Retrofitting Text Editors with Ragged Blocks

Sam Cohen
University of Chicago
Chicago, IL, USA
samcohen@uchicago.edu

Jackson Slipock
University of Chicago
Chicago, IL, USA
jacksons@uchicago.edu

Ravi Chugh
University of Chicago
Chicago, IL, USA
rchugh@cs.uchicago.edu

***Abstract*—Although many compelling graphical code-editing interactions have been explored, practical programming environments remain almost exclusively text-based. In this paper, we offer a design for extending a conventional text-based code editor with graphical interactions. The key innovation from prior work that drives our design is a layout primitive called *ragged blocks*, which allow heavily nested spans of text—like code—to be decorated with visual styles while minimizing the difference in appearance compared to a plain-text layout. Whereas the prior techniques support only static, read-only code visualizations, in this work we develop extensions to the ragged-text layout algorithm which can serve as the basis for an interactive code editor that mixes text- and graphical-editing. We implement our design in a prototype code editor called Jeuce, which is configured to emulate, and extend, multiple GUI interactions from existing block-based and structure editors within our text-based design.**

***Index Terms*—blocks, ragged blocks, IDEs, CodeMirror.**

I. Introduction

> Blocks editors are inferior environments for writing code because they lack the full range of tools and tooling provided by text-based IDEs. We either need blocks editors embedded in IDEs or fully-realized block-based IDEs.
>
> —Ko (2020), "What's wrong with blocks..." [17]

Decades of research and development have led to compelling user-interface and system designs for *structure editing*, yet text editors continue to reign supreme. We take on the perennial question: ***Can existing text editors be enhanced to support direct-manipulation structure-editing?*** Embracing the viewpoint above that we "need blocks editors embedded in IDEs", we are interested in two influential structure-editing paradigms—one traditionally limited to block-based editors and the other limited to text-based IDEs.

Block editors, such as Scratch [31] and Snap [13], support ***drag-and-drop syntax editing***, in which the user clicks and drags syntactic elements into "holes" in program expressions or "gaps" between program statements. Although used pervasively in classroom settings with young learners, sadly, drag-and-drop syntax editing has not been successfully integrated into an editor that also freely supports text editing. Considering the gulf between existing text-based and block-based editors, it comes as no surprise that some students perceive the latter as being inauthentic and less powerful [32].

Another influential structure-editing paradigm is ***menu-based refactoring***, commonly found in text-based IDEs such as Eclipse [11] and VS Code [23]. In this workflow, the user chooses a transformation from a set of named options and applies it to select program fragments. Unfortunately, menu-based refactoring is limited by its reliance on *text*-selection to identify the *structural* abstract-syntax tree (AST) elements to be transformed—this is one significant barrier to even wider adoption of refactoring tools [26].

We present the design and implementation of a programming interface called Jeuce, which integrates three paradigms: (a) text editing, (b) drag-and-drop syntax editing, and (c) a menu-based refactoring workflow that centers *structural*, rather than textual, selection.

In order to visualize program structure in a text editor, we adopt ***ragged blocks*** [7], a layout primitive that allows nested code fragments to be rendered with visual decorations, including borders and padding, while minimizing the difference in appearance compared to the underlying plain text. The first contribution of our work is a variation of prior layout algorithms [6], [7], which now allows ragged-block program visualizations to be embedded within an existing, interactive text editor (namely, CodeMirror [14]), thus inheriting text editing and related conveniences "for free."

Thus equipped with a substrate supporting both conventional text-editing as well as (ragged-)block-based visualizations, we implement several block-based structure-editing interactions in Jeuce. Compared to disparate, state-of-the-art tools that support *either* drag-and-drop syntax editing (e.g., Scratch, Snap, and DNDRefactoring [21]) *or* block-and-menu based refactoring (i.e., Deuce [15]), in Jeuce these two types of structure-editing interactions are smoothly integrated. Furthermore, the combination of text- and block-based editing also enables some novel interaction opportunities.

In the rest of this short paper, we demonstrate the GUI interactions in Jeuce through an example (§ II), followed by a discussion of design and implementation details (§ III), related work, (§ IV) and future work (§ V).

II. GUI Interactions in Jeuce

The following example demonstrates an imagined programming workflow in Jeuce. It is also available in video form [9], which may give the reader a more complete understanding of the example. Our goal is to generate an image of confetti, whose position and rotation are randomized to give the impression that they have been thrown in the air and have landed

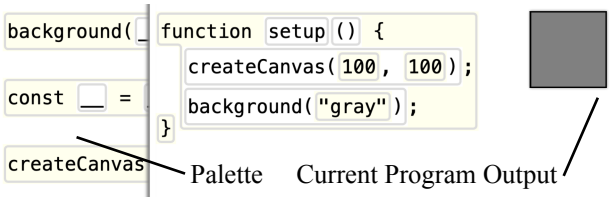

Fig. 1. Initial program.

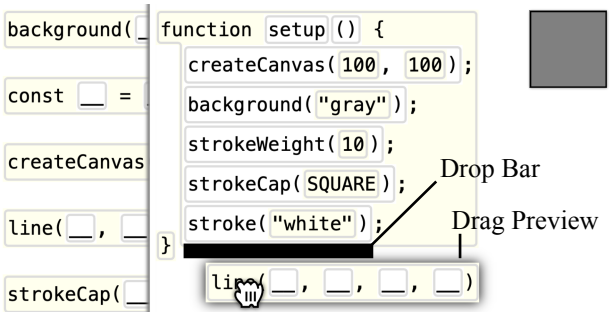

Fig. 2. Dragging `line` into the program.

randomly on a surface. The example uses p5 [22], a JavaScript library for making art and interactive visualizations. Figure 1 shows a typical initial program in p5. Unlike a traditional text editor, in Jeuce, most terms in the program are surrounded with a border and padding, allowing them to be selected by the cursor, and explicitly differentiating them from their subterms.

The programmer's immediate goal is to draw a line to represent a single piece of confetti, or one *confetto*. In Jeuce, like in Scratch or Snap, common functions are kept in the *palette* which is an area to the left of the editor from which program fragments can be picked. Looking through the palette, they find the `line` function and drag it into the editing area. Here, the user selects a term by clicking along its boundary, and specifies where it will be inserted by dragging it over a statement. If the hovered statement is a valid drop target, a black "drop bar" indicates where the dropped term will go (see Figure 2). The user introduces some variable declarations, also by dragging, as shown in Figure 3.

Drag-and-drop gestures are well suited to the task of inserting statements into a program, but they sometimes break down when dealing with *expressions*. Users are accustomed to typing mathematical expressions using their keyboard (when

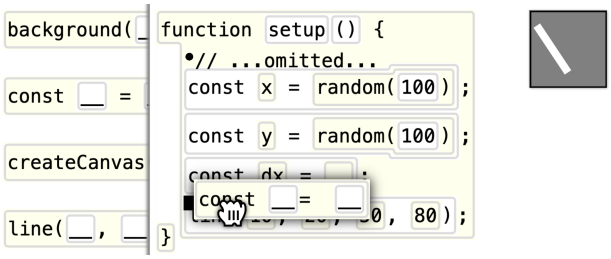

Fig. 3. Dragging a variable declaration into the program.

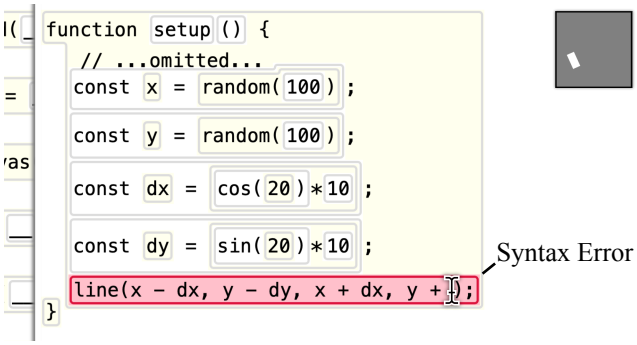

Fig. 4. Filling in an expression by typing.

using, e.g., a calculator), so when adjusting the arguments to the `line` statement, or filling in the right-hand side of the variable declarations, the programmer decides to type directly into the editing area. One key property of Jeuce is that the user can choose to begin typing at any time, and do so at any position in the document, and the visualization of the ragged blocks is maintained throughout.

Of course, if the user can type arbitrary text into the editor, they can also break the syntactic properties of the program which are the basis for the ragged-blocks visualizations. To combat this, the visualization attempts to *localize* the extent of syntax errors so that most of the structure can continue to be rendered despite momentary violations of the syntax. Figure 4 shows the user midway through an edit to an argument of `line`, showing how the syntax error, visualized in red, is local to the area of the program being edited.

Visualizing terms in the program also enables selections on those terms, and with those selections we can invoke program transformations, which can cause larger changes to the program than the local edits available through drag-and-drop gestures. Figure 5 shows a transformation which, given one or more statements, and zero or more *variables* within those statements, produces a function definition whose body is the selected statements, and whose parameters are the selected variables. The programmer uses this transform to abstract the drawing code for a single piece of confetti into the new `confetto` function. In Figure 6, they refine this definition by selecting several constants in the body of `confetto`, before turning them into function arguments.

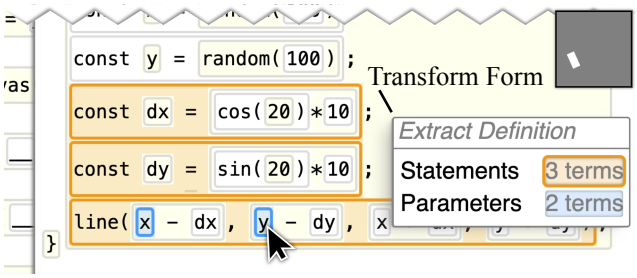

Fig. 5. Extracting a function definition.

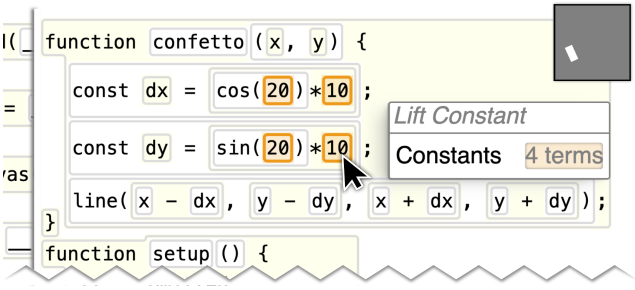

Fig. 6. Lifting constants into parameters.

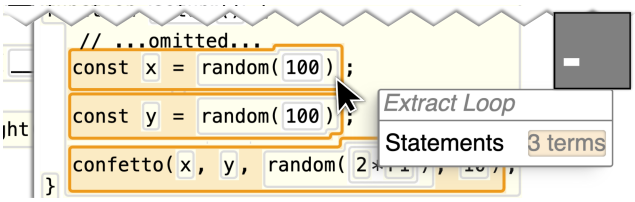

Fig. 7. Extracting a loop.

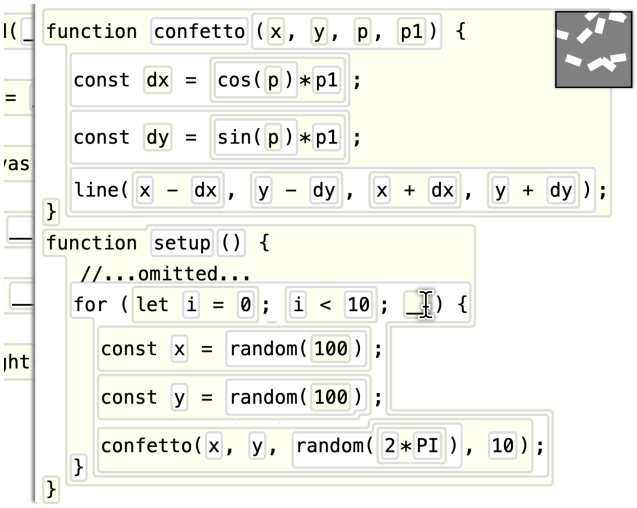

Fig. 8. The finished program. Confetti!

Having drawn one piece of confetti, the programmer now wants to draw multiple. In Figure 7, they use another program transformation to introduce a loop around the selected statements, and fill in the loop parameters using text edits, yielding the final program shown in Figure 8.

III. Design and Implementation

Next, we describe in more detail the editing paradigms in Jeuce, namely, (a) text editing, (b) drag-and-drop syntax editing, and (c) menu-based refactoring, followed by implementation details regarding (d) the layout algorithm and (e) AST transformations. The implementation itself uses 3.8k lines of TypeScript, and is available online (https://github.com/sbcohen2000/jeuce).

A. Basic Text Editing with Ragged Blocks

Though Jeuce inherits most of its text-editing functionality from CodeMirror, there are some technical considerations in making text edits operate smoothly while also rendering ragged blocks. This structured visualization takes as input a *layout tree* [6], [7], whose leaves are strings describing the document's textual content, and whose nodes describe the styles (such as fill, border, and padding) which are used to visually demarcate the subtree. So that the visualization does not diverge from the rendered text, it is important that this layout tree perfectly captures the text of the buffer. Stated more precisely, if the layout tree were flattened into a list of leaves, then concatenating the text of each leaf should yield the exact same string as that displayed in the code buffer.

It is not difficult to construct a layout tree satisfying this property (a single node with a single child, containing the entire text of the document would suffice), but it can be difficult to also meaningfully reflect the abstract syntax of the program being edited, especially in the presence of syntax errors, which occur very often in text editing. To aid this task, we have employed Tree-sitter [5], an error-tolerant parser, to find the structure expressed in the layout tree. Internally, we convert syntax trees from Tree-sitter into our own datatype (representing a small subset of JavaScript) which is used for rendering and for program transformations. Each node in this tree records its own concrete syntax so that the program's text can be reconstructed from its syntax tree.

When adding additional statements into the program via GUI actions, it is necessary to generate a little bit of "glue" syntax, including newlines, indentation, and semicolons. Jeuce implements a few heuristics to determine the correct syntax to insert for a given node. But because there is no restriction on the formatting conventions the programmer uses when constructing programs in Jeuce, these heuristics can fail to produce programs which match the surrounding "code style." Improved heuristics or a more principled way to construct new syntax would be an interesting avenue for future work.

B. Drag-and-Drop Syntax Editing

Jeuce's drag-and-drop interaction is largely inspired by existing block-based editors, such as Snap [13], Scratch [31], Blockly [12], and MakeCode [2]. In these editors, programs are constructed by dragging terms from a palette into an editing area. In designing Jeuce's drag-and-drop interaction, we most closely follow the visual conventions of Snap, which are to superimpose the "held" term underneath the mouse during a drag and to preview the location that a statement will be dropped with a horizontal bar. Figure 9 compares the user interfaces during a drag-and-drop interaction between Snap and Jeuce. The primary difference is that in Jeuce, program fragments are restricted to exist strictly *above* or *below* one another in the document (as in a traditional text editor), rather than also *beside* as in most block-based editors (which typically support arbitrary positioning of blocks in a two-dimensional editing area).

Though drag-and-drop gestures are the main way users build programs in Snap, they are not the only way to manipulate

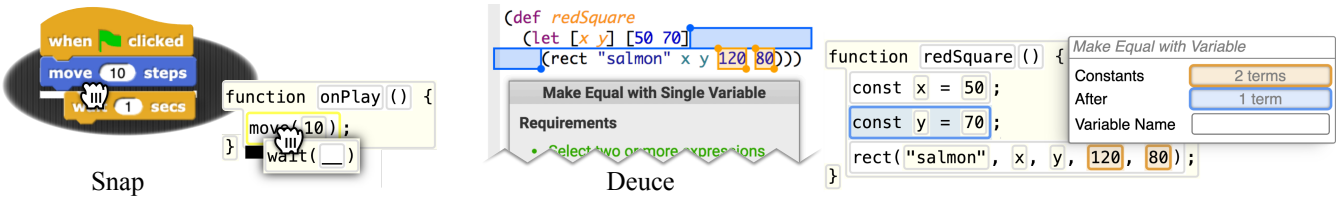

Fig. 9. Drag-and-drop interactions in Snap and Jeuce (left), and menu-based refactorings in Deuce and Jeuce (right).

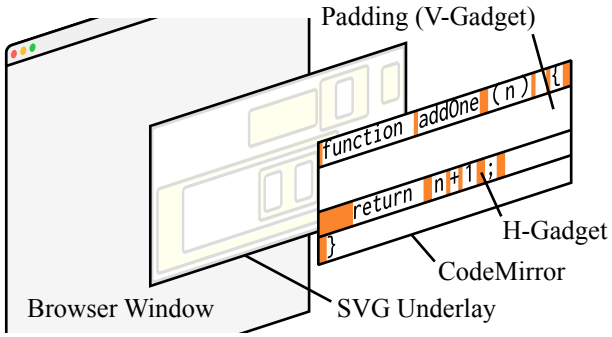

Fig. 10. Exploded view of the Jeuce user interface.

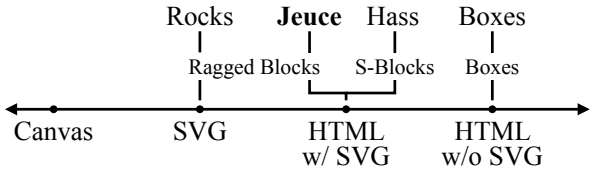

Fig. 11. Web-based options for rendering program text as blocks.

terms. For example, single-operand math functions, such as `sqrt` or `sin`, exist under the same block, and are accessible through a drop-down menu, preempting a gesture which would otherwise be required to swap out one block for another. Jeuce does not implement these menus, but they are potentially less necessary in an environment where text edits have a low cost. However, they are not incompatible with Jeuce, and adding this functionality could help prevent an additional class of errors.

C. Menu-Based Refactoring

Users can build up programs in Jeuce using text edits or drag-and-drop gestures, but they can also manipulate programs through block-based selections and menu-based program transformations. Figure 9 shows the transform specification interface in Jeuce. Jeuce's implementation of this idea is mainly inspired by Deuce [15], but there are a few differences, the most fundamental of which is that ragged blocks in Jeuce can be arbitrarily nested. In addition to offering more opportunities for visualization, this capability provides new interactions involving multiple (possibly nested) selections (e.g., Figure 5).

D. Translating Ragged Blocks to HTML + SVG

The Jeuce interface uses a structured text visualization to surface manipulable terms in the program. The most common approach to rendering structure atop text is to use boxes—HTML's native mechanism for rendering elements—but this technique, though simple and computationally efficient, tends to produce text layouts that are too dissimilar to their plain-text counterparts (see, for example, Sandblocks [4, Fig. 12]; cf. Figure 8). This consideration is especially important in Jeuce, where we expect the user to be able to perform text edits at any position in the source text. If the visualization of the program as a string of characters (the model of text edits) is too different from the visualization as a tree (the model for structural edits), the text edits won't be intuitive.

Yet, it is essential that Jeuce interact with the page's HTML since the text editor's user interface is represented by HTML elements. The layout algorithm, which determines the position of the fragments of text on the page, must coordinate with the text editor so that the program text doesn't fall out of correspondence with its structure. In prior work we described Hass [6], a system for structured text visualization which places spacers (called *horizontal* or *vertical gadgets*) between certain fragments of text in the program in order to make space for borders and padding which may exist around terms in the program. Like Jeuce, these borders, which are in general not rectangular, are rendered underneath the text using an SVG. Our approach in Hass of inserting gadgets between fragments as a means to position them is advantageous for Jeuce because text editors often provide a mechanism to place decorations in between arbitrary characters of the document, or between lines. Therefore, we can position the program text as required to visualize structure, but do so in a way which is supported by the text editor. Our layout algorithm must then determine the position and size of these gadgets, passing them to the text editor so that the accompanying underlay matches up.

Figure 10 visualizes the main components of our implementation. CodeMirror [14] serves as Jeuce's text editor, with an SVG placed underneath in order to visualize the program's structure. The structural and textual views are synchronized using horizontal and vertical gadgets (h- and v-gadgets, respectively), examples of which are indicated in Figure 10.

Hass introduces a useful mechanism to synchronize the SVG underlay with the program text, but the algorithm it uses to position text and find the outlines of terms produces layouts which are unnecessarily tall [7]. In order to improve upon this deficiency, in subsequent work we developed Rocks [7], offering a layout algorithm which produces more compact layouts, albeit with more complex outlines. In Jeuce, we hypothesize that these more complex layouts will be important

in editing programs with lots of nesting, and so we would like to take advantage of the improved algorithm described in Rocks. However, Rocks is formulated in a way which does not use gadgets to separate the fragments of text, instead returning the absolute position of each fragment. This is not amenable to integration with existing text editors. Our Jeuce prototype uses an adaptation of the Rocks algorithm which re-inserts these gadgets, the core semantics of which are described in a supplementary appendix [8]. Figure 11 summarizes the landscape of available rendering algorithms, primitives, and rendering targets. As mentioned, we choose to target HTML since it is compatible with web-based text editors such as CodeMirror, but use an SVG to render borders since they are sometimes non-rectangular. Of the prior work, only Hass supports this target, and it does so using gadgets. To produce more compact layouts, we adapted ragged blocks to emit gadgets, permitting this algorithm to interact with HTML.

E. AST Transformations

Users can specify a transform to apply by right-clicking on the editing area and choosing from the list of available transforms. Once a transform is selected, the user produces selections to fill in the arguments to the transform. These arguments are, in the common case, structural selections constructed by clicking terms in the user interface, as demonstrated in the example of § II.[1] It is also possible to specify a string argument, useful when a transform calls for a new name to be introduced into the program.

Transforms in Jeuce are represented by arbitrary AST-manipulating functions. Each function is associated with some metadata describing the kinds of arguments it expects, which permits the form interface to be generated. Each argument can also define a validator function, which takes as arguments the transform arguments provided so far, and can produce an error message describing why the transform argument is invalid, as the case may be. Support for validators goes beyond the API exposed by Deuce for defining transforms.

IV. Related Work

Jeuce permits the user to decide the concrete syntax—including formatting of whitespace—of any term in the program. In contrast, prior approaches have restricted syntax to some degree, but promise fewer or better-controlled errors in return. Tylr [24], [25] permits text edits that temporarily violate the grammar by tracking what textual edits are needed to restore well-formedness. Another approach is to completely restrict concrete syntax, but design keyboard interactions which remain intuitive, even though they don't correspond to textual edits, which is the approach taken in Sandblocks [3], [4]. In both of these systems, and many others, edits are expressed directly in terms of rewrites to the program's syntax tree. Jeuce explores an alternative where arbitrary text edits are possible because the concrete syntax tree (which may contain errors) is continually reconstructed from the program text.

Regarding structure editing, we have focused on two interaction paradigms that are fundamental to combining text- and block-based editing, but there are other relevant designs to consider. Several systems, including Barista [18] and Hazel [27], among others [1], [10], [16], [29], provide *type-specific GUIs* (variously named) for editing program elements, often literal values but sometimes more general syntactic forms.

Text and structure edits can be combined more coarsely as well. Greenfoot [19], [20] provides *frame-based* editing, where blocks are used for statements, but not for expressions. Compared to Jeuce, the visual structure is simplified because in statement-based languages statements almost always occupy a proper rectangle (as opposed to an "improper" ragged block). In Greenfoot, text-editing is obligatory for expressions, which are the most tedious to edit using drag-and-drop operations.

Whereas our focus in this work is on user interface concerns, Hazel [28], [33] and Pantograph [30] make progress towards defining semantic foundations for structure editing. Techniques such as these could make it easier to extend Jeuce with new transformations, and help ensure that they operate and compose in sensible ways.

V. Conclusion and Future Work

We have presented a design methodology for retrofitting existing text editors with a variety of block-based structure-editing interactions, and implemented a proof-of-concept system called Jeuce that demonstrates its promise.

There are many fruitful directions for future work. One is for Jeuce to support a larger subset of JavaScript, as well as additional languages. Another is to instantiate the design in other existing, full-featured text-based IDEs, such as VS Code. In addition to these engineering concerns, there are several fundamental design questions to investigate.

As mentioned, several systems have further investigated ways in which the editor can preserve well-formedness of the syntax tree across arbitrary text edits. Such techniques, if applied to Jeuce, could help prevent more classes of errors.

More compact layouts could also prove opportune in implementing type-specific GUIs. These type-specific GUIs could also appear in the syntax palette, which could be dragged into the program, in contrast to typing, say, `$slider` to introduce a numeric slider widget in Hazel.

Ultimately, this line of work aims to enable the engineering of heterogeneous [10] editors for practical use. We are particularly interested to incorporate Jeuce into a classroom setting, such that teachers and students can configure the editor on a spectrum from "blocks only" to "blocks plus text" as appropriate, obviating the need for specialized languages which are amenable to blocks representations, and easing the transition to mainstream languages. We are eager to accompany such a deployment with a user study to understand how students use Jeuce in practice. Our work provides a design method for building these kinds of editors based on mature implementations of existing tools and algorithms.

[1]The task of selecting terms is slightly complicated by the fact that a text editor *also* generates selections from clicks. So, Jeuce classifies clicks into structural selections, which occur *inside* a ragged block but *outside* of a text fragment, or text selections, which occur strictly *inside* text fragments.

Acknowledgments

This work was supported in part by the University of Chicago College Innovation Fund.

References

[1] L. Andersen, M. Ballantyne, and M. Felleisen. Adding Interactive Visual Syntax to Textual Code. *Object Oriented Programming Systems Languages & Applications OOPSLA*, 2020.

[2] 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.

[3] T. Beckmann, L. Böhme, M. Taeumel, and R. Hirschfeld. Block-Based Editing in a Textual World. In *Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments (PAINT)*, 2025.

[4] T. Beckmann, P. Rein, S. Ramson, J. Bergsiek, and R. Hirschfeld. Structured Editing for All: Deriving Usable Structured Editors from Grammars. In *Conference on Human Factors in Computing Systems (CHI)*, 2023.

[5] M. Brunsfeld et al. Tree-sitter, 2013–2024. https://tree-sitter.github.io/tree-sitter/.

[6] S. Cohen and R. Chugh. Code Style Sheets: CSS for Code. *Proceedings of the ACM on Programming Languages (PACMPL), Issue OOPSLA1*, 2025.

[7] S. Cohen and R. Chugh. Ragged Blocks: Rendering Structured Text with Style. In *Symposium on User Interface Software and Technology (UIST)*, 2025.

[8] S. Cohen, J. Slipock, and R. Chugh. Retrofitting Text Editors with Ragged Blocks: Supplementary Appendices. 2026. To appear on https://arxiv.org/.

[9] S. Cohen, J. Slipock, and R. Chugh. Retrofitting Text Editors with Ragged Blocks: Video Figure, 2026. https://doi.org/10.5281/zenodo.21515436.

[10] M. Erwig and B. Meyer. Heterogeneous Visual Languages: Integrating Visual and Textual Programming. *Symposium on Visual Languages (VL)*, 1995.

[11] E. Foundation. Eclipse, 2001–2026. https://eclipseide.org/.

[12] Google. Blockly, 2013–2024. https://developers.google.com/blockly.

[13] B. Harvey, D. D. Garcia, T. Barnes, N. Titterton, D. Armendariz, L. Segars, E. Lemon, S. Morris, and J. Paley. Snap! (Build Your Own Blocks). In *Technical Symposium on Computer Science Education (SIGCSE TS)*, 2013.

[14] M. Haverbeke. CodeMirror: Extensible Code Editor, 2007–2026. https://codemirror.net/.

[15] 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.

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

[17] A. Ko. What's Wrong with Blocks Languages According to Ben Shapiro, 2020. https://medium.com/bits-and-behavior/whats-wrong-with-blocks-languages-according-to-ben-shapiro-7a74dd77b43b.

[18] A. J. Ko and B. A. Myers. Barista: An Implementation Framework for Enabling New Tools, Interaction Techniques and Views in Code Editors. In *Conference on Human Factors in Computing Systems (CHI)*, 2006.

[19] M. Kölling. Greenfoot: A Highly Graphical IDE for Learning Object-Oriented Programming. In *Conference on Innovation and Technology in Computer Science Education (ITiCSE)*, 2008.

[20] M. Kölling, N. C. C. Brown, and A. Altadmri. Frame-Based Editing: Easing the Transition from Blocks to Text-Based Programming. In *Workshop in Primary and Secondary Computing Education (WiPSCE)*, 2015.

[21] Y. Y. Lee, N. Chen, and R. E. Johnson. Drag-and-Drop Refactoring: Intuitive and Efficient Program Transformation. In *International Conference on Software Engineering (ICSE)*, 2013.

[22] L. McCarthy et al. p5.js, 2013–2026. https://p5js.org.

[23] Microsoft. Visual Studio Code, 2015–2026. https://code.visualstudio.com/.

[24] D. Moon, A. Blinn, and C. Omar. Gradual Structure Editing with Obligations. In *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2023.

[25] D. Moon, A. Blinn, T. J. Porter, and C. Omar. Syntactic Completions with Material Obligations. *Proceedings of the ACM on Programming Languages (PACMPL), Issue OOPSLA2*, 2025.

[26] E. Murphy-Hill and A. P. Black. Breaking the Barriers to Successful Refactoring: Observations and Tools For Extract Method. In *International Conference on Software Engineering (ICSE)*, 2008.

[27] C. Omar, D. Moon, A. Blinn, I. Voysey, N. Collins, and R. Chugh. Filling Typed Holes with Live GUIs. In *Conference on Programming Language Design and Implementation (PLDI)*, 2021.

[28] 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.

[29] C. Omar, Y. Yoon, T. D. LaToza, and B. A. Myers. Active Code Completion. In *International Conference on Software Engineering (ICSE)*, 2012.

[30] J. Prinz, H. Blanchette, and L. Lampropoulos. Pantograph: A Fluid and Typed Structure Editor. *Proceedings of the ACM on Programming Languages (PACMPL), Issue POPL*, 2025.

[31] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, et al. Scratch: Programming for All. *Communications of the ACM (CACM)*, 2009.

[32] D. Weintrop and U. Wilensky. To Block or Not to Block, That is the Question: Students' Perceptions of Blocks-Based Programming. In *International Conference on Interaction Design and Children (IDC)*, 2015.

[33] 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), Issue POPL*, 2024.

APPENDIX A
ALGORITHM DESCRIPTION

In § III, we informally described changes to the layout algorithm proposed in prior work [7]—specifically, algorithm $\mathrm{L1_P}$—which accomplish our goal of rendering ragged blocks underneath an HTML-based text editor. In this appendix, we make these changes more precise, describing a minimal version of the algorithm implemented in our Jeuce prototype, using the same language as $\mathrm{L1_P}$ whenever possible. The primary difference is that we make the visual separation between fragments concrete through the insertion of h- and v-gadgets. Algorithm $\mathrm{L1_P}$ is not immediately suited to this task because, though it finds the position of the fragments in the document, it also forgets the document structure in doing so, making it difficult to associate the space between each fragment with an h-gadget, and with each line a v-gadget.[2]

We solve this problem by retaining some essential document structure (the location of newlines) throughout. In $\mathrm{L1_P}$, a function called layout is used to process a layout tree into a list of regions, each one representing a line. These lines are then merged into a single *region* using the merge function. This final merge is problematic because it forgets the extent of the input lines. In the algorithm below, this final merge is implemented by positionLines, whose output is roughly a nested list, rather than a single list (i.e., a *region*). The output of positionLines is only *roughly* a list because it must produce not only the positioned lines, but the v-gadgets between them. To represent fragments or lines separated by gadgets, we define a gadget-list (*gList*), which is a list where each element is separated by a gadget, in our case represented by a number *n*.

The structure of this algorithm is thus slightly different from $\mathrm{L1_P}$; rather than producing a list of lines through a single pass of the layout tree, then finding each line's vertical position, the algorithm here performs one additional pass. We first flatten the layout tree into a list of lines, then find the horizontal position of each fragment in each line, then find the vertical position of each line. (This alternate approach is discussed briefly in the prior work, since retaining line-by-line structure is a prerequisite for managing constraints between fragments on different lines [7, § A], or adding *glue* between fragments [7, § B]. Incidentally, these changes are only sketched in the prior work, whereas here we give a complete description of the "structure preserving" variant of $\mathrm{L1_P}$.)

In light of these additional passes, we introduce the following notation:

x, x_i, ...	Fragments with a known size
$\dot{x}$, $\dot{x}_i$, ...	Fragments with a known size and horizontal position
$\ddot{x}$, $\ddot{x}_i$, ...	Fragments with a known size, horizontal position, and vertical position
n, n_i, ...	Integers
α	Type variables
	Additional names (such as *rest* or *line*) are used when they would add clarity

We record in the names and type signatures the information we know about each fragment at each stage in the algorithm. For example, after flattening the layout tree, we know how many lines the document will have, and we know the fragments on each line, but we do not know the fragments' horizontal or vertical positions, only their sizes, and how they should be wrapped up in layers of padding (encoded by the *stack*). Hence, the type signature of flatten is:

$$\text{flatten } t = [[\mathit{stack}\langle x\rangle]]$$

The final layout—which comprises a double-nested list of fragments (whose position we now know), the space between fragments, and lines adorned with gadgets—is then represented by a $\mathit{gList}\langle\mathit{gList}\langle\mathit{stack}\langle\ddot{x}\rangle\rangle\rangle$. Given a layout tree, the final layout can be found using:

$$\text{positionLines} \circ \mathsf{map}\ (\text{positionLine} \circ \text{layoutLine}) \circ \text{flatten}$$

The outer composition represents the three aforementioned phases, the second phase being itself implemented as a map over two per-line sub-phases. What follows are the definitions of flatten, layoutLine, positionLine, and positionLines. Signatures which are gray have definitions which are identical to that found in [7], so we don't reproduce them here. Functions typeset in sans-serif font are primitives, and include map, zip, and zipWith, which have their usual meanings.

$$\begin{array}{lcl} t & ::= & \mathsf{Atom}(z) \mid \mathsf{Wrap}_{id}(t, \mathit{padding}) \mid \mathsf{JoinV}(t,t) \mid \mathsf{JoinH}(t,t) \\ \mathit{stack}\langle\alpha\rangle & ::= & \mathsf{Stack}(\alpha, \mathit{cells}) \text{ where } \mathit{cells} ::= [\mathsf{Cell}_{id}(\Sigma_p)] \\ \mathit{gList}\langle\alpha\rangle & ::= & \mathsf{Empty} \mid \mathsf{GList}(\alpha, [(n,\alpha)]) \end{array}$$

[2]We use the terms h-gadget and v-gadget as introduced in [6] because they serve similar functions, but there is a small difference in our notion of v-gadget. In [6], multiple v-gadgets can appear between lines because they serve two purposes: to separate fragments vertically, but also to make up the boundary of the s-blocks which surround the fragments. In $\mathrm{L1_P}$ and our algorithm, we are able to more directly calculate the *leading* between two lines, which is the distance between their baselines. This is the value recorded in a v-gadget, and it is the reason why there is *one* v-gadget per inter-line space, rather than many.

Advance & Leading

$$\text{spaceBetween } \textit{cells } \textit{cells} = (n, n)$$
$$\text{spaceBetween}_\text{S } \textit{stack}\langle\alpha\rangle \ \textit{stack}\langle\alpha\rangle = n$$
$$\text{leading}_\text{S } \textit{stack}\langle\ddot{x}\rangle \ \textit{stack}\langle\dot{x}\rangle = n$$
$$\text{leading}_\text{R } [\textit{stack}\langle\ddot{x}\rangle] \ [\textit{stack}\langle\dot{x}\rangle] = n$$

Union & Wrap

$$\text{wrap}_\text{S } \textit{id } \textit{padding } \textit{stack}\langle\alpha\rangle = \textit{stack}\langle\alpha\rangle$$
$$\text{wrap}_\text{R } \textit{id padding } [\textit{stack}\langle\alpha\rangle] = [\textit{stack}\langle\alpha\rangle]$$
$$\text{wrap}_\text{L } \textit{id padding } [[\textit{stack}\langle\alpha\rangle]] = [[\textit{stack}\langle\alpha\rangle]]$$

Layout

$$\text{join}_\text{V } [\alpha] \ [\alpha] = [\alpha]$$
$$\text{join}_\text{H } [[\alpha]] \ [[\alpha]] = [[\alpha]]$$
$$\text{flatten } t = [[\textit{stack}\langle x\rangle]]$$
$$\text{layoutLine } [\textit{stack}\langle x\rangle] = \textit{gList}\langle\textit{stack}\langle x\rangle\rangle$$
$$\text{positionLine } \textit{gList}\langle\textit{stack}\langle x\rangle\rangle = \textit{gList}\langle\textit{stack}\langle\dot{x}\rangle\rangle$$
$$\text{positionLine\_ } n \ [(n, \textit{stack}\langle x\rangle)] = [(n, \textit{stack}\langle\dot{x}\rangle)]$$
$$\text{width}_\text{S } \textit{stack}\langle x\rangle = n$$
$$\text{positionH } n \ \textit{stack}\langle x\rangle = \textit{stack}\langle\dot{x}\rangle$$
$$\text{positionLines } [\textit{gList}\langle\textit{stack}\langle\dot{x}\rangle\rangle] = \textit{gList}\langle\textit{gList}\langle\textit{stack}\langle\ddot{x}\rangle\rangle\rangle$$
$$\text{positionLines\_ } [\textit{stack}\langle\ddot{x}\rangle] \ n \ [\textit{gList}\langle\textit{stack}\langle\dot{x}\rangle\rangle] = [(n, \textit{gList}\langle\textit{stack}\langle\ddot{x}\rangle\rangle)]$$
$$\text{stacks } \textit{gList}\langle\alpha\rangle = [\alpha]$$
$$\text{positionV } n \ \textit{stack}\langle\dot{x}\rangle = \textit{stack}\langle\ddot{x}\rangle$$

Leading

$$\text{spaceBetween}_\text{S } \mathsf{Stack}(x_i, \textit{cells}_i) \ \mathsf{Stack}(x_j, \textit{cells}_j) = n_i + n_j \text{ where } (n_i, n_j) = \text{spaceBetween } x_i \ x_j$$

$$\begin{aligned}\text{leading}_\text{S } \mathsf{Stack}(\ddot{x}_i, \textit{cells}_i) \ \mathsf{Stack}(\dot{x}_j, \textit{cells}_j) &= \text{bottom } \ddot{x}_i + n_i + n_j \text{when left } \ddot{x}_i - n_i < \text{right } \dot{x}_j + n_j \text{ and} \\ &\qquad\qquad \text{left } \dot{x}_j - n_j < \text{right } \ddot{x}_i + n_i \\ &= 0 \qquad\qquad \text{otherwise}\end{aligned}$$

$$\text{leading}_\text{R } \textit{line}_i \ \textit{line}_j = \text{maximum } [\text{leading}_\text{S } \textit{stack}_i \ \textit{stack}_j \mid \text{for each } \textit{stack}_i \in \textit{line}_i \text{ and } \textit{stack}_j \in \textit{line}_j]$$

Union & Wrap

$$\begin{aligned}\text{wrap}_\text{R } \textit{id padding stacks} &= \mathsf{map} \ (\text{wrap}_\text{S } \textit{id padding}) \ \textit{stacks} \\ \text{wrap}_\text{L } \textit{id padding lines} &= \mathsf{map} \ (\text{wrap}_\text{R } \textit{id padding}) \ \textit{lines}\end{aligned}$$

Layout

$$\begin{aligned}\text{join}_\text{H } a \ b &= a \ +\!\!+ \ b && \text{when } a = [] \text{ or } b = [] \\ \text{join}_\text{H } a \ b &= a' \ +\!\!+ \ [l \ +\!\!+ \ r] \ +\!\!+ \ b' && \text{otherwise}\end{aligned}$$

where

a' (resp. b') are all but the last (resp. first) line of a (resp. b), and l (resp. r) is the first line of a (resp. b).

$$\begin{aligned}\text{flatten } \mathsf{Atom}(x) &= [[\mathsf{Stack}(x, [])]] \\ \text{flatten } \mathsf{JoinV}(t_1, t_2) &= \text{join}_\text{V } (\text{flatten } t_1) \ (\text{flatten } t_2) \\ \text{flatten } \mathsf{JoinH}(t_1, t_2) &= \text{join}_\text{H } (\text{flatten } t_1) \ (\text{flatten } t_2) \\ \text{flatten } \mathsf{Wrap}_{\textit{id}}(t, \ \textit{padding}) &= \text{wrap}_\text{R } \textit{id padding } (\text{flatten } t)\end{aligned}$$

$$\begin{aligned}\text{layoutLine } [] &= \mathsf{Empty} \\ \text{layoutLine } \textit{cell} : \textit{rest} &= \mathsf{GList}(\textit{cell}, \mathsf{zip} \ \textit{gadgets } \textit{rest}) \\ &\qquad \text{where } \textit{gadgets} = \mathsf{zipWith} \ \textit{spaceBetween}_\text{S} \ (\textit{cell} : \textit{rest}) \ \textit{rest}\end{aligned}$$

$$
\begin{aligned}
\text{positionLine } \mathsf{Empty} &= \mathsf{Empty} \\
\text{positionLine } \mathsf{GList}(\mathit{stack}, \mathit{rest}) &= \mathsf{GList}(\text{positionH } 0 \ \mathit{stack},\ \text{positionLine\_ } (\text{width}_\text{S}\ \mathit{stack})\ \mathit{rest})
\end{aligned}
$$

$$
\begin{aligned}
\text{positionLine\_ } x\ [] &= [] \\
\text{positionLine\_ } x\ ((g, \mathit{stack}) : \mathit{rest}) &= (g, \text{positionH } x'\ \mathit{stack}) : \text{positionLine\_ } (x' + w)\ \mathit{rest} \\
&\quad \text{where} \\
&\qquad x' = x + g \\
&\qquad w = \text{width}_\text{S}\ \mathit{stack}
\end{aligned}
$$

$$
\begin{aligned}
\text{width}_\text{S}\ \mathsf{Stack}(x, \mathit{cells}) &= \text{ The width of } x \\
\text{positionH } n\ \mathsf{Stack}(x, \mathit{cells}) &= \mathsf{Stack}(\dot{x}, \mathit{cells}) \\
&\quad \text{where } \dot{x} \text{ is an atom whose left edge is } n \text{ and whose size is } x
\end{aligned}
$$

$$
\begin{aligned}
\text{positionLines } [] &= \mathsf{Empty} \\
\text{positionLines } \mathit{line} : \mathit{rest} &= \mathsf{GList}(\mathit{line}_\text{T},\ \text{positionLines\_ } (\text{stacks } \mathit{line}_\text{T})\ 0\ \mathit{rest}) \\
&\quad \text{where } \mathit{line}_\text{T} = \mathsf{map}\ (\text{positionV } 0)\ \mathit{line}
\end{aligned}
$$

$$
\begin{aligned}
\text{positionLines\_ } \mathit{above}\ y\ [] &= [] \\
\text{positionLines\_ } \mathit{above}\ y\ (\mathit{line} : \mathit{rest}) &= (l, \mathit{line}_\text{T}) : \text{positionLines\_ } (\mathit{above}\ +\!\!+\ \mathit{stacks}_\text{T})\ l\ \mathit{rest} \\
&\quad \text{where} \\
&\qquad y' \quad\ = \text{leading}_\text{R}\ \mathit{above}\ (\text{stacks } \mathit{line}) \\
&\qquad \mathit{line}_\text{T} = \mathsf{map}\ (\text{positionV } y')\ \mathit{line} \\
&\qquad \mathit{stacks}_\text{T} = \text{stacks } \mathit{line}_\text{T} \\
&\qquad l \quad\ = y' - y
\end{aligned}
$$

$$
\begin{aligned}
\text{stacks } \mathsf{GList}(\mathit{stack}, \mathit{rest}) &= \mathit{stack} : \mathsf{map}\ \mathsf{snd}\ \mathit{rest} \\
\text{positionV } n\ \mathsf{Stack}(\dot{x}, \mathit{cells}) &= \mathsf{Stack}(\ddot{x}, \mathit{cells}) \\
&\quad \text{where } \ddot{x} \text{ is an atom whose top edge is } n
\end{aligned}
$$