Code Style Sheets

Sam Cohen[1] **and Ravi Chugh**[1]

[1] *The University of Chicago*

Abstract

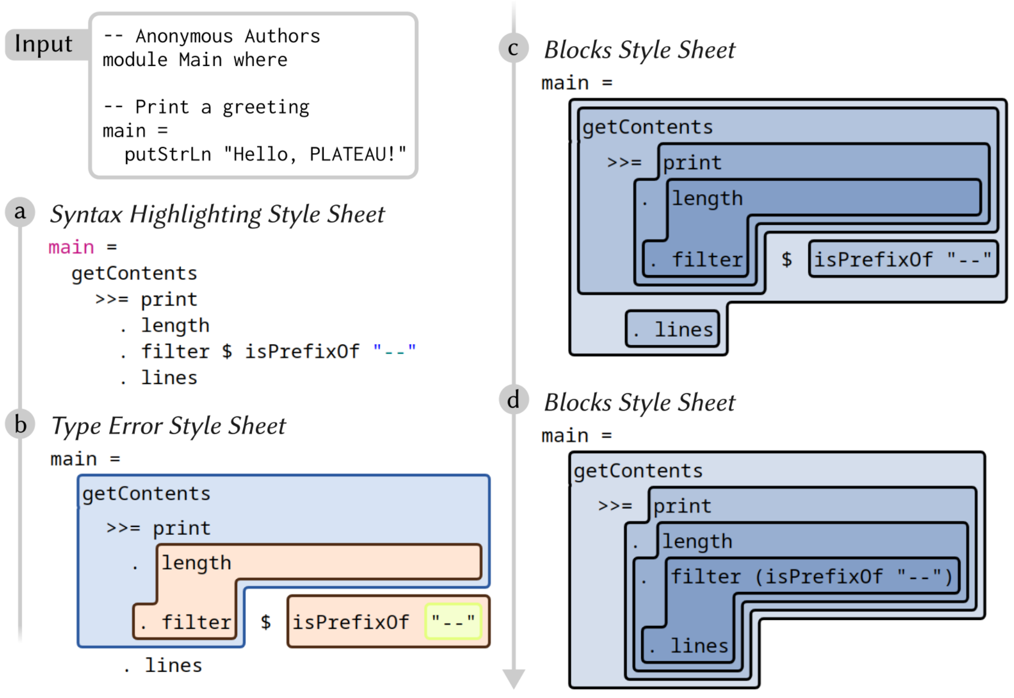
Program text is rendered using impoverished typographic styles. Beyond choice of fonts and syntax-highlighting colors, code editors and related tools utilize very few text decorations. Yet, there is much information that we might want to show alongside program text as it is written, read, or run. Scopes, types, and runtime values, for example, are all relevant to the task of programming, but we lack mechanisms to uniformly present them. In this work, we present a motivating example for a work-in-progress framework, *Code Style Sheets*, and demonstrate how it can be used to visualize multivarious program analyses.*

Keywords: Code Style Sheets, CSS, Text Layout, Program Visualization, Structure Editors.

∗ The text and figures below are exerpted from a draft manuscript [1].

1 A Motivating Scenario

Imagine a programming environment that offered a flexible framework for styling and rendering programs, incorporating both textual and graphical elements (e.g., blocks). Multivarious, configurable kinds of information about programs could be conveyed as users and usage scenarios change. For example, consider a programmer working to debug a "CLOC" program, written in Haskell, that aims to count the number of non-comment lines of code in a given source file.



```
-- Anonymous Authors
module Main where

-- Print a greeting
main =
  putStrLn "Hello, PLATEAU!"
```

```
main =
  getContents
    >>= print
      . length
      . filter $ isPrefixOf "--"
      . lines
```

```
main =
getContents
  >>= print
    . length
    . filter  $  isPrefixOf "--"
  . lines
```

```
main =
getContents
  >>= print
    . length
    . filter  $  isPrefixOf "--"
  . lines
```

```
main =
getContents
  >>= print
    . length
    . filter (isPrefixOf "--")
    . lines
```

Figure 1. Count Lines of Code (CLOC) Program with Type Error and Blocks Style Sheets

PLATEAU
15th Annual Workshop at the Intersection of PL and HCI

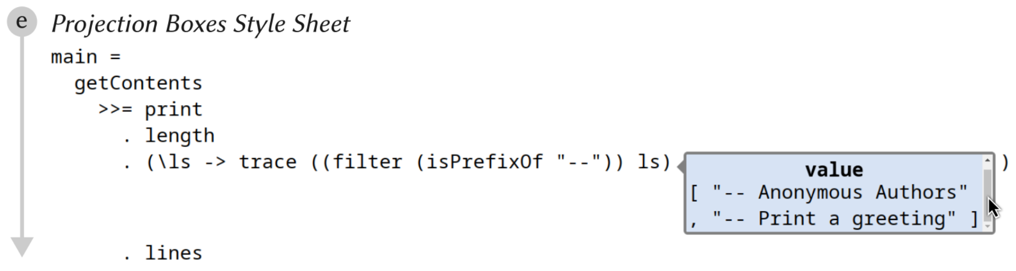
Organizers:
Jon Bell, Sarah Chasins, Elena Glassman, and Joshua Sunshine

1.1 Debugging a Binary Operation Error

Figure 1 (a) shows an initial version of the program, which fails to compile with a type error. In addition to the error message itself, the compiler provides a *code style sheet*, used to decorate the program with a color-coded visualization of the involved types and expressions (Figure 1 (b)). Unsure of why the function is failing, but beginning to suspect something to do with the usage of binary operators, the programmer decides to apply a "blocks style sheet" to help debug (Figure 1 (c)). The programmer sees that the line of code `filter $ isPrefixOf "--"` is not contained within a box, suggesting that maybe the infix function application operator (`$`) has lower precedence than the function composition operator (`.`). So, they add parentheses around the call to `isPrefixOf`. Edits made, the programmer views the blocks style sheet again to confirm that the change worked (Figure 1 d).

1.2 Debugging a Filter Predicate Error

Satisfied that the program now compiles, they run it but observe that the resulting number (two, not shown in Figure 1) is not what they expect (four). The programmer decides to add calls to `trace`—a standard debugging mechanism in Haskell—to inspect the run-time values that flow through the program. The style sheet in Figure 2 e shows a projection boxes–like display [2] of values that flowed through the `trace` expression. Seeing that the two resulting string values are commented (rather than non-commented) lines, the programmer quickly identifies a common (and forgivable) mistake: misremembering the behavior of `filter`, which keeps (rather than discards) values that satisfy the given predicate. To fix the issue, they compose the boolean predicate with `not` to negate its result, and re-run the tests (not shown).


e *Projection Boxes Style Sheet*

```
main =
  getContents
    >>= print
      . length
      . (\ls -> trace ((filter (isPrefixOf "--")) ls)  )


      . lines
```

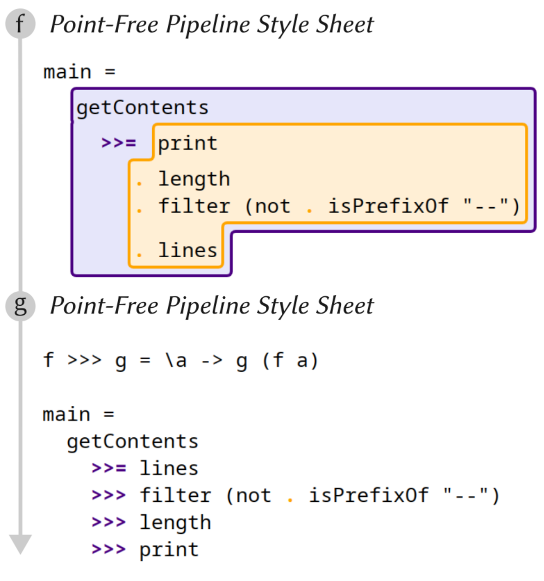
```
               value
[ "-- Anonymous Authors"
, "-- Print a greeting" ]
```

Figure 2. Projection Boxes Style Sheet

1.3 Resolving a Point-Free Pipeline Warning

The program meets its specification, but as depicted in Figure 3 f the linter applies a new style sheet to convey a warning: The program chains together two binary operators, (`>>=`) and (`.`), which propagate data in opposite "directions." Whereas `mx >>= f` denotes (a kind of) left-to-right function application, `f . g` denotes right-to-left function composition. The style sheet chooses a different color for each of the two directions that data flows through the pipeline of operators, highlighting those expressions on either side of a *transition* from left-to-right or right-to-left.

The programmer agrees that maintaining a single direction will be more readable, so they reorder functions using the left-to-right composition operator (`>>>`), resolving the warning (Figure 3 g).


f *Point-Free Pipeline Style Sheet*

```
main =
  getContents
    >>= print
      . length
      . filter (not . isPrefixOf "--")
      . lines
```

g *Point-Free Pipeline Style Sheet*

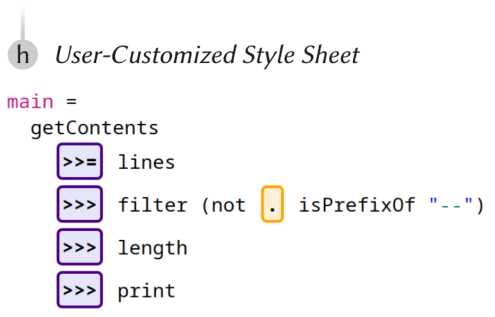
```
f >>> g = \a -> g (f a)

main =
  getContents
    >>= lines
    >>> filter (not . isPrefixOf "--")
    >>> length
    >>> print
```

Figure 3. Point-Free Pipeline Style Sheet

1.4 Updating User Preferences

The programmer appreciates the suggestion about the flow of data through the operator pipeline: When rewriting code for clarity, they lack a reliable heuristic for deciding about when to use left-to-right versus right-to-left function application and composition operators. They believe proactively seeing the directionality information of these binary operators may help when reading and writing other code fragments. So, they wish to repurpose some of the linter's styles to incorporate into their own, personal style sheet for styling by default—in the absence of overriding styles produced by the compiler or editor.

They identify the selector and attributes used by the point-free pipeline style sheet to effect the two-coloring. They copy the rules into their default style sheet, modifying them slightly so that the styles are applied only to the relevant operators rather than the entire binary operation expressions. The resulting styles for the final CLOC program are shown in Figure 4 h.



```
main =
  getContents
    >>= lines
    >>> filter (not . isPrefixOf "--")
    >>> length
    >>> print
```

Figure 4. User-Customized Style Sheet

2 Research Directions

In this section, we describe several research directions that might help us to achieve the vision presented in the motivating example. We have explored some of these areas by building prototypes, while others are left entirely for future work.

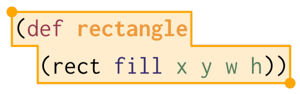
2.1 Layout: Rendering Structured Text with Style

Most of the editors that programmers commonly use to write programs, like VS Code [3] or CodeMirror [4], are what we might describe as "flat" editors because they can only show one layer of structure at a time (often in the form of squiggly underlines or text highlights). Several systems (e.g. [5]–[8]) are *structure* editors—editors for which edits are primarily on the syntax tree of the program, not its textual "view." For these editors, a flat code view is often insufficient because it does not visually surface the objects that the user manipulates. Many editors use *boxes* to represent nested structures because they can be easily composed. However, forcing each term into a box often leads to unatural layouts which don't resemble their corresponding textual view.

In Code Style Sheets, we attempt to bridge the gap between raw unstructured text and boxes by proposing a primitive called *stylish blocks* or *s-blocks* which surface structure while attempting to leave the text layout undisturbed. The shape of these blocks is inspired by the shape of text selections in most graphical text editors. All of the examples shown in the motivating scenario use s-blocks.

S-blocks succeed in producing structured visualizations which resemble their unstructured "flat" counterparts, but they also have some limitations: We might want to find tight-fitting boundaries around expressions, or maintain the vertical alignment of columns, even after some spans of text have been wrapped in outlines. These are important preferences if we want visually-compact, readable, and unobtrusive code visualizations. Yet s-blocks do not have ready solutions to these problems.

We are exploring some other layout primitives that promise to solve the aforementioned issues. One such primitive is the *ragged-block* or *rock*, which allows the boundaries of a text span to be tight-fitting. This allows for more compact text layouts, and more flexibility in determining text alignment and spacing. Rocks, however, produce outlines with many corners, and it is often necessary to simplify them in order to achieve a visually simple layout. There is clearly a wide domain of primitives and algorithms for rendering structured text, each with tradeoffs, and it seems worthwhile to explore this design space more thoroughly.



```
(def rectangle
  (rect fill x y w h))
```

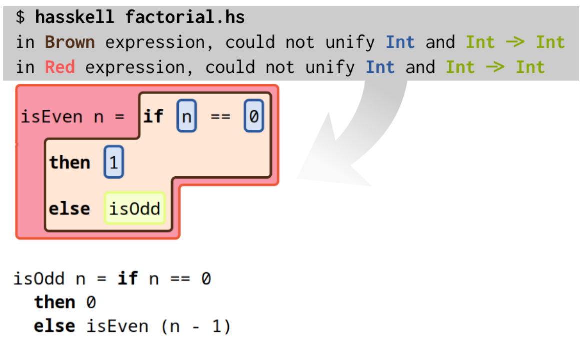
Figure 5. In Deuce [7], text selections are tight-fitting (like rocks), but can't be nested (unlike rocks).

2.2 Applications: Beyond Visualizing Parse Information

The examples above preview mainly syntactic visualizations, but code visualizations are not limited to showing only the parse tree of a program. We might, for example, visualize a program according to its types after a type error has occured.

Figure 6 shows a program with a type error. The visualization is inspired by the error messages produced by Pyret [9], where colors in the error message correspond to highlights in the code. Even in the case of nested type errors (as Figure 6 demonstrates), it is possible to disambiguate between the errors, and show all expressions of interest, independent of their formatting or position in the AST.

Future work could explore whether s-blocks would be a useful visual aid for explaining the provenance of type errors (e.g. [10], [11]).



```
$ hasskell factorial.hs
in Brown expression, could not unify Int and Int -> Int
in Red expression, could not unify Int and Int -> Int
```

```
isEven n = if n == 0
  then 1
  else isOdd
```

```
isOdd n = if n == 0
  then 0
  else isEven (n - 1)
```

Figure 6. Type Error Style Sheet

2.3 A Style Sheet Language for Code

Our motivating scenario presented a workflow which involved using different visualizations for different tasks. But, how are these visualizations specified? In the design of a system for styling code, it seems necessary to give some consideration to the description of styles as well as the styles themselves.

Just as with the choice of layout primitive, there are some tradeoffs to consider when designing a style description language (or *style sheet*). On one hand, we could describe style sheets as arbitrary programs which take as input an AST, and return as output an HTML document (or image, or any other visual representation we can imagine). This approach offers the utmost expressivness and flexibility, but writing style sheets as arbitrary programs has some disadvantages. If we have no uniform interface for describing *what* can be selected and *how* it can be styled, then we have no hope of composing style sheets. That is, to compose style sheets, the style sheets must agree on which elements in the AST can be selected, and once we have a selected element, how to denote that styles have been applied.

In Code Style Sheets, we use a language inspired by Cascading Style Sheets (CSS [12]) to style programs. Instead of selecting HTML elements by tag or class, our style language selects values in an AST with pattern matching. Importantly, the language of selectors operates over the *original program*, not its view. This means that rules from multiple style sheets may be freely composed, since style sheet rules designed for one view will work with another.

2.4 Code Style Sheets in the Editor

So far we've shown examples of static ("read-only") code displays, but one of the most attractive applications of structured text displays are interactive editors. Structure editors, in particular, often expose interactions that operate not over the program text, but over terms in the program. However, some systems, like Tylr [13] and Sandblocks [5], go to great lengths to make text edits feel intuitive, while other systems, such as Scratch [14], do away with text edits entirely, or switch between text and structure edits, but don't allow them simultaneously (e.g. [15], [16]).

S-blocks or rocks could be opportune user interface frameworks for designing these kinds of editors since they can visualize structure, but are invariant to the formatting of the underlying text. This could help make both structural and text edits feel intuitive.

References

[1] S. Cohen and R. Chugh, *Code Style Sheets: CSS for Code*, 2024.

[2] S. Lerner, "Projection Boxes: On-the-fly Reconfigurable Visualization for Live Programming," in *Conference on Human Factors in Computing Systems (CHI)*, 2020. DOI: 10.1145/3313831.3376494. [Online]. Available: https://doi.org/10.1145/3313831.3376494.

[3] Microsoft, *Visual Studio Code*, 2024. [Online]. Available: https://code.visualstudio.com/.

[4] M. Haverbeke, *CodeMirror: Extensible Code Editor*, 2024. [Online]. Available: https://codemirror.net/.

[5] T. Beckmann, P. Rein, S. Ramson, J. Bergsiek, and R. Hirschfeld, "Structured Editing for All: Deriving Usable Structured Editors from Grammars," 2023. DOI: 10.1145/3544548.3580785. [Online]. Available: https://doi.org/10.1145/3544548.3580785.

[6] C. Omar, I. Voysey, M. Hilton, J. Aldrich, and M. A. Hammer, "Hazelnut: A Bidirectionally Typed Structure Editor Calculus," in *Symposium on Principles of Programming Languages (POPL)*, 2017. DOI: 10.1145/3009837.3009900. [Online]. Available: https://doi.org/10.1145/3009837.3009900.

[7] B. Hempel, J. Lubin, G. Lu, and R. Chugh, "Deuce: A Lightweight User Interface for Structured Editing," in *International Conference on Software Engineering (ICSE)*, 2018. DOI: 10.1145/3180155.3180165. [Online]. Available: https://doi.org/10.1145/3180155.3180165.

[8] JetBrains, *MPS (Meta Programming System)*, 2011–2024. [Online]. Available: https://en.wikipedia.org/wiki/JetBrains_MPS.

[9] Brown PLT, *Picking Colors for Pyret Error Messages*, 2018. [Online]. Available: https://blog.brownplt.org/2018/06/11/philogenic-colors.html.

[10] I. Bhanuka, L. Parreaux, D. Binder, and J. I. Brachthäuser, "Getting into the Flow: Towards Better Type Error Messages for Constraint-Based Type Inference," *Proceedings of the ACM on Programming Languages (PACMPL)*, no. OOPSLA, 2023. DOI: 10.1145/3622812. [Online]. Available: https://doi.org/10.1145/3622812.

[11] E. Zhao, R. Maroof, A. Dukkipati, A. Blinn, Z. Pan, and C. Omar, "Total Type Error Localization and Recovery with Holes," *Proceedings of the ACM on Programming Languages (PACMPL)*, no. POPL, 2024. DOI: 10.1145/3632910. [Online]. Available: https://doi.org/10.1145/3632910.

[12] World Wide Web Consortium (W3C), *Cascading Style Sheets (CSS) 3*, 2024. [Online]. Available: https://www.w3.org/Style/CSS/.

[13] D. Moon, A. Blinn, and C. Omar, "Gradual Structure Editing with Obligations," in *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2023. DOI: 10.1109/VL-HCC57772.2023.00016. [Online]. Available: https://doi.org/10.1109/VL-HCC57772.2023.00016.

[14] M. Resnick, J. Maloney, A. Monroy-Hernández, *et al.*, "Scratch: Programming for All," *Communications of the ACM (CACM)*, 2009. DOI: 10.1145/1592761.1592779. [Online]. Available: https://doi.org/10.1145/1592761.1592779.

[15] T. Ball, A. Chatra, P. de Halleux, S. Hodges, M. Moskal, and J. Russell, "Microsoft MakeCode: Embedded Programming for Education, in Blocks and TypeScript," in *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E*, 2019. DOI: 10.1145/3358711.3361630. [Online]. Available: https://doi.org/10.1145/3358711.3361630.

[16] D. Bau, D. A. Bau, M. Dawson, and C. S. Pickens, "Pencil Code: Block Code for a Text World," in *International Conference on Interaction Design and Children (IDC)*, 2015. DOI: 10.1145/2771839.2771875. [Online]. Available: https://doi.org/10.1145/2771839.2771875.