by Google. 3

codepoint Used in character encodings as key for a symbol defined in the specific encoding standard. 2, 9, 10, 12

GUI Graphical User Interface. That part of software, that the user interacts with by clicking, dragging, or typing. 1–3

IDE Integrated Development Environment, a software that helps developers to write code. 1, 2

indentation Horizontal offset applied to the contents of a block after a line break. 4, 19, 20, 23–25

Isabelle An automated theorem prover. 1–4, 8, 9, 12, 17, 22, 24

Isabelle/HOL The higher order logic part of Isabelle. 1

Isabelle/Scala Scala/JVM layer of Isabelle for system integration and PIDE: bridges the Isabelle/HOL part with tools, builds, (Y)XML/JSON codecs, and IDE front-ends (jEdit, VSCode/LSP). 2, 3, 8, 10, 22

Isar/ML “Intelligible semi-automated reasoning”, the main proof language of Isabelle, as well as the language its core is written in. Based on ML. 1

jEdit an IDE/Texteditor. 1–4, 22–25

JSON JavaScript Object Notation, open standard file format to exchange data in a human-readable way. 11, 22

JSON-RPC JSON Remote Procedure Call, a standard for sending notifications and data to a server, with different modi: with or without expecting a response. 3

LSP Language Server Protocol, a standard to transport information between the front and the backend of an IDE, based on JSON-RPC. 3, 10

margin Maximum line width used by the pretty-printer when deciding line breaks. 1, 2, 4, 6–9, 14–17, 22–25

Node.js Cross-platform, open-source JavaScript runtime environment. It executes JavaScript outside of a browser. 3

PIDE "Prover IDE", the part of Isabelle that orchestrates the information flow between the stateless Isabelle core and the stateful GUI components of the IDE. 1, 2, 4, 8

pretty printing The way, in which a logical text is formatted to make it easy for the user to understand the semantic connections. 3, 4, 6, 8, 22, 24

Unicode A character encoding standard designed to support the use of text in all writing systems. 2, 3, 9

UTF-8 8-Bit Universal Coded Character Set Transformation Format, the most commonly used character encoding for the output of human-readable characters. 9, 22

VSCode Visual Studio Code, an IDE. 1–4, 6, 7, 9, 10, 14, 22–24

XML "Extensible Markup Language", a language to describe hierarchically organized data. 2, 4, 9, 24

References

APPEL, K., und W. HAKEN. 1976. Every planar map is four colorable. *Bulletin of the American Mathematical Society* 82.711–712. URL https://scispace.com/pdf/every-planar-map-is-four-colorable-3sfur3t8fu.pdf.

BERNARDY, JEAN-PHILIPPE. 2017. A pretty but not greedy printer (functional pearl). *Proceedings of the ACM on Programming Languages* 1.1–21. URL http://dx.doi.org/10.1145/3110250.

ELECTRON. 2025. Introduction — electron. https://www.electronjs.org/docs/latest/.

GONTHIER, GEORGES. 2023. A computer-checked proof of the Four Color Theorem. Tech. rep., Inria. URL https://inria.hal.science/hal-04034866.

KORCHMAR, DIANA. 2025. *Webview panels for isabelle: From implementation to unified interfaces*. Bachelor's thesis, Ludwig Maximilian University of Munich.

LINDAE, THOMAS. 2024. *Improving isabelle/vscode: Towards better prover ide integration via language server*. Bachelor's thesis, Technical University of Munich. URL https://www21.in.tum.de/students/past/vscode_plugin_improvements/assets/Isabelle_LSP_Thesis.pdf.

OPPEN, DERECK C. 1980. Prettyprinting. *ACM Trans. Program. Lang. Syst.* 2.465–483.

OVERFLOW, STACK. 2025. Stackoverflow survey 2025 — dev ides. https://survey.stackoverflow.co/2025/technology#1-dev-id-es.

PALUCA, DENIS. 2021. *Isabelle/vscode: Editor improvements and prover ide integrations*. Bachelor's thesis, Technical University of Munich. URL https://www21.in.tum.de/students/past/vscode_plugin_improvements/assets/Isabelle_VSCode_Thesis.pdf.

PAULSON, LAWRENCE C. 1986. Natural deduction as higher-order resolution. *The Journal of Logic Programming* 3.237–258. URL http://dx.doi.org/10.1016/0743-1066(86)90015-4.

PESTOV, SLAVA. 2024. Slava pestov's corner of the web. https://factorcode.org/slava/.

TUM, TECHNICAL UNIVERSITY OF MUNICH. 2024. Overview. https://isabelle.in.tum.de/overview.html.

WENZEL, MAKARIUS. 2014. *Asynchronous user interaction and tool integration in isabelle/pide*, 515–530. Springer International Publishing. URL https://www21.in.tum.de/~wenzelm/papers/itp-pide.pdf.

WENZEL, MAKARIUS. 2017. Isabelle2017 - visual studio marketplace. https://marketplace.visualstudio.com/items?itemName=makarius.Isabelle2017.

WENZEL, MAKARIUS. 2018. Isabelle/pide after 10 years of development. *User Interfaces for Theorem Provers* UITP 2018. URL https://sketis.net/wp-content/uploads/2018/08/isabelle-pide-uitp2018.pdf.

WENZEL, MAKARIUS. 2022a. [isabelle-dev] isabelle/vscode as bundled application. https://isabelle.systems/zulip-archive/stream/247542-Mirror.3A-Isabelle-Development-Mailing-List/topic/.5Bisabelle-dev.5D.20Isabelle.2FVSCode.20as.20bundled.20application.html.

WENZEL, MAKARIUS. 2022b. Todo.md@45ef14a473ad. https://makarius.sketis.net/repos/isabelle_vscode_development/file/tip/TODO.md.

12 Appendix files

in `https://github.com/alp-traum/bachelor_thesis_appendix`

- `Raw_Data/Width_Comparison/Metric_Vergleich` - all calculations and comparisons of the symbol width .csv output
- `Raw_Data/Width_Comparison/vscode_metric-5185206488796536290..csv` - raw output of symbol widths for VSCode_Metric
- `Raw_Data/Width_Comparison/vscode_metric-5950105775932996473..csv` - raw output of symbol widths for Symbol.Metric
- `Raw_Data/Code_Quality.xlsx` - observations of observed code quality and calculations for error score
- `Raw\_Data/Screenshots.pptx` - all used and unused screenshots as slides
- `Raw\_Data/fontsize_effect_on_width_18vs30.tsv` - raw data of the pixelwidths and the relative differences at font size 18 vs font size 30
- `prokoph_final.patch` - the Mercury patch file for the Isabelle repository
- `thesis_signed.pdf` - this thesis, signed