Pantograph: A Fluid and Typed Structure Editor

JACOB PRINZ, University of Maryland, USA
HENRY BLANCHETTE, University of Maryland, USA
LEONIDAS LAMPROPOULOS, University of Maryland, USA

Structure editors operate directly on a program's syntactic tree structure. At first glance, this allows for the exciting possibility that such an editor could enforce correctness properties: programs could be well-formed and sometimes even well-typed by construction. Unfortunately, traditional approaches to structure editing that attempt to rigidly enforce these properties face a seemingly fundamental problem, known in the literature as *viscosity*. Making changes to existing programs often requires temporarily breaking program structure—but disallowing such changes makes it difficult to edit programs!

In this paper, we present a scheme for structure editing which always maintains a valid program structure without sacrificing the fluidity necessary to freely edit programs. Two key pieces help solve this puzzle: first, we develop a novel generalization of *selection* for tree-based structures that properly generalizes text-based selection and editing, allowing users to freely rearrange pieces of code by cutting and pasting one-hole contexts; second, we type these one-hole contexts with a category of *type diffs* and explore the metatheory of the system that arises for maintaining well-typedness systematically. We implement our approach as an editor called *Pantograph*, and we conduct a study in which we successfully taught students to program with Pantograph and compare their performance against a traditional text editor.

CCS Concepts: • **Software and its engineering** → **General programming languages**; • **Theory of computation** → **Type theory**.

Additional Key Words and Phrases: structure editors, type inference, type errors, term rewriting

ACM Reference Format:
Jacob Prinz, Henry Blanchette, and Leonidas Lampropoulos. 2025. Pantograph: A Fluid and Typed Structure Editor. *Proc. ACM Program. Lang.* 9, POPL, Article 28 (January 2025), 30 pages. https://doi.org/10.1145/3704864

1 Introduction

Structure editors allow a programmer to view and operate on the tree structure of their source code. For decades, many structure editors such as Alfa [Allgren 1996], Scratch [Maloney et al. 2010], MPS [Voelter 2013], or Hazel [Omar et al. 2017], have allowed for partial programs by leaving the nodes of the program that are yet to be written as (usually typed) holes, which can be filled by any construct of the correct type.

For a concrete example, consider the following expression in some such structure editor, which filters out the negative numbers from the result of appending a list `l1` to a term hole of type `List Int`:

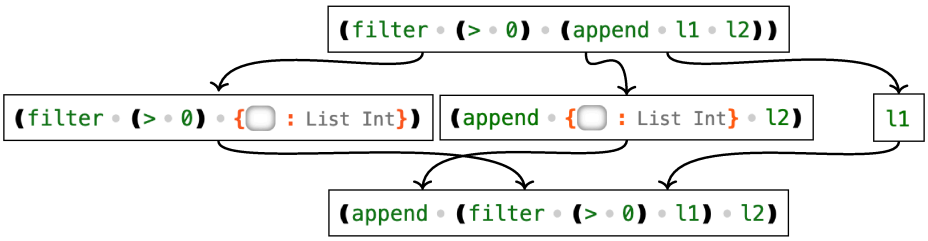
```
(filter • (> • 0) • (append • l1 • {□ : List Int}))
```

The programmer may then place their cursor in the term hole and fill it with a value `l2` in scope:

```
(filter • (> • 0) • (append • l1 • l2))
```

Authors' Contact Information: Jacob Prinz, University of Maryland, College Park, USA, jprinz@umd.edu; Henry Blanchette, University of Maryland, College Park, USA, blancheh@umd.edu; Leonidas Lampropoulos, University of Maryland, College Park, USA, leonidas@umd.edu.

ACM 2475-1421/2025/1-ART28
https://doi.org/10.1145/3704864



(a) Structured editing traditionally requires breaking a term into a forest of syntactically valid components.

edit	state	problems
select	(filter (> 0) (append l1 l2))	
cut	(filter (> 0) l1 l2))	ill-formed, ill-typed
paste	(append (filter (> 0) l1 l2))	ill-typed
select	(append (filter (> 0) l1 l2))	ill-typed
cut	(append (filter (> 0) l1)	ill-formed, ill-typed
paste	(append (filter (> 0) l1) l2)	

(b) Traditional text editing permits convenient grammar- and type-breaking edits.

Fig. 1. Structured vs Text Editing Example

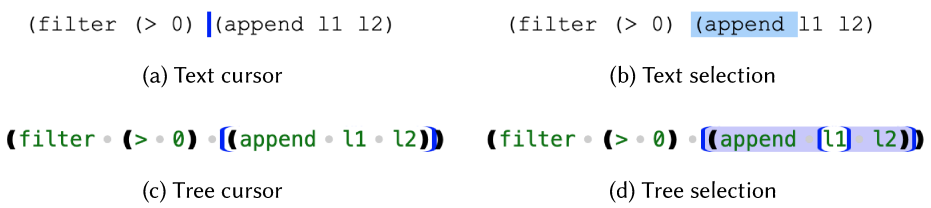
In this manner one can fill in the entire program, and never be permitted to make a syntax or type error. Unsurprisingly, this quite rigid approach very quickly breaks down: how can one edit existing code?

Suppose that the programmer realizes that l2 never contains negative numbers, and decides to optimize the expression by moving the execution of the append operation to after the filter. For this, the mechanism of filling typed holes won't help. In fact, the very constraint that programmers must work with entire typed terms becomes burdensome. The details depend on the editor, but as Figure 1a shows, rearranging expressions like this is difficult when you can only operate on entire terms: one essentially has to break down the tree structure into a forest of small sub-trees—in our example filter (> 0) □, append □ l2, and l1—and recombine them after the fact. This difficulty to edit existing code is known as viscosity in the literature [Blackwell and Green 2003].

By contrast, this edit is strikingly easy to make in a text editor, which allows users to break the grammatical and semantic structure of the program. In particular, one can make this edit with two cut and paste operations as shown in Figure 1b. Each cut operation temporarily breaks the grammatical structure, and the types don't work out until the very end. Still, programmers perform such operations all the time.

One approach to solving the viscosity problem is to introduce specific useful actions that make particular structured program edits. For example, in Alfa [Allgren 1996], the user edits the program though a menu of language-specific actions. This contrasts with text editing, which has a simple core interface which allows any edit on any language, on top of which language specific features may be added.

Another approach to solving the viscosity problem, explored by recent work in the structure editor literature, is to allow the user to make text-like edits to the linear projection of a structured program: rather than rigidly operating on a program only as an intrinsically typed term, such editors allow the user to break the program into syntax which can afterwards be interpreted into typed terms. In particular, Tylr [Moon et al. 2022, 2023] allows a user to move delimiters (such as parentheses), temporarily breaking the grammar of the program, but enforces that these delimiters



(a) Text cursor (b) Text selection

(c) Tree cursor (d) Tree selection

Fig. 2. Text- and Tree-based Cursors and Selection

are eventually placed in a valid location. Similarly, recent versions of Hazel (as described in [Zhao et al. 2024]) allow the user to operate over an untyped grammar, and then mark the program with error forms after the user's edit. Still, the question remains:

Could there be a general fluid editor which operates on strictly structured and typed programs?

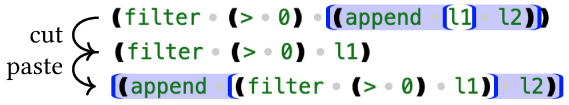
Such an editor would never reduce terms to untyped or ill-formed syntax, and would be characterized by never needing to re-parse or re-typecheck the program. But at the same time, unlike traditional attempts at such editors, it would not weigh down the programmer when editing existing code. Further, it would not require the programmer to learn various ad-hoc actions for specific edits, but rather have a small consistent language-generic interface. If indeed possible, what advantages would such an editor facilitate?

In this paper, we take on this challenge by demonstrating the feasibility and exploring the benefits of such an intrinsically typed editing scheme. A key observation that enables this fluidity is that existing structure editors lack a proper generalization of text-based selection and related operations. We develop this single language-generic abstraction based on the well-known notion of one-hole contexts, and it allows for a wide range of structure-preserving operations. We imbue these operations with a typing structure and explore their metatheory, rooting our design in firm formal foundations.

One-Hole Contexts. Standard text editors allow two different mechanisms of editing a program: a cursor and a selection. In a text editor, cursors exist between characters, while selections correspond to the area between two text based cursors. Going back to our earlier example, the text-based cursor in Figure 2a is before "(append", while the text-based selection of the same string (Figure 2b) ranges between two cursor locations and (roughly) corresponds to an application of append to a value. Traditional structure editors offer users a tree-based analogue for cursors: where text-based cursors live before characters, tree-based cursors live on top of tree nodes. For example, the tree-based cursor corresponding to the text one ranges over an entire inner subtree (Figure 2c).

However, traditional structure editors have no analogue for selection. Generalizing text-based selection to trees would mean that structure editor users should be able to select and edit what lies between two tree-based cursors; they should be able to select and edit the part of the program that lies between one node and one of its descendants; they should be able to select and edit expressions with a single subexpression missing; they should be able to select and edit *one-hole contexts*.

One of the contributions of this paper is to introduce such a generalization of structured editing: when a one-hole context is cut, the expression in its hole takes its place; when a one-hole context is pasted onto an expression, that expression fills in its hole. For example, our scheme allows the following manipulation with a single cut and paste of a one hole context:



Such selection-based manipulation of trees properly generalizes selection from traditional text-based editing, allowing for fluid editing while preserving the well-formedness of programs at all times. In Section 2, we provide additional intuition about the edits that this approach enables through examples. On its own, however, it does nothing to preserve the well-typedness of programs.

Typed Edits. For an editor to respect not only syntactic structure but also types, edits must be able to affect multiple locations in a program at once. Consider, for instance, the following simple recursive program which applies the not function to every element of an input list l:

```
let map-not : List Bool → List Bool =
    λ l : List Bool .
        match l with
        nil ⇒ nil
        cons h t ⇒ (cons (not h) (map-not t)) in
(map-not (cons true (cons false nil)))
```

Naturally, a functional programmer will want to abstract away the recursion pattern (here, map) from the particular function being applied (here, not). This seemingly requires five distinct edits: an argument must be added to the signature, a lambda to the definition, and an application at both function calls, including the recursive call, and a call to f (replacing the call to not). This amounts to inserting five different one-hole contexts into the program:

```
let map : [(Bool → Bool) →] List Bool → List Bool =
    [λ f : Bool → Bool .] λ l : List Bool .
        match l with
        nil ⇒ nil
        cons h t ⇒ (cons [(f h)] (map [f] t)) in
([map] [not] (cons true (cons false nil)))
```

Any of these five edits made individually results in an intermediate ill-typed state. Our second main contribution is a system of typed edits where each one-hole context is typed by a *type diff*, corresponding to the difference between its inner and its outer type. Armed with this abstraction, programmers need only to insert a lambda at the body of map, and the extra parameter in the type and applications and holes at the two call sites are automatically inserted by the system.

This intrinsically typed editing system facilitates useful behaviors that are not possible in other editing systems. Here, every edit at any term or type results in a well-typed refactoring operation that updates corresponding parts of the program. In some cases these updates can insert intrinsic type error forms in order to encode type errors into the program itself. However, this approach has access to more information than a traditional type checker because it inputs the edit rather than merely the untyped state after the edit. This allows it to automatically update corresponding parts of the program, including by placing errors, in ways that are impossible for a traditional type checking or inference algorithm.

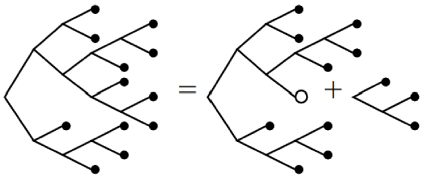
We present the following contributions:

- We present a new structure editing paradigm that we call "zipper editing" which generalizes text selection to tree-based structures and provides standard edit operations which always maintain the well-formedness of the source program (Section 2).
- We define a category of diffs for typing these edits, in which the edits act as refactoring operations on the source program which preserve well-typedness—modulo sometimes inserting explicit type error boundaries (Section 3). We model such refactoring operations as propagating typed edits through a program in a core language with explicit typed edits (Section 4); and we explore the metatheory of this language, proving normalization, type-preservation, and confluence (Section 5).

- We implement our approach in a structure editor we call Pantograph. To demonstrate the feasibility of our editing paradigm, we conducted a user study in which participants solved functional programming tasks in both our prototype implementation and a traditional text editor (Section 6).

We discuss limitations of our editing paradigm in Section 7, related work in Section 8, and finally conclude with future work in Section 9.

2 Zipper Editing



Zippers, originally proposed by Huet [1997], allow trees to be decomposed into a one-hole context and a "subtree of interest", as in the figure on the right found in McBride [2001]. This decomposition allows for efficient local editing of tree-like data structures. Naturally, it is well known that this notion can represent cursors in both text and structure editors. In text editors—ignoring their 2D layout aspect—the cursor position could be seen as a pair of strings: the one hole context that precedes the cursor, and the text that follows. In structure editors, the cursor position is the one-hole context that surrounds a program node, together with the node under focus. In this representation, local program operations such as inserting a construct into the program or navigating the cursor are easily and efficiently expressed [Huet 1997]. For example, Figure 3 shows how easy it is to insert a lambda around a term at the cursor in Pantograph.

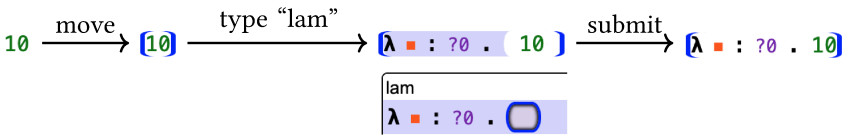


Fig. 3. Inserting a λ around a term of the program: Users query the name of the construction that they want to insert, and Pantograph wraps it around the term at the cursor; ?0 represents an unknown type.

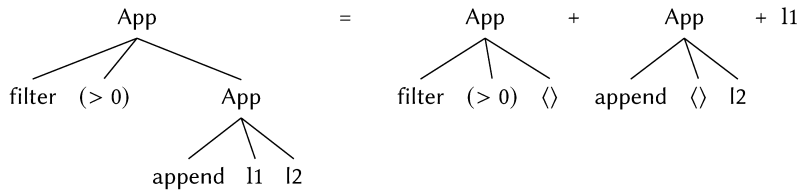
For concreteness, consider the following simple functional language core, consisting of λ abstractions, applications, variables, **let** expressions, and term holes.

$$t ::= \textbf{let}\ x : T = t\ \textbf{in}\ t \mid \lambda x : T\,.\,t \mid t\ t \mid x$$
$$C ::= \textbf{let}\ x : T = C\ \textbf{in}\ t \mid \textbf{let}\ x : T = t\ \textbf{in}\ C \mid \lambda x : T\,.\,C \mid C\ t \mid t\ C \mid \langle\rangle$$

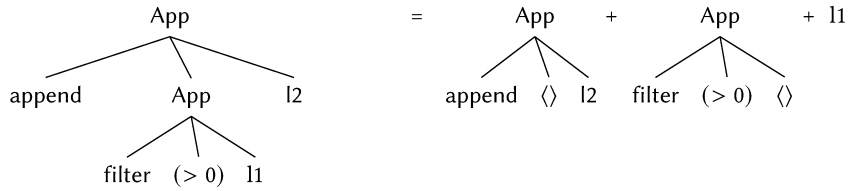
The corresponding grammar of one-hole contexts allows for exactly one subterm to be missing at the hole $\langle\rangle$. Given a one hole context C and a term t, we write $C[t]$ to represent filling in the hole in C with t to get a single term. For instance, if the term in Figure 3 was the focus of some bigger program $C[10]$, inserting that λ expression at the cursor results in the new program $C[\lambda x.10]$.

However, text editors wouldn't be very useful without the ability to select text. But what is text selection? Following the same intuition as before, it's a triple of strings: the text before the selection, the selection itself, and the text after it. How does that notion generalize to trees?

According to this analogy, selection in trees can be defined as a triple consisting of two one hole contexts C_1 and C_2 and one term t, arranged as $C_1[C_2[t]]$, where C_2 is the selected region. In our example from the introduction, the selection of append and l2 corresponds to the following decomposition of the whole program:



This decomposition allows for very quick selection-based operations similar to text editing. For example, by reordering the one hole contexts, we can quickly rearrange this expression:



Any number of features that a structure editor could have would allow this particular edit. But, we take inspiration from the design of the standard text editor, where a few simple operations enable arbitrary text manipulation. One-hole context selection is similarly easy to use, as shown in Figure 4. It also is similarly versatile; Figure 5 shows a wide variety of edits that can be made with these selections. In each example, the user performs a single cut and paste operation.

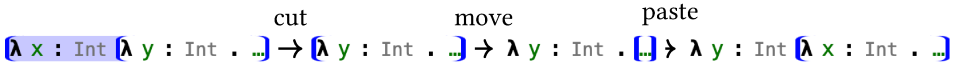


Fig. 4. Cutting and pasting a selection into a different location in Pantograph. Zipper editing lends itself to a nice user interface similar to text editing. The user may make a selection with a familiar click and drag motion. The user may also cut selections into a clipboard, and paste them later at the cursor.

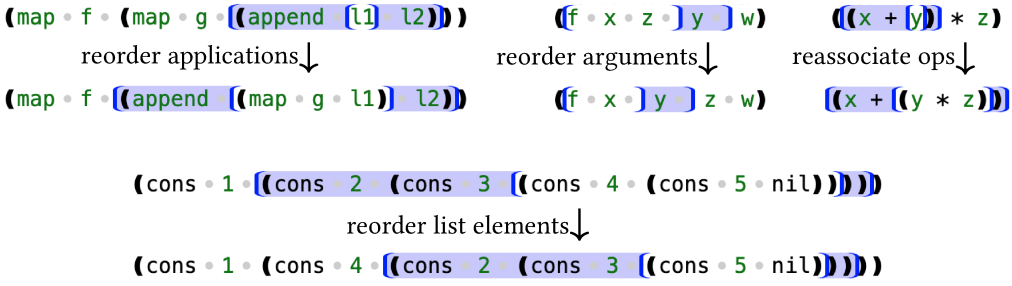


Fig. 5. Various edits each performed with a single cut and paste of a one-hole context selection. The first selection in each example is made by the user, but the second selection is only illustrative.

$$T ::= \mathsf{Int} \mid \mathsf{Bool} \mid T \to T \mid \mathsf{List}\ T \qquad \frac{}{\Gamma \vdash \square_T : T}$$

$$\frac{i \in \mathbb{Z}}{\Gamma \vdash i : \mathsf{Int}} \quad \frac{b \in \mathbb{B}}{\Gamma \vdash b : \mathsf{Bool}} \quad \frac{}{\Gamma \vdash \mathsf{nil} : \mathsf{List}\ T} \quad \frac{}{\Gamma \vdash \mathsf{cons}\ h\ t : T \to \mathsf{List}\ T \to \mathsf{List}\ T}$$

$$\frac{}{\Gamma, x : T \vdash x : T} \quad \frac{\Gamma, x : T_1 \vdash b : T_2}{\Gamma \vdash \boldsymbol{\lambda} x : T_1 \,.\, b : T_1 \to T_2} \quad \frac{\Gamma \vdash f : T_1 \to T_2 \qquad \Gamma \vdash a : T_1}{\Gamma \vdash f\ a : T_2}$$

$$\frac{\Gamma, x : T_1 \vdash a : T_1 \qquad \Gamma, x : T_1 \vdash b : T_2}{\Gamma \vdash \textbf{let}\ x : T_1 = a\ \textbf{in}\ b : T_2} \quad \frac{\Gamma \vdash a : \mathsf{List}\ T_1 \qquad \Gamma \vdash b : T_2 \qquad \Gamma, h : T_1, t : \mathsf{List}\ T_1 \vdash c : T_2}{\Gamma \vdash \textbf{match}\ a\ \textbf{with}\ \mathsf{nil} \Rightarrow b \,;\, \mathsf{cons}\ h\ t \Rightarrow c : T_2}$$

Fig. 6. A typed core language

3 Typed Editing

Zipper editing, like the traditional structure editing it extends, preserves the syntactic well-formedness of programs. However, it does not necessarily preserve well-typedness. Let's add a few types to our core language of the previous section (integers, booleans, and lists) that will enable us to encode the map example of the introduction, and define a standard simple type system with recursive let, pattern matching on lists, and typed holes $\square_T$, as shown in Figure 6.

Recall the edits required to add a parameter to the map function (Figure 1). While the result of performing all four of the edits yields a well-typed program, performing just the first edit of inserting $\lambda \mathsf{f} : \mathsf{Bool} \to \mathsf{Bool} \,.\, \langle\rangle$ leaves the program in an intermediary ill-typed state since type annotation of map would still be $\mathsf{List\ Bool} \to \mathsf{List\ Bool}$.

Any system that aims to operate on intrinsically typed terms needs to account for how such an edit changes types in the program. To that end, we will introduce a grammar of *type diffs*, encoding precisely how a type is transformed into a new type after an edit. We will use δ to range over these diffs and write $T_1 \overset{\delta}{\Longrightarrow} T_2$ for a diff δ which changes type T_1 to T_2. The grammar of diffs for our core language is shown below; we will systematically generalize to other language features in Section 5:

$$\delta ::= \mathsf{Int} \mid \mathsf{Bool} \mid \delta \to \delta \mid \mathsf{List}\ \delta \mid {}^{+}\langle T \to \langle \delta \rangle\rangle \mid {}^{-}\langle T \to \langle \delta \rangle\rangle \mid T \,/\, T$$

The first two of these constructors are identity diffs which don't alter the type:

$$\mathsf{Int} \overset{\mathsf{Int}}{\Longrightarrow} \mathsf{Int} \qquad \mathsf{Bool} \overset{\mathsf{Bool}}{\Longrightarrow} \mathsf{Bool}$$

The next two constructors represent diffs which preserve a top level constructor of a type, and apply diffs to the child types:

$$\frac{T_1 \overset{\delta_1}{\Longrightarrow} T_1' \qquad T_2 \overset{\delta_2}{\Longrightarrow} T_2'}{T_1 \to T_2 \overset{\delta_1 \to \delta_2}{\Longrightarrow} T_1' \to T_2'} \qquad \frac{T \overset{\delta}{\Longrightarrow} T'}{\mathsf{List}\ T \overset{\mathsf{List}\ \delta}{\Longrightarrow} \mathsf{List}\ T'}$$

The final three constructors represent diffs which actually alter types: the first two insert or remove a function type respectively, and then (potentially) change the output type of the function by a diff; the latter simply replaces one type with another:

$$\frac{B_1 \overset{\delta}{\Longrightarrow} B_2}{B_1 \overset{{}^{+}\langle A \to \langle \delta \rangle\rangle}{\Longrightarrow} A \to B_2} \qquad \frac{B_1 \overset{\delta}{\Longrightarrow} B_2}{A \to B_1 \overset{{}^{-}\langle A \to \langle \delta \rangle\rangle}{\Longrightarrow} B_2} \qquad \frac{}{T \overset{T \,/\, T'}{\Longrightarrow} T'}$$

Let's once again revisit our map example, where the user inserts a one hole context $\lambda f : \text{Bool} \to \text{Bool} \,.\, \langle\rangle$. This can be given the following diff between two types:

$$T_1 \overset{{}^{+}\langle (\text{Bool} \to \text{Bool}) \to \langle \text{List Bool} \to \text{List Bool} \rangle\rangle}{\Longrightarrow} T_2$$

$$\begin{aligned} \text{where} \quad T_1 &= \text{List Bool} \to \text{List Bool} \\ T_2 &= (\text{Bool} \to \text{Bool}) \to \text{List Bool} \to \text{List Bool} \end{aligned}$$

When the user inserts a λ expression around the body of map, Pantograph automatically makes the edits necessary to keep the program well typed. The system adds an application to a hole at the two call sites, and alters the type signature, as shown in Figure 7. In the next section, we will explain how our typed editing system uses diffs to calculate and perform these changes. But first, we will further demonstrate the system's capabilities with a few more examples.

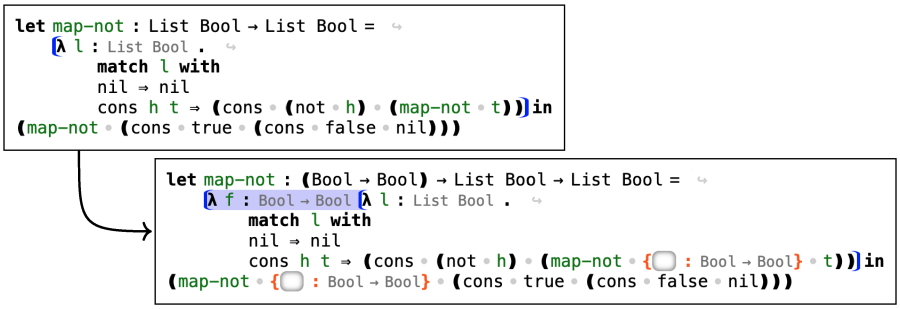


Fig. 7. Typed editing example 1: adding a parameter to map-not.

The inductive structure of diffs allows us to represent edits to the higher-order structure of functions. For example, suppose that the user wants to generalize map so that the mapped function takes an index. A type diff can describe the resulting change to the type of map:

$${}^{+}\langle \text{Int} \to \langle \text{Bool} \to \text{Bool} \rangle\rangle \to \text{List Bool} \to \text{List Bool}.$$

Again, when the user edits the type of f, Pantograph makes various edits to the rest of the code to maintain its well-typedness. This time, Pantograph wraps a λ around the not function at the call site to account for the change in its type (Figure 8).

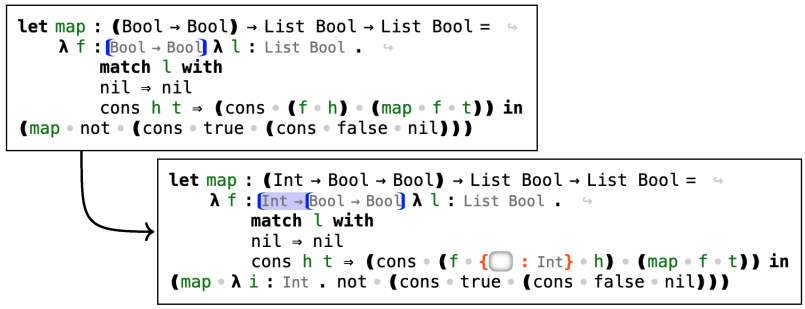


Fig. 8. Typed editing example 2: adding a higher-order parameter to map.

Of course, it is not always desirable for an editor to fix typing issues automatically. Sometimes, Pantograph leaves errors in the program for the user to fix later. For example, suppose that the user deletes the f parameter from a finished map function. This amounts to removing the one hole context $\lambda f : \text{Bool} \to \text{Bool} \,.\, \langle\rangle$ from the program. This time, the edit gets a diff which subtracts a function argument:

$$\overline{\langle(\text{Bool} \to \text{Bool}) \to \langle\text{List Bool} \to \text{List Bool}\rangle\rangle}$$

When the user makes the deletion, it leaves a couple of errors in the program. There is an unbound call to f, and an out of place argument at the two call sites to map. While the system could simply replace the former with a hole and remove the latter two from the program, this would likely erase valuable work that the programmer wanted to keep.

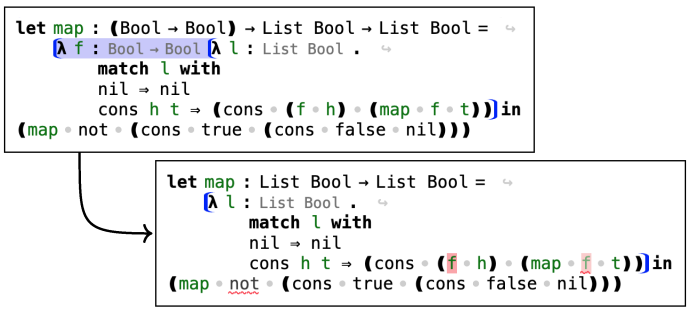


Fig. 9. Typed editing example 3: deleting a parameter. This deletion requires updating the rest the program according to the corresponding type diff, which also describes a change in the context in the body of map where f is no longer bound.

To allow such errors to exist in an otherwise well typed program, Pantograph has three final constructions. The first two of these forms are: *free variables*, which are variables whose binders have been removed, like f in the example; and *commented applications*, which are function arguments which no longer fit in place, like the first argument to map.

$$\frac{\Gamma \vdash f : A \qquad \Gamma \vdash a : B}{\Gamma \vdash \text{//}\langle\langle f\rangle\ a\rangle : A}\ \textsc{Commented Application} \qquad \frac{}{\Gamma \vdash \text{//}x_T : T}\ \textsc{Free Variable}$$

Any errors resulting from the user deleting a selection can be accounted for with these two constructs. However, there is one final way for a user to create an error that can't be resolved in this way: the user can simply replace one type annotation with another. Suppose that the user decides to delete the Bool type in the output of the higher order argument to map, and replace it with Int. This is represented with a diff that has a replacement:

$$(\text{Bool} \to \text{Bool} \,/\, \text{Int}) \to \text{List Bool} \to \text{List Bool}$$

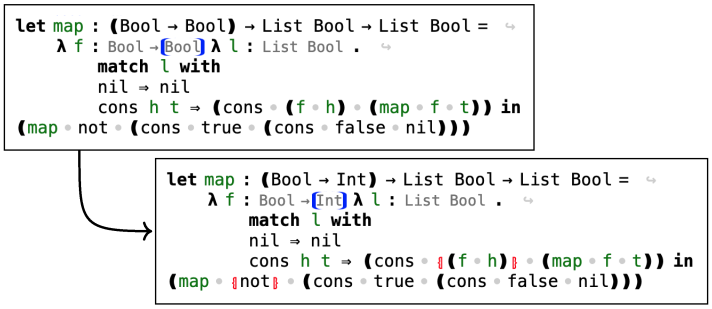


Fig. 10. Typed editing example 4: replacing one type with another. The corresponding type diff for this edit leaves some expressions ill-typed, so they must be placed in type error boundaries.

Pantograph is able to update the annotation of map, but it can't resolve the type error at the call to f so it inserts a *type error boundary*. This is the third and final form that Pantograph uses to represent errors in a program. Its typing rule reflects it's meaning: it has a single child term, which can have a different type than the surrounding term.

$$\frac{\Gamma \vdash t : T_1}{\Gamma \vdash \{t\}^{!}_{T_1/T_2} : T_2} \textsc{Type Error Boundary}$$

Type error boundaries are similar to the type errors that are placed by usual type checkers, except that they are a first-class term within the program that the user can interact with. Such first-class errors are characteristic of structure editors that rigidly operate on typed terms (such as early iterations of Hazel [Omar et al. 2017]) because they allow errors to exist without resorting to fully untyped syntax. However, Pantograph's ability to respond to an edit non-locally makes them especially useful. The user can select and delete it just like any other form in the language — and Pantograph will alter the surrounding code to fit the type inside!

For instance, suppose that the programmer decides that the output type of Int is actually the correct type for the output list of map. The programmer can delete the type error boundary around f, resulting in the diff Bool / Int (Figure 11). Deleting a type error boundary tells the system that the term inside actually has the desired type, and the system will make the program surrounding it conform to that choice. Crucially, this isn't a special feature of error boundaries; deleting any one hole context will have the same effect.

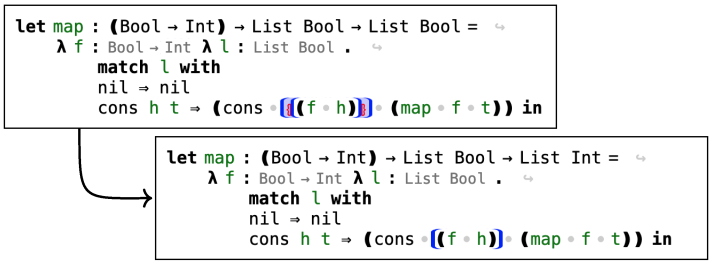


Fig. 11. Typed editing example 5: deleting a type error boundary. When the user deletes the type boundary, Pantograph automatically fixes the output type of map to be Int.

4 Computing Refactorings with diffs, by example

In the previous section, we showed that type diffs precisely encode how an edit changes the type of a term, and gave several examples of how Pantograph changes a program in response. In this section, we will describe at a high level how our system computes these changes automatically. For now, we will elide some technical details in order to facilitate an end-to-end example; then, in the next section, we will formally develop our theory of typed edits in its entirety.

Consider the following simple example of an identity function applied to the constant 10, where the user inserts a $\lambda y : \text{Bool}.\langle\rangle$ one-hole context around the term 10, and Pantograph creates an application around the call to x in response:

$$(\lambda x.x)[10] \xrightarrow{\text{Insert } \lambda y : \text{Bool}.\langle\rangle} (\lambda x.x\ \square_{\text{Bool}})(\lambda y.10)$$

To represent intermediate states of a computation, we introduce two forms $\{t\}^{\uparrow}_{d}$ and $\{t\}^{\downarrow}_{d}$ of *diff boundaries*, which we call *up* and *down* boundaries respectively, into our grammar of terms. Our system inserts an up boundary at the location of the user's edit to represent the change in types induced by the edit:

$$(\lambda x.x)\{\lambda y.10\}^{\uparrow}_{+\langle \text{Bool} \to \langle \text{Int} \rangle \rangle} \qquad \text{(step 1)}$$

Next, Pantograph propagates this boundary through the surrounding program one language form at a time, making use of the corresponding typing rule at each step. In our example the up boundary surrounds the argument to an application, so Pantograph looks at that typing rule, and in particular it's second premise:

$$\frac{\overbrace{\Gamma \vdash t_1 : A \to B}^{\text{Output}} \qquad \overbrace{\Gamma \vdash t_2 : A}^{\text{Unify input diff with } A}}{\Gamma \vdash t_1\ t_2 : B}$$

Pantograph, then, needs to relate the type A of this premise with the diff on the boundary $^{+}\langle \text{Bool} \to \langle \text{Int} \rangle \rangle$. In general this is done using a unification procedure described in the following section; in this case, it discovers that A maps to this diff. Next, this new information needs to be propagated 'through' the form, placing new boundaries around the form or its other children to account for the changes to A. In this case, A appears in the type of the first premise $A \to B$, and so our algorithm creates a down boundary around the left child of the application:

$$\{\lambda x.x\}^{\downarrow}_{+\langle \text{Bool} \to \langle \text{Int} \rangle \rangle \to \text{Int}}(\lambda y.10) \qquad \text{(step 2)}$$

At this point, the down boundary surrounds a λ abstraction so our algorithm looks at the corresponding typing rule (on the right). To proceed, Pantograph must unify the incoming diff with the type of the conclusion of the rule, $A \to B$. The result is that B is mapped to an identity diff Int, so our system ignores it. On the other hand, A maps to the interesting part of the type diff. The other appearance of A in the rule is in the type of x in the context of the premise, which means we need to propagate a change to a *context*.

$$\frac{\overbrace{\Gamma, x : A \vdash t : B}^{\text{Output}}}{\underbrace{\Gamma \vdash \lambda x.t : A \to B}_{\text{Unify input diff with } A \to B}}$$

In order to express such a change, we introduce a notion of *context diffs*, which describe how a context changes, either by altering a variable by a given type diff, inserting a new variable, or removing a variable. We will discuss these further in Section 5.2.

$$\Delta \ ::= \ \varnothing \quad | \quad \Delta, x : \delta \quad | \quad {}^{+}\langle\langle \Delta \rangle, x : T \rangle \quad | \quad {}^{-}\langle\langle \Delta \rangle, x : T \rangle$$

Using a context diff, our system can propagate the knowledge of how the type of x changes into the body of the abstraction. So far we have elided context diffs on the diff boundaries to facilitate a simpler introduction. However, the full form of diff boundaries that we need to use when there is a nontrivial context diff is $\Delta \vdash \delta$, encoding a change to both the context and type. Propagating such a diff results in the following term:

$$(\lambda x : \{x\}^{\downarrow}_{\varnothing, x : {}^{+}\langle \mathsf{Bool} \to \langle \mathsf{Int} \rangle \rangle \vdash Int})(\lambda y.10) \qquad \text{(step 3)}$$

The boundary now surrounds a variable and Pantograph looks up the corresponding rule:

$$\frac{}{\Gamma, x : T \vdash x : T}$$

There are two occurrences of T in the conclusion of the rule, one in the context and one in the type. The existence of such nonlinearities, where the same variable appears twice in the same part of a typing rule, will inform some of the details of our exposition later on. In this case, the algorithm unifies the diff with the first occurrence of the T in the context, creating a new boundary in the opposite direction with a diff corresponding to the other occurrence of T in the type:

$$(\lambda x : \{x\}^{\uparrow}_{{}^{+}\langle \mathsf{Bool} \to \langle \mathsf{Int} \rangle \rangle})(\lambda y.10) \qquad \text{(step 4)}$$

Finally, the system has arrived at the location where the edit must be made. Until now, propagation of the diffs though the syntax was determined by the structure of the typing rules. However, at this final step, the choice of what the system does with the diff is a more subjective design decision—we could chose to simply leave an error boundary. Pantograph automatically inserts arguments to functions when necessary, according to the following rule:

$$\{t\}^{\uparrow}_{\Delta \vdash {}^{+}\langle A \to \langle \delta \rangle \rangle} \quad \rightsquigarrow \quad \{t \ \square_A\}^{\uparrow}_{\Delta \vdash \delta}$$

Pantograph has 8 such *alteration* rules hard-coded in to determine how it makes edits, which we describe fully in Section 5.5. But in this case, applying this rule gives us the final result of the edit:

$$(\lambda x.x \ \square_{\mathsf{Bool}})(\lambda y.10) \qquad \text{(step 5)}$$

5 Diffs and Diff Propagation In Detail

So far we have shown examples of the automatic changes that Pantograph makes in response to a user's edit. In this section, we will finally describe the underlying mechanism in detail. First, we will unify the treatment of type and context diffs, introducing a category of diffs over arbitrary trees. Then, we will describe an algorithm to propagate these boundaries through the program as a small-step operational semantics, and explore the metatheory of the entire system.

5.1 Terms and Types as Trees

To operate on our types, contexts, and typing rules algorithmically, we will represent them in a standard intrinsic style: whereas in extrinsic typing rules the term is included in the typing judgement, in intrinsic ones the premises are implicitly understood to refer to the children of the form. For example, below is the typing rule for an application $t_1\ t_2$ rewritten into intrinsic style; the two premises are derivations for t_1 and t_2.

$$\frac{\Gamma \vdash t_1 : A \to B \qquad \Gamma \vdash t_2 : A}{\Gamma \vdash t_1\ t_2 : B} \quad \xrightarrow{becomes} \quad \frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}$$

Types and contexts both have particular grammars, but almost all of these details are irrelevant to the propagation algorithm. As a result, we can take inspiration from S-expressions [McCarthy 1960], and instead consider both of them as trees over a set of labels l which we represent with s. Each tree consists of a label and a list of children:

$$s ::= l\ \overline{s}$$

We collect all of the type constructors $\{l_{\to}, l_{\texttt{Int}}, l_{\texttt{Bool}}, l_{\texttt{List}}\}$ and context constructors $\{l_{\varnothing}, l_{\square,\square}\}$ into a set of labels l. We also make a label l_x for each variable x, since these appear in typing rules too. In addition, we have a set of metavariables α and for each a label l_α. Finally, we include a label for our typing judgement $l_{\vdash}$. A typing rule can then be understood as $\overline{s} \times s$, a list of trees for each premise, and a tree for the conclusion. For example, we can represent the context and type of the first premise of the **let** typing rule ($\Gamma, x : A \vdash A$) with a tree:

$$(l_{\vdash}\ (l_{\square,\square}\ l_\Gamma\ l_x\ l_A)\ l_A)$$

With this representation, all one hole contexts are simply trees with a child missing; we will use c to represent single-label steps of a one-hole context, and C to represent a list of such steps.

$$c ::= l\ \overline{s}\ \langle\rangle\ \overline{s} \qquad\qquad C ::= \overline{c}$$

5.2 The Category of Diffs

Now that we have abstracted away any specific notion of types, contexts, or judgements into the single abstraction that a tree of labels provides, we can define diffs between arbitrary trees. We denote these with d, generalizing both the type diffs δ and the context diffs Δ. There are only four constructors, in direct correspondence with the diffs presented in Section 3: the first leaves a top level label unchanged, and changes each child by a diff. The second and third constructors add or remove a single label step c, and then apply another diff inside the hole of c. The final constructor simply replaces one tree with another.

$$d ::= l\ \overline{d} \quad | \quad {}^{+}\langle c\langle d\rangle\rangle \quad | \quad {}^{-}\langle c\langle d\rangle\rangle \quad | \quad s \,/\, s$$

We can also generalize our judgement $s_1 \overset{d}{\Longrightarrow} s_2$ to range over these general purpose diffs:

$$\frac{\forall i, s_i \overset{d_i}{\Longrightarrow} s_i'}{l\ \overline{s_i} \overset{l\ \overline{d_i}}{\Longrightarrow} l\ \overline{s_i'}} \qquad \frac{s \overset{d}{\Longrightarrow} s'}{s \overset{{}^{+}\langle c\langle d\rangle\rangle}{\Longrightarrow} c[s']} \qquad \frac{s \overset{d}{\Longrightarrow} s'}{c[s] \overset{{}^{-}\langle c\langle d\rangle\rangle}{\Longrightarrow} s'} \qquad \frac{}{s_1 \overset{s_1\ /\ s_2}{\Longrightarrow} s_2}$$

These diffs admit an identity and a composition operator, which will be useful to define Pantograph's algorithm.

Endpoints. If $s_1 \overset{d}{\Longrightarrow} s_2$, then we define $d.1$ and $d.2$ as the endpoints s_1 and s_2 respectively.

Identity diffs. Given any tree s, there is an identity diff $s \overset{id_s}{\Longrightarrow} s$ given by

$$id_{l\ s_1 \ldots s_n} = l\ id_{s_1} \ldots id_{s_n}$$

We will often write just id when the tree is unambiguous.

Composition. Given two diffs sharing an endpoint $t_1 \overset{d_1}{\Longrightarrow} t_2$ and $t_2 \overset{d_2}{\Longrightarrow} t_3$, we can compose them together: $t_1 \overset{d_1 \circ d_2}{\Longrightarrow} t_3$. Composition is defined recursively over the structure of the diffs. Each diff represents a change to a tree, and their composition is a single diff which performs both changes. Therefore, the resulting diff generally contains all of the '+' and '-' constructors from the two inputs combined. The exception is if we compose two diffs which add and then remove the same one-hole context — then the two cancel out. In the following definition, we write $\overline{d}$ for a list of diffs, and $\overline{d}.1$ (or $\overline{d}.2$) for the list of left (or right) endpoints.

$$\begin{aligned}
(l\ d_1\ \ldots\ d_n) \circ (l\ d_1'\ \ldots\ d_n') &= l\ (d_1 \circ d_1')\ \ldots\ (d_n \circ d_n') \\
{}^{+}\langle c\langle d\rangle\rangle \circ {}^{-}\langle c\langle d'\rangle\rangle &= d \circ d' \\
d \circ {}^{+}\langle c\langle d'\rangle\rangle &= {}^{+}\langle c\langle d \circ d'\rangle\rangle \\
{}^{-}\langle c\langle d\rangle\rangle \circ d' &= {}^{-}\langle c\langle d \circ d'\rangle\rangle \\
{}^{+}\langle \overline{a}.1\ \langle d\rangle\ \overline{b}.1\rangle \circ (l\ \overline{a}\ d'\ \overline{b}) &= {}^{+}\langle \overline{a}.2\ \langle d \circ d'\rangle\ \overline{b}.2\rangle \\
(l\ \overline{a}\ d'\ \overline{b}) \circ {}^{-}\langle \overline{a}.2\ \langle d\rangle\ \overline{b}.2\rangle &= {}^{-}\langle \overline{a}.1\ \langle d \circ d'\rangle\ \overline{b}.1\rangle \\
d_1 \circ d_2 &= d_1.1\ /\ d_2.2 \qquad \text{otherwise}
\end{aligned}$$

Finally, the next two theorems state that identities and compositions behave as expected, forming a category where the objects are trees and the morphisms are diffs:

Theorem 1 (Identity-Compose).

$$id \circ d = d \circ id = d$$

Theorem 2 (Associativity of composition). *Given any three diffs* $s_1 \overset{d_1}{\longrightarrow} s_2 \overset{d_2}{\longrightarrow} s_3 \overset{d_3}{\longrightarrow} s_4$,

$$(d_1 \circ d_2) \circ d_3 = d_1 \circ (d_2 \circ d_3)$$

The proofs of these two theorems proceed by induction over the diffs. The full proofs can be found in the extended version of this article [Prinz et al. 2024], Appendix C.

5.3 Typing One-Hole Contexts with Diffs

As we have shown, Pantograph performs automatic edits when the user inserts or removes a one-hole context. Each automatic edit begins by placing diff boundaries around the location of the edit. We associate each one-hole context with a diff that describes how it changes the type and context of the program where it is inserted. For a one-hole context *step* c, we write $\Delta \vdash c : \delta$ to represent that from the inside to the outside, the context and type of c change by diffs Δ and δ. For most language constructs, the types and contexts inside and outside their one-hole context c are either exactly the same (which gives rise to an identity diff), or unrelated (which gives rise to a replace diff). Only for a few constructs is the relationship between inside and outside types and contexts more involved: λ abstractions, applications, and forms which introduce variables.

$$\begin{gathered}
{}^{-}\langle\langle \Delta \rangle, x : A\rangle \vdash \mathbf{\lambda} x : A \,.\, \langle\rangle : {}^{+}\langle A \to \langle B \rangle\rangle \\
\Delta \vdash \langle\rangle\ t : {}^{-}\langle A \to \langle B \rangle\rangle \\
{}^{-}\langle\langle \Delta \rangle, x : T_1\rangle \vdash \mathbf{let}\ x : T_1 = t\ \mathbf{in}\ \langle\rangle : T_2 \\
{}^{-}\langle\langle \Delta \rangle, x : T_1\rangle \vdash \mathbf{let}\ x : T_1 = \langle\rangle\ \mathbf{in}\ t : T_1\ /\ T_2 \\
{}^{-}\langle\langle {}^{-}\langle\langle \Delta \rangle, h : T\rangle\rangle, t : \mathsf{List}\ T\rangle \vdash \mathbf{match}\ a\ \mathbf{with}\ \mathsf{nil}\ \Rightarrow\ c\ ;\ \mathsf{cons}\ h\ t\ \Rightarrow\ \langle\rangle : \mathsf{List}\ T
\end{gathered}$$

To lift diffs from one-hole context steps to entire one-hole contexts, we can simply compose the diffs along the individual steps:

$$\frac{}{id \vdash \langle\rangle : id} \qquad \frac{\Delta_2 \vdash c : \delta_2 \qquad \Delta_1 \vdash C : \delta_1}{\Delta_1 \circ \Delta_2 \vdash c[C] : \delta_1 \circ \delta_2}$$

This allows us to get a diff for any one hole context which goes from the context and type inside to the context and type outside. In order to get the diff going the other way from the outside to the inside, we define a function which flips a diff and swaps its endpoints:

$$\begin{aligned}
\mathsf{flip}\ (l\ d_1\ \ldots\ d_n) &= l\ (\mathsf{flip}\ d_1)\ \ldots\ (\mathsf{flip}\ d_n) \\
\mathsf{flip}\ {}^{+}\langle c\langle d\rangle\rangle &= {}^{-}\langle c\langle \mathsf{flip}\ d\rangle\rangle \\
\mathsf{flip}\ {}^{-}\langle c\langle d\rangle\rangle &= {}^{+}\langle c\langle \mathsf{flip}\ d\rangle\rangle \\
\mathsf{flip}\ (s_1\ /\ s_2) &= s_2\ /\ s_1
\end{aligned}$$

When the user makes an edit, Pantograph sets up diff boundaries into the program around the location of the edit. The type boundaries introduced in Section 4 have unsurprising typing rules:

$$\frac{\Delta.1 \vdash t : \delta.1}{\Delta.2 \vdash \{t\}^{\downarrow}_{\Delta \vdash \delta} : \delta.2} \qquad \frac{\Delta.2 \vdash t : \delta.2}{\Delta.1 \vdash \{t\}^{\uparrow}_{\Delta \vdash \delta} : \delta.1}$$

There are three ways to make such an edit: inserting a one-hole context, removing one, or directly editing a type annotation.

Inserting a one-hole context. If the context and type of C' change by diffs Δ and δ such that $\Delta \vdash C' : \delta$, then the user may insert it if the term at the cursor has context $\Delta.2$ and type $\delta.1$.

Pantograph then sets up the configuration $C[\{C'[\{t\}^{\downarrow}_{(\text{flip }\Delta)\vdash id}]\}^{\uparrow}_{id\vdash\delta}]$. For example, if the user adds a parameter to a map function as in Figure 7, the system will set up the following configuration:

$$
\begin{array}{l}
\textbf{let}\ \text{map} : \text{List Bool} \to \text{List Bool} = \\
\quad \{\ \lambda f : \text{Bool} \to \text{Bool}. \\
\qquad \{\ \ldots\ \}^{\downarrow}_{+\langle\langle id\rangle, f:\text{Bool}\to\text{Bool}\rangle\vdash id} \\
\quad \}^{\uparrow}_{id\vdash+\langle(Bool\to Bool)\to\langle List\ Bool\to List\ Bool\rangle\rangle} \\
\textbf{in}\ \ldots
\end{array}
$$

Deleting a one-hole context. Given a selection $C_1[C_2[t]]$, the user can delete C_2, as in Figure 9. If $\Delta \vdash C_2 : \delta$, then Pantograph will set up $C_1[\{\{t\}^{\downarrow}_{\Delta\vdash id}\}^{\uparrow}_{id\vdash(\text{flip }\delta)}]$.

Editing a type annotation. When the user edits the annotation of a λ abstraction as in Figure 8 by a type diff δ, Pantograph sets up the configuration

$$\{\lambda x : \delta.2\ .\ \{t\}^{\downarrow}_{id,x:\delta\vdash id}\}^{\uparrow}_{id\vdash\delta\to id}$$

Similarly, when the user edits the annotation of a **let** expression as in Figure 10 by a type diff δ, Pantograph sets up the configuration

$$\textbf{let}\ x : \delta.2 = \{t_1\}^{\downarrow}_{id,x:\delta\vdash\delta}\ \textbf{in}\ \{t_2\}^{\downarrow}_{id,x:\delta\vdash id}$$

5.4 Diff Propagation

Recall the example in Section 4, and in particular the steps 1-3 where a diff boundary passed 'through' an application and a λ abstraction. Making use of our understanding of syntax as trees, we can define the algorithm which performs these steps.

Recall the intrinsic typing rule for applications from earlier (on the right). It has three metavariables: A, B, and Γ. For each application in the program, each of these metavariables is instantiated as some specific tree. When we propagate a diff boundary through a construct of our language, the construct remains the same, but the values of the metavariables from the typing rule change. Therefore, we introduce the notion of a *diff substitution* σ, a mapping from the metavariables in a given typing rule to diffs. Whenever a diff boundary is propagated through a form, our algorithm finds a diff substitution on the metavariables in its typing rule which describes how the instantiations of the metavariables change. We write $\sigma\ s$ to represent the substitution of the metavariables in s by σ; any metavariable not in the domain of σ is mapped to an identity diff.

$$\frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}$$

Using this notion of a diff substiution, we can attempt to propagate diff boundaries through any language construct. Suppose that we have some language construct r with n premises $s_1 \ldots s_n$ and conclusion s which may all refer to a shared set of metavariables in its intrinsic typing rule:

$$\frac{s_1 \ldots s_n}{s}$$

Then, in order to propagate a diff boundary down into such a form, we need only find a diff substitution σ, with domain of the metavariables in s, such that the incoming diff has the form $\sigma\ s$. We may then apply that same substitution to each s_i to get the diff which must be propagated down into the ith child:

$$\{r\ t_1 \ldots t_n\}^{\downarrow}_{\sigma\ s} \quad \rightsquigarrow \quad r\ \{t_1\}^{\downarrow}_{\sigma\ s_1} \ldots \{t_n\}^{\downarrow}_{\sigma\ s_n}$$

We can write a similar rule for dealing with an upwards boundary into a form:

$$r\ t_1 \ldots \{t_i\}^{\uparrow}_{\sigma\ s_i} \ldots t_n \quad \rightsquigarrow \quad \{r\ \{t_1\}^{\downarrow}_{\sigma\ s_1} \ldots t_i \ldots\ \{t_n\}^{\downarrow}_{\sigma\ s_n}\}^{\uparrow}_{\sigma\ s}$$

Specialized to applications in particular, we get the following three propagation rules:

$$\begin{aligned}
\{t_1\}^{\uparrow}_{\Delta\vdash\delta_1\to\delta_2}\ t_2 \quad &\rightsquigarrow \quad \{t_1\ \{t_2\}^{\downarrow}_{\Delta\vdash\delta_1}\}^{\uparrow}_{\Delta\vdash\delta_2} \\
t_1\ \{t_2\}^{\uparrow}_{\Delta\vdash\delta} \quad &\rightsquigarrow \quad \{\{t_1\}^{\downarrow}_{\Delta\vdash\delta\to id}\ t_2\}^{\uparrow}_{\Delta\vdash id} \\
\{t_1\ t_2\}^{\downarrow}_{\Delta\vdash\delta} \quad &\rightsquigarrow \quad \{t_1\}^{\downarrow}_{\Delta\vdash id\to\delta}\ \{t_2\}^{\downarrow}_{\Delta\vdash id}
\end{aligned}$$

The second of these application rules recovers the behavior from step (2) in Section 4.

We can similarly specialize the two generic rules above to any form in Pantograph. However, they are not quite sufficient in the case of *nonlinearities*, or typing rules in which the same metavariable appears multiple times in a premise or the conclusion. For example, consider the intrinsic typing rule for a recursive **let** construct:

$$\frac{\Gamma, x : A \vdash A \qquad \Gamma, x : A \vdash B}{\Gamma \vdash B}$$

The metavariable A appears twice in the first premise. If one were to apply a substitution to that first premise $\sigma(\Gamma, x : A \vdash A)$, then both As would necessarily map to the same diff. However, we would like to have the following propagation rule for recursive **let** constructs, in which a type diff propagates up from the definition, causing a context diff to be sent back down.

$$\textbf{let}\ x : A.1 = \{t_1\}^{\uparrow}_{\Delta, x:id\vdash\delta}\ \textbf{in}\ t_2 \quad \rightsquigarrow \quad \{\textbf{let}\ x : \delta.2 = \{t_1\}^{\downarrow}_{id, x:\delta\vdash id}\ \textbf{in}\ \{t_2\}^{\downarrow}_{\Delta, x:\delta\vdash id}\}^{\uparrow}_{\Delta\vdash id}$$

Our propagation rule from before won't help, because there is no substitution of metavariables such that $\Gamma, x : A \vdash A$ can be made equal to $\Delta, x : id \vdash \delta$, since $id \neq \delta$. The solution will be to focus onto only the part of the incoming diff which is not the identity. In particular, we can decompose the diff into a one-hole context consisting only of identity diffs, and a non-identity diff:

$$\Delta, x : id \vdash \delta = (\Delta, x : id \vdash \langle\rangle)[\delta]$$

We can now finally write the propagation rules in their fully general form. Any language construct r with a typing rule of the form above gives rise to the following propagation rules (where $dom(\sigma)$ is the domain of σ, and $FV(s)$ is the set of metavariables in s):

$$\frac{s = c[s'] \qquad dom(\sigma) = FV(s')}{\{r\ t_1\ \ldots\ t_n\}^{\downarrow}_{(\sigma_{id}\ C)[\sigma\ s']} \quad \rightsquigarrow \quad \{r\ \{t_1\}^{\downarrow}_{\sigma\ s_1}\ \ldots\ \{t_n\}^{\downarrow}_{\sigma\ s_n}\}^{\uparrow}_{(\sigma\ C)[\sigma_{id}\ s']}}\ \textsc{Propagate}\downarrow$$

$$\frac{s_i = C[s_i'] \qquad dom(\sigma) = FV(s_i')}{r\ t_1 \ldots \{t_i\}^{\uparrow}_{(\sigma_{id}\ C)[\sigma\ s_i']} \ldots t_n \quad \rightsquigarrow \quad \{r\ \{t_1\}^{\downarrow}_{\sigma\ s_1}\ \ldots\ \{t_i\}^{\downarrow}_{(\sigma\ C)[\sigma_{id}\ s_i']}\ \ldots\ \{t_n\}^{\downarrow}_{\sigma\ s_n}\}^{\uparrow}_{\sigma\ s}}\ \textsc{Propagate}\uparrow$$

This final version of the propagation rules is capable of propagating diff boundaries through every form in every situation that arises in Pantograph, even in the presence of nonlinearities.

5.5 Pantograph's Automatic Edits

Once diff boundaries are placed after an edit, Pantograph propagates the diff boundaries through the program and alters the program according to a small-step operational semantics, which we describe in this section. We also reproduce all of the rules in a compact table in the extended version of this article [Prinz et al. 2024], Appendix A.

Propagation rules. The first two rules are the Propagate ↓ and Propagate ↑ rules from the previous section which propagate diff boundaries through every form in our language. The only exception is variables, whose propagation rules we list below [1] :

$$\{x\}^{\downarrow}_{C[\Delta,\ x:\delta]\vdash id} \quad \rightsquigarrow \quad \{x\}^{\uparrow}_{id\vdash\delta} \qquad \textsc{Propagate-Var}\downarrow 1$$
$$\{x\}^{\downarrow}_{C[\Delta,\ x:id]\vdash\delta} \quad \rightsquigarrow \quad \{x\}^{\uparrow}_{C[\Delta,\ x:\delta]\vdash id} \qquad \textsc{Propagate-Var}\downarrow 2$$

Alteration rules. Next, Pantograph has a set of rules which enable the automatic edits shown by example in Section 3. First, Pantograph can automatically insert and remove λ abstractions (Figure 8). The following three rules enable this behavior:

$$\{t\}^{\downarrow}_{\Delta\vdash{+}\langle A\to\langle\delta\rangle\rangle} \quad \rightsquigarrow \quad \boldsymbol{\lambda} x : A\ .\ \{t\}^{\downarrow}_{{+}\langle\langle\Delta\rangle,\ x:A\rangle\vdash\delta} \qquad \textsc{Insert-Abs}\ \downarrow$$
$$\{\boldsymbol{\lambda} x : A\ .\ t\}^{\downarrow}_{\Delta\vdash{-}\langle A\to\langle\delta\rangle\rangle} \quad \rightsquigarrow \quad \{t\}^{\downarrow}_{{-}\langle\langle\Delta\rangle,\ x:A\rangle\vdash\delta} \qquad \textsc{Delete-Abs}\ \downarrow$$
$$\boldsymbol{\lambda} x : A\ .\ \{t\}^{\uparrow}_{\Delta,x:id\vdash{+}\langle A\to\langle\delta\rangle\rangle} \quad \rightsquigarrow \quad \{\{t\}^{\downarrow}_{{-}\langle\langle\Delta\rangle,x:A\rangle\vdash id}\}^{\uparrow}_{\Delta\vdash A\to\delta} \qquad \textsc{Delete-Abs}\ \uparrow$$

Pantograph can also automatically insert applications, or replace them with commented ones as the type of a function changes (Figure 8). This is governed by the following rules:

$$\{t\}^{\uparrow}_{\Delta\vdash{+}\langle A\to\langle\delta\rangle\rangle} \quad \rightsquigarrow \quad \{t\ \square_A\}^{\uparrow}_{\Delta\vdash\delta} \qquad \textsc{Insert-App}\ \uparrow$$
$$\{t_1\}^{\uparrow}_{\Delta\vdash{-}\langle A\to\langle\delta\rangle\rangle}\ t_2 \quad \rightsquigarrow \quad \{/\!/\langle\langle t_1\rangle\ \{t_2\}^{\downarrow}_{\Delta\vdash id}\rangle\}^{\uparrow}_{\Delta\vdash\delta} \qquad \textsc{Displace-App}\ \uparrow$$
$$\{t_1\ \square_A\}^{\downarrow}_{\Delta\vdash{+}\langle A\to\langle\delta\rangle\rangle} \quad \rightsquigarrow \quad \{t_1\}^{\downarrow}_{\Delta\vdash id\to\delta} \qquad \textsc{Delete-App}\ \downarrow$$

Finally, Pantograph will automatically replace bound variables with free variables when necessary and vice versa (Figure 9), according to the following rules:

$$\{x\}^{\downarrow}_{C[{-}\langle\langle\Delta\rangle,\ x:T\rangle]\vdash id} \quad \rightsquigarrow \quad /\!/x_T \qquad \textsc{Local-To-Free}$$
$$\{/\!/x_T\}^{\downarrow}_{C[{+}\langle\langle\Delta\rangle,\ x:T\rangle]\vdash T} \quad \rightsquigarrow \quad x \qquad \textsc{Free-To-Local}$$

All together, these eight rules describe how Pantograph automatically changes a program in response to a user's edit. The alteration rules take precedence over the propagation rules when both can apply. Additionally, Delete-App ↓ takes precedence over Insert-Abs ↓.

Diff Boundary Rules. Finally, Pantograph has eight rules used to deal with special cases involving diff boundaries themselves. First, two rules eliminate diff boundaries containing only identity diffs.

$$\{t\}^{\downarrow}_{id\vdash id} \quad \rightsquigarrow \quad t \qquad \textsc{Identity}\ \downarrow$$
$$\{t\}^{\uparrow}_{id\vdash id} \quad \rightsquigarrow \quad t \qquad \textsc{Identity}\ \uparrow$$

Next, two rules enable diff boundaries to cross each other when the meet, so long as they don't both modify the same side of the typing judgement:

$$\{\{t\}^{\uparrow}_{id\vdash\delta}\}^{\downarrow}_{\Delta\vdash id} \quad \rightsquigarrow \quad \{\{t\}^{\downarrow}_{\Delta\vdash id}\}^{\uparrow}_{id\vdash\delta} \qquad \textsc{Interchange}\ \ 1$$
$$\{\{t\}^{\uparrow}_{\Delta\vdash id}\}^{\downarrow}_{id\vdash T} \quad \rightsquigarrow \quad \{\{t\}^{\downarrow}_{id\vdash T}\}^{\uparrow}_{\Delta\vdash id} \qquad \textsc{Interchange}\ \ 2$$

[1] In our implementation, using a typed representation of de Bruijn indices [de Bruijn 1972] the two variable propagation rules are also actually derived from the generic propagation rules as well.

Two more rules stop type diffs from propagating in or out of function applications. For example, in Figure 10, a type diff is propagated up from the f variable. The following two rules stop it around the function application, keeping the effect of the edit local to the definition it was made on and giving the user the choice of how to proceed. We define a *neutral form* as either a variable, a built-in function (such as cons), or a neutral form applied to an argument. These rules take precedence over the Propagate but not the alteration rules.

$$\left.\begin{array}{rcl} \{t\}^{\downarrow}_{\Delta\vdash\delta} & \rightsquigarrow & \{\{t\}^{\downarrow}_{\Delta\vdash id}\}^{!}_{\delta.1/\delta.2} \\ C[\{t\}^{\uparrow}_{\Delta\vdash\delta}] & \rightsquigarrow & C[\{\{t\}^{!}_{\delta.2/\delta.1}\}^{\uparrow}_{\Delta\vdash id}] \end{array}\right\} \text{ if } t \text{ is a neutral form, and } C \neq \langle\rangle\ t_2 \qquad \begin{array}{ll} \textsc{Neutral-Error} & \downarrow \\ \textsc{Neutral-Error} & \uparrow \end{array}$$

Finally, if no other rules apply, diff boundaries are converted into type error boundaries in the program:

$$\left.\begin{array}{rcl} \{t\}^{\downarrow}_{\Delta\vdash\delta} & \rightsquigarrow & \{\{t\}^{\downarrow}_{\Delta\vdash id}\}^{!}_{\delta.1/\delta.2} \\ \{t\}^{\uparrow}_{\Delta\vdash\delta} & \rightsquigarrow & \{\{t\}^{!}_{\delta.2/\delta.1}\}^{\uparrow}_{\Delta\vdash id} \end{array}\right\} \text{ if no other rules apply and } \delta \neq id \qquad \begin{array}{ll} \textsc{Fallthrough-Error} & \downarrow \\ \textsc{Fallthrough-Error} & \uparrow \end{array}$$

5.6 Metatheory of the Propagation Rules

The system we have presented in this paper is quite intricate. In this section, we will state and sketch the proofs of the correctness properties of our automatic edit system — full proofs can be found in the extended version of this article [Prinz et al. 2024], Appendix B.

Theorem 3 (Progress). *If t resulting from an edit has a diff boundary (other than an up boundary at the top), then for some t', $t \rightsquigarrow t'$*

If at the end of propagation the program has the form $\{t\}^{\uparrow}_{\Delta\vdash\delta}$, we prove an invariant on the contexts in up boundaries that shows that $\Delta = id$. In this case, Pantograph removes the boundary and the program has a new type.

Otherwise, given a boundary with diff $\Delta \vdash \delta$, if $\delta \neq id$, then by definition either the Fallthrough-Error rule or some other rule applies. On the other hand, if $\delta = id$, then we must show that a Propagate rule always applies. For most of the typing rules in Figure 6, this is straightforward, as the context in most conclusions and premises is a single metavariable Γ with which any diff can unify. However, a few typing rules have a non-trivial context, such as that of a λ abstraction. In the full proof we use our invariant to deal with these cases.

Theorem 4 (Type Preservation). *For any $t \rightsquigarrow t'$, if $\Gamma \vdash t : T$ then $\Gamma \vdash t' : T$*

All of the diff boundary rules preserve the type of the term, as well as the type of sub-terms. The proof proceeds by case analysis over the small-step propagation rules. Full details can be found in the full proof.

Theorem 5 (Termination). *For any program with diff boundaries resulting from an edit in Pantograph, there is no infinite sequence of step rules that can be applied.*

Proving termination of the boundary propagation is more involved. Taking a look at the rules above, we can observe that up boundaries can turn into down boundaries, but with the exception of the Variable* rules, no down boundary can ever turn into an up boundary. Also, with the exception of the Insert-* rules, the upwards boundaries make progress to the top of the program and the downwards boundaries make progress to the leaves. Therefore, the path of a boundary through the program will be to first go up, and then go down, and then disappear. We formalize this intuition in the full proof, providing a decreasing metric over terms with diff boundaries.

Nondeterminism. In most cases, there is only one rule that can apply to a given diff boundary. However, the order in which multiple boundaries are propagated is left unspecified, and in some situations it is not obvious that the order does not matter. For example, the following situation arises during diff propagation at the second call to f in Figure 8:

$$\{\{f\}^{\uparrow}_{+\langle \mathsf{Int}\to\langle \mathsf{Bool}\to\mathsf{Bool}\rangle\rangle}\}^{\downarrow}_{+\langle \mathsf{Int}\to\langle \mathsf{Bool}\to\mathsf{Bool}\rangle\rangle}$$

Either the down boundary can be stepped first, applying Insert-Abs ↓ and then Delete-Abs ↑, or the up boundary can be stepped first, applying Insert-App ↑ and then Delete-App ↓. But either way, the result is the same term f.

More generally, we have proven the following confluence result. Using $t_1 \rightsquigarrow^* t_2$ to mean that t_1 steps to t_2 after zero or more steps,

Theorem 6 (Confluence). *For any program with diff boundaries t resulting from an edit in Pantograph, if $t \rightsquigarrow^* t_1$ and $t \rightsquigarrow^* t_2$, then there exists t' such that $t_1 \rightsquigarrow^* t'$ and $t_2 \rightsquigarrow^* t'$*

The core idea of the confluence proof is to consider all of the pairs of rules which could step the same term, and show that the same term can be reached regardless of which rule was chosen first. The difficult cases are those like the example above, where an up diff boundary is inside a down diff boundary, because in these cases two different rules can alter the same part of the program. However, such situations are rare. We again observe that up boundaries can only turn to down boundaries at the Propagate-Variable rules. Therefore, with an exception in neutral forms, the property that no up boundary is a descendant of a down boundary is preserved by the rules. In the full proof, we formalize this intuition by proving two invariants that are preserved by all of Pantograph's rules. These invariants reduce the number of possible cases to only eight, all of which we discuss in the full proof.

6 User Study

To convincingly establish the effectiveness of a particular editor, one would need to undertake a thorough empirical comparison of its effectiveness compared to other editors—a herculean task way beyond the scope of this work. Instead of comparing our editor to other state-of-the-art structure editors (such as Hazel or MPS), in this section we empirically investigate the feasibility of our approach—whether programmers can learn to think and program in the paradigm presented in this paper. Furthermore, we investigate qualitatively which aspects of Pantograph worked well or badly, and compare and contrast how participants use Pantograph and the text editor. We describe a user study in which participants were first given a short introductory tutorial on using Pantograph, before being asked to complete a series of basic programming tasks using either Pantograph or a text editor.

Participants. We sought participants with prior experience with strongly-typed and functional programming languages, so we recruited 13 students (11 male, 1 female, 1 unstated; ages 19-25) from CMSC 433, an upper level functional programming course at the University of Maryland, and offered participants a $30 compensation. Recruitment materials included a link to Pantograph, and one participant used it recreationally before the study.

Editors and Tasks. We designed a version of Pantograph [2] and a text editing environment, both using a simple ML-style programming language, for the study. The text editor was based on Monaco [Microsoft 2024], the library behind the popular editor Visual Studio Code [Microsoft

[2]Available here.

2015]. We augmented Monaco with syntax highlighting, automatic type-checking, and inline typing diagnostics. Further, we designed software to conduct the study, available here.

Each participant solved the same sequence of 12 tasks (T1-T12). The tasks were designed in six pairs and each participant solved one task from each pair in each editor. Four tasks (T1, T2, T7, T8) were taken from a prior study on Tylr [Moon et al. 2022], asking participants to transcribe a given program and then edit it into a second given one. Another pair of tasks (T5, T11) was inspired by a prior study of MPS [Berger et al. 2016], in which participants were instructed to fix definitions of a common mathematical law given a correct version to copy. Because we also wanted to test each participant's ability to program using our typed editing paradigm rather than merely copy given code, we designed six programming tasks: two tasks to implement a function using a given fold function (T3, T9), two tasks to generalize a given function implementation in a specified way (T4, T10), and two tasks to implement a simple recursive function over a list (T6, T12).

Procedure. The participants were given a 1-hour tutorial on Pantograph consisting of interactive examples and exercise which explained typed tree editing, zipper editing and selections, and type boundaries. During the tutorial, we actively answered questions and assisted participants. Additionally, each participant was given 5 minutes to familiarize themselves with the text editor.

After the tutorial, the participants were randomly divided into two groups. Group A solved the first six tasks in Pantograph and the last six tasks in the text editor, while group B used the editors in the reverse order. While solving tasks, participants were allowed to ask questions about the language or editor but not about how to solve the task. When a participant's code passed a task's test suite, the software indicated that they could continue to the next task. Participants were informed that they should skip any task that they thought they couldn't finish or was taking them too long.

6.1 Quantitative Analysis

The user study yielded roughly 10 hours of screen recordings. In all, 156 tasks were attempted by the participants. Two participants (one in Group A and one in Group B) skipped at least six tasks, including all of the programming tasks in both editors—we chose to omit these two participants' data from the following analysis, although we discuss this issue further in the limitations section. Further, we discarded the data points (4 in Pantograph and 2 in text) where our test suite mistakenly indicated to a participant that their solution was correct when it was actually incorrect. This left us with 124 data points on tasks, 121 of which were solved and 3 of which were skipped.

Figure 12 shows the duration of each task for each editor. Overall, Pantograph was slower than the text editor for most tasks. It was 1.4× slower on average overall, measured as the geometric mean of the ratio between the average duration of each editor on each task.

In order to view these task durations by participant, we need a way to compare times across different tasks. We define a participant's *relative task time* as the ratio of their time to the average time taken on that task. We plotted the average relative task time in each editor for each participant in Figure 13. To account for skipped tasks in this calculation, we used the maximum time taken for that task by any participant. Some of the participants performed similarly in both editors, while some performed significantly more slowly in Pantograph than the text editor. Overall, most of the participants in the study were able to effectively solve the tasks using Pantograph.

6.2 Qualitative Analysis

While the quantitative data demonstrate that the participants were generally able to solve tasks in Pantograph, albeit somewhat slower than in the text editor, a qualitative analysis is necessary to answer more specific questions. In order to discover which aspects of Pantograph helped or

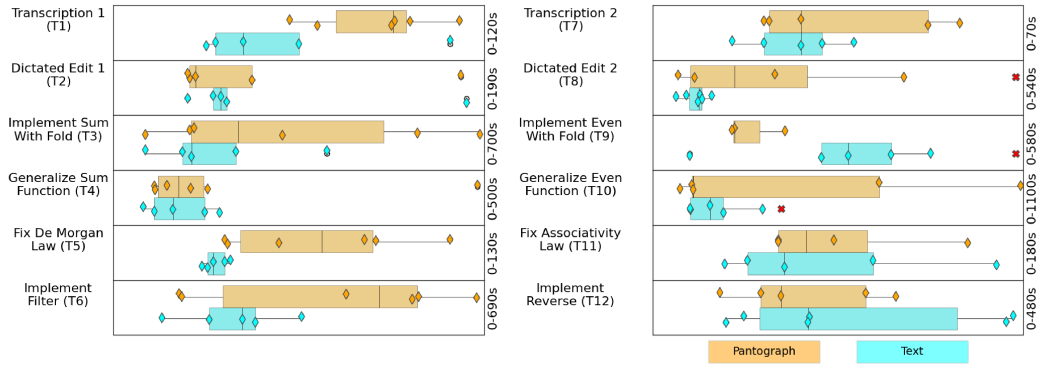


Fig. 12. Task durations (in seconds) split by editor. For each task, Pantograph times are on the top in orange, and text times are on the bottom in blue. Tasks that participants chose to skip are marked as a red 'X'.

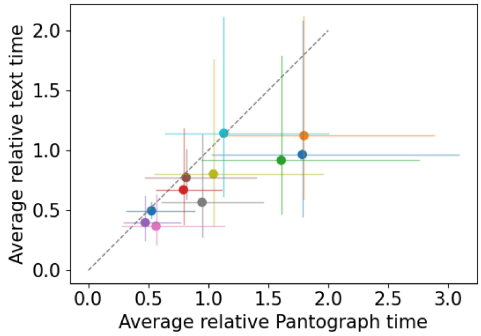


Fig. 13. Geometric mean and variance of relative task times for each participant (denoted as a dot), and for each editor. The dashed line shows parity between the editors — participants below the line solved tasks slower in Pantograph then the text editor. One standard deviation is shown as horizontal and vertical lines.

hindered the participants, and to compare the way that participants solved problems in the two editors, we performed a reflexive thematic analysis following the six step process described in Braun and Clarke [2006].

First, we coded the screen recordings, over several viewings, by noting common behavioral patterns as participants solved tasks. While in many cases we were unable to interpret what a participant may have been thinking from a screen recording, we focused on patterns that could be identified unambiguously. Next, we organized the codes into themes. Finally, we developed these themes into the final analysis that is the rest of this section. We emphasize that reflexive thematic analysis is inherently subjective [Braun and Clarke 2019], particularly in our choices of which codes and themes to focus on.

Attempts to use Pantograph's interface like a text editor. There was a wide variety in participants' understanding of Pantograph's basic interface. One common theme that emerged in some participants were attempted actions with Pantograph that would make sense in a text editor but do not work in Pantograph's interface. Two codes in particular seemed to unambiguously show this principle. The first is that some participants tried to input an entire expression consisting of

multiple forms into a query such as `1 + 1`, despite that in Pantograph forms can only be input one at a time and so these queries are not valid. The second is that some participants, in a situation where the cursor was around a term that they intended to apply to an argument, attempted to immediately query the argument without first inserting an application form. These particular patterns only occurred for the 4 participants who had the longest relative Pantograph times, and one participant who skipped all of the coding problems.

Conversely, one code represented an understanding that Pantograph should not be used like a text editor. Four participants opted to skip writing type annotations while transcribing programs in T1 and T7, and instead allow Pantograph to infer them automatically — a behavior with no analog in text editing. These were the 4 participants with the fastest relative Pantograph times.

Diff propagation working well. Two tasks (T4 and T10) had a simple intended solution in Pantograph which would make use of diff propagation to add an argument to a function. All but two participants made use of the diff propagation to solve the problem - one manually rewrote a function call instead, and the other skipped the problem. Additionally, while no other tasks necessarily required diff propagation, most participants made use of it to at some point to fix a mistake.

Diff propagation getting in the way. On the other hand, there were two common patterns in which diff propagation obstructed a participant from programming in the way they wanted to. These were not characteristic of only participants who had longer relative Pantograph times, but rather there was no clear correlation - we believe these to represent flaws with Pantograph. Most of these cases involved creation of type boundaries.

The first pattern was that participants accidentally changed the type of a variable by making an edit to one of its arguments. We can be certain that these alterations were unintentional because they occurred in problems where a type annotation was given and should not be changed. Most participants (7 of 13) had this problem at some point. Typically the edits were locally sensible, but the edit unintentionally caused a type diff to propagate to a variable definition.

The second pattern of confusion arose when a participant wanted to insert the child of a form at their cursor *before* inserting the parent form. In some cases Pantograph supports with, like wrapping $\langle\rangle + \square$ around 1. But in particular on task T12, participants needed to input lambda abstractions surrounding a match expression. A few participants decided to first input the match before the lambda abstractions (1 in text and 3 in Pantograph), even though directly inputting the lambda abstractions afterwards alters the type of the surrounding function by diff propagation.

Syntax Errors. In both editors, participants tried to use invalid syntax. In the text editor the invalid syntax was marked by the editor with a syntax error, and in Pantograph the query is invalid and cannot be inserted into the program in the first place. Syntax errors were much more common in the text editor — all but 1 participant input syntax errors in text at some point, while only 3 participants ever input invalid syntax into a Pantograph query, excepting the earlier mentioned pattern of inputting multiple forms in one query. One benefit of a structure editor like Pantograph is that it requires less memorization of syntax, although it is possible that more practice with the text editor could have prevented many of the syntax errors made in text.

Type errors. Again in both editors, participants tried to input ill-typed forms—in the text editor this leads to a type error after the fact, while in Pantograph either a query is not allowed, or a type boundary appears elsewhere in the program. Type errors appeared in 24 attempted tasks in the text editor, while participants attempted to input ill-typed queries in 14 attempted tasks in Pantograph. Visible types on holes in Pantograph may have helped participants avoid trying to directly input an ill-typed term more often than in the text editor, although as mentioned above there were also

many situations where type boundaries appeared due to diff propagation, not counted here as ill-typed queries.

Selections. Two tasks (T2 and T11) had a simple intended solution involving a selection. Most participants used selections, except a few who rewrote the expressions entirely (1 time in Pantograph and 6 times in text). Additionally, almost all edits made using text selections (excepting small selections in a single word) corresponded to structured edits: 5 selection based text edits were directly equivalent to a zipper edit with a one-hole context; 7 were not quite equivalent to a zipper edit but were immediately followed by a small additional edit that made it equivalent to one; 5 selection based text edits amounted to swapping two entire terms; and 2 more amounted to moving a term. On the other hand, only one selection based text edit did not correspond to a structured edit - a participant wrote part of an expression, and then deleted it with a selection. Finally, in Pantograph, participants were often unsure about how to make a selection, as evidenced by behavior making several experimental selections before deciding what to do next—which occurred 15 times during tasks attempted in Pantograph, compared to only 6 times in those attempted in the text editor.

6.3 Limitations

As discussed above, no other structure editor was used in this study, so our results offer little insight into how our system compares to other structure editors. Additionally, while the tasks in our study were intended to represent realistic programming scenarios, the results may not generalize to real world programming in a fully developed system. Finally, the participants had no experience with Pantograph before the study (with one exception) but had an hour to practice with it during the study; at the same time, they had plenty of prior experience using a text editor for similar languages, but only five minutes to practice with it during the study. It is not clear how the results might change if participants had the same amount of practice in both editors.

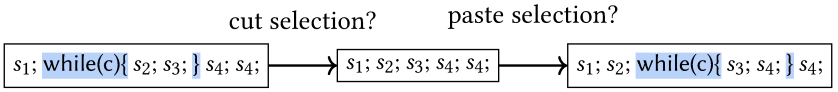
Unlike prior studies on structure editors such as Berger et al. [2016]; Moon et al. [2022, 2023] which focus on transcription tasks, we also included programming tasks. Although we intended the tasks to be easy enough to complete in the time allotted for the study, two participants were still not able to complete any of these coding tasks in either editor, getting stuck on an early problem and then quickly skipping later problems. As we had not planned task time limits beforehand we decided on our problem skipping protocol during the study, and it is possible that this design flaw influenced the participant's decisions to skip problems.

Finally, there was a large potential for bias and non-reproducibility: we facilitated the user study in person, and the participants were aware that the we created Pantograph, which has been shown to cause bias [Dell et al. 2012]. Additionally, the Pantograph tutorial involved us actively teaching participants in a one-on-one manner, which could lead to different results with different teachers. Our decision to allow participants to ask questions about the editor and language while solving the tasks may have additionally affected the results of the study, as our decisions in answering these questions were subjective.

Taking into account all of these limitations, we re-emphasize that our goal was to demonstrate our approach's feasibility rather than its effectiveness relative to state of the art structure editors and explore the advantages and disadvantages of Pantograph compared to a traditional text editor. Through this user study, we learned that students were able to learn to use Pantograph relatively effectively in a short amount of time, but we also discovered many opportunities for improvement in our ongoing efforts to develop an intrinsically typed structure editor. We leave a thorough empirical evaluation of different structure editors for future work.

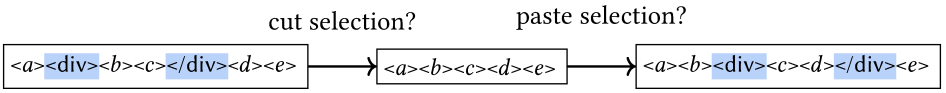
7 Discussion

Limitations of Zipper Editing. In Section 2, we showed that in our simply-typed ML-like language, most common edits take the form of adding or removing a one-hole context in the program. However, in some other settings this form of edit doesn't turn out to be as useful. For example, consider the following edit to an imperative c-style program:



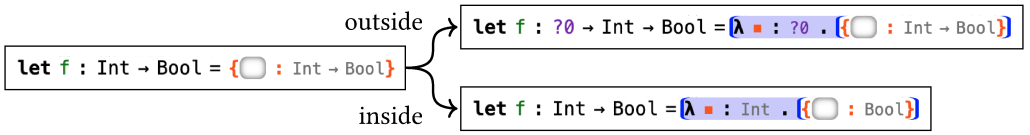
Visually, this sequence of edits appears like a zipper cut and paste. However, it doesn't seem possible to understand it mathematically as a zipper selection. If ';' is a binary operator, then we may be able to understand the initial cut as the deletion of a one-hole context. However, this would leave the ';' operators associated in the wrong way for the paste to be possible.

This situation is not limited to imperative programming. For instance, consider a similar example that might occur in a markup language like HTML:

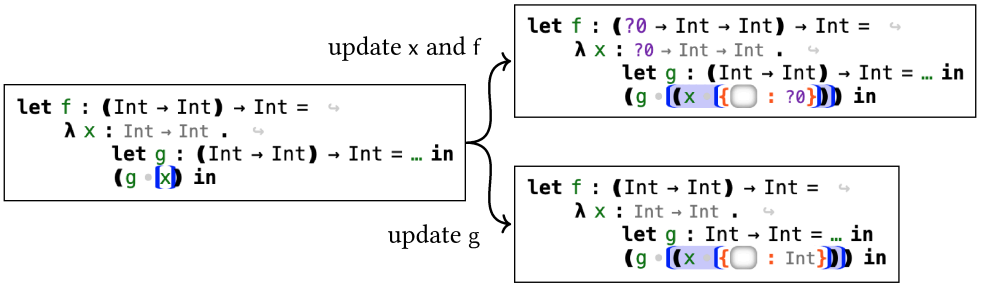


Limitations of Typed Editing. Our goal in creating Pantograph was to build an editor which only allows edits which are locally well-typed, and yet the user doesn't need to edit typing annotations. Instead, the user may edit terms in locally sensible ways, and the corresponding annotations are automatically updated. However, we found that certain trade-offs are required.

For example, consider the following program in which the user inserts a lambda at a hole. The hole is already at a function type, so there are two options for what might happen: the lambda can either fill the hole, therefore leaving the type of the term unchanged. Or, the lambda can go around the hole, thus adding another argument to the type of the term.



As another example, consider the following program in which the user inserts an application around x. Here, x is already a function so there are two possible outcomes. Either x gets a new argument, leaving the type of g unchanged. Or, x is applied to its existing argument, changing the type of g.



Of course, many less rigid systems exist which re-typecheck the program after an edit. Our typed refactoring system could be integrated with such tools while retaining many of its benefits. Instead, we opted to explore a more opinionated point in the design space of editors. In intrinsically typed editors, as the above examples show, either the programmer must sometimes be required to edit the type annotations directly, or the programmer must sometimes be required to input extra information with an edit which can be used to distinguish between the multiple possible outcomes.

In Pantograph, we chose a mixture of the two options. We solve the first example in the user interface by making two distinct cursor positions: inside a hole, and around a hole. Inputting the λ expression inside the hole gives a function of the existing type, while wrapping it around the hole adds a new parameter. We solve the second example by requiring the user to edit the type annotation in order to achieve the first outcome.

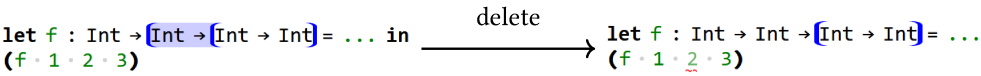
8 Related Work

Traditional Tree-Based Structure Editors. Most existing structure editors share a common core design. As described in the introduction, these editors represent the program as a tree with holes. The user may insert forms into holes, delete nodes, and copy/paste entire nodes. Often, there are additional list structures in which sub-lists may be manipulated.

A wide variety of tree-based structure editors have been developed by both academia and industry with a wide variety of design goals. Some are designed for advanced type theories, such as ALF [Pareto 1995] for Martin-Löf type theory, and Alfa [Allgren 1996] for the Agda dependently-typed programming language. Others represent the tree as nested blocks, such as Scratch [Maloney et al. 2010], a well-known structure editor designed for teaching programming to children. Others have been designed for general purpose programming, such as MPS [Voelter 2013] which allows the user to edit languaged defined by custom DSLs, or Lamdu [Lotem and Chuchem 2015] which edits a Haskell-like functional language.

While these editors offer a variety of innovative designs, two problems often arise in tree editors. First, as we described in the introduction, it is difficult to edit code by only moving entire subtrees, a problem often referred to as viscosity [Blackwell and Green 2003]. Second, while the approach straightforwardly maintains well-typedness for a simple type system like that in Scratch, most syntactic structure editors with a more complex type system resort to type checkers which give errors in the same way as on a text editor, leading to semantically meaningless partial programs.

While many tree editors re-typecheck the program after each edit, others are intrinsically typed, like Pantograph. Intrinsically typed editors place errors *during* an edit, giving them access to more information than a checking algorithm which runs *after* the edit. For example, suppose that a programmer deletes an argument from a function which inputs three integers in Pantograph:



Pantograph is able to place a commented application at the corresponding argument. In contrast, a type checker only has access to the (untyped) program state after the edit, and so can not distinguish which argument of the function was deleted.

Hazel. The closest related work that addresses the viscosity problem—and one of the most successful structure editors in recent history in general, is Hazel [Future of Programming Lab 2017], a structured editing system with a typed language and an edit semantics.

Early iterations of the Hazel structure editor ensure that every state is statically meaningful, by defining statics [Omar et al. 2017] and dynamics [Omar et al. 2019] for incomplete programs in a tree-based structure editor. In particular, Hazel leverages a gradual type system [Siek and Taha

2006; Siek et al. 2015], which allows ill-typed (according to a strict type system) programs to be considered well-typed by converting the type errors into dynamic type checks. It can also run programs with holes, giving users feedback about partial programs.

To address viscosity, *Tylr* [Moon et al. 2022, 2023] introduces gradual structure editing, allowing the user to locally break the tree structure of the program and get hints about how to fix the ill-formed fragment. Tylr introduced a concept of "structural obligations" that, given an ill-formed program, encode where certain syntactic delimiters must be inserted in order to yield a well-formed program. In essence, gradual structured editing allows the user to edit their program as if it was text, get structured support for the fragment of the program that is well-structured, and hints about how to fix the fragment that is ill-formed. The authors conducted two user studies, which demonstrate that this approach can improve fluidity in comparison to a more traditional rigidly tree-structured editor.

Moon et al. [2023] also identify several specific barriers to fluidity with strict tree editors, including *multiplicity*, or the lack of the ability to place terms temporarily anywhere in the program during edits. While our zipper editing paradigm improves the fluidity of a tree editor, multiplicity remains a problem in Pantograph — in particular, swapping two terms is difficult.

Recent versions of Hazel incorporate Tylr, replacing its older tree-based interface. Furthermore, they re-check the program after each edit using a novel algorithm. Chen et al. [2024]; Zhao et al. [2024] replace a standard type checking algorithm with a marking algorithm, which inputs untyped preterms and marks them with various marks indicating typing problems, including ones similar to our type error boundaries. The algorithm is superior at placing type errors to many prior type checking algorithms because of its focus on neutrality, or not making unwarranted assumptions about error placements. In addition, the algorithm can assign static meaning to many programs on which a traditional type checker would give up, and can give suggestions for automatic fixes for type errors. However, as we discussed earlier, approaches like this which re-check the program after an edit contrast with Pantograph's approach of placing errors during an edit.

Other Editors. Other ways of manipulating structured syntax have been devised. Paredit [Paredit Team 2022] is a tree editor which provides structured operations on S-expressions [McCarthy 1960]. Beyond tree editors, graph editors, like the system design environment LabVIEW [Bitter et al. 2006], represent the program as nodes which the programmer can connect with edges.

In order to compactly represent and contrast the diverse designs of existing structure editors, we chose 8 representative editors and compare them on four criteria. The first two criteria pertain to how strictly structured the editor is. The last two criteria pertain to fluidity; they are not intended as a benchmark of fluidity, but rather are chosen to highlight the differences between different approaches.

Editor	All states are well-formed trees	Intrinsically typed edits or re-check	Swap terms conveniently (multiplicity)	Can re-order e.g. $C[((x*2)+1)]$ and $C[((x+1)*2)]$
Text	No	Re-check	Yes	Yes, 1 cut-paste
Paredit	Yes	(Untyped)	Yes	Yes, many actions
MPS	Yes	Re-check	No	No
Scratch	Yes	Intrinsic	Yes	Yes, 6 moves
Early Hazel	Yes	Intrinsic	No	No
Tylr	Partially	(Untyped)	Yes	Yes, 2 cut-paste
Current Hazel	Partially	Re-check	Yes	Yes, 2 cut-paste
Pantograph	Yes	Intrinsic	No	Yes, 1 cut-paste

Other related works. Many approaches have been developed for locating (and fixing) errors in existing programs, contrasting with Pantographs intrinsically typed approach as we discussed earlier. There is a large body of work on improving a type checker's ability to locate type errors, including Wand [1986], which improves on a Hindley Millner type inference algorithm to track the reasons for failed unification; and Tip and Dinesh [2001] and Schilling [2012], which report a slice of all program locations contributing to an error. Bidirectional type checking [Dunfield and Krishnaswami 2021], prescribes specific locations in programs where types are inferred or checked.

Automated program repair [Le Goues et al. 2021] uses various techniques to automatically find and fix bugs in a program. For example, GenProg [Le Goues et al. 2012] uses a genetic algorithm to randomly vary a program until it passes test cases. In contrast to Pantograph, these algorithms operate on existing code rather than as part of the editing process.

Ornaments [McBride 2010] encode relationships between data types. Williams and Rémy [2017] use ornaments to facilitate automatic refactoring of code. Ornaments can encode more possible relations between types than our diffs (although only at data types), but require a user-written definition rather than being derived from the shape of any edit as diffs are in Pantograph.

9 Conclusion

In this paper we presented a zipper editing scheme which generalizes text editing in a structured setting, allowing users to edit a program while maintaining syntactic well-formedness. In this way, zipper editing is conceptually analogous to text editing while providing the advantages of structure editing. We also introduced a type editing system which extends zipper editing. Without requiring the user to learn any new interactions beyond zipper editing, it allows edits to maintain the well-typedness of the program. The user may make any edit which locally can be reconciled to be well-typed, and our diff propagation system will keep the rest of the program aligned to whatever changes were made.

The modern text editing interface has stood as the dominant code editing interface for decades, and programmers have correspondingly internalized its concepts as second nature. This familiarity creates a barrier to entry for alternative editing systems that ask users to learn new concepts. However, the widespread adoption of complex IDEs and editor plugins suggests a strong desire for functionality that is difficult to integrate with text editing.

Since programs are fundamentally tree-structured, structure editing offers innate advantages conceptually organizing basic and advanced edits alike. In the introduction, we posed the possibility of designing a general fluid structure editor that never resorts to operating over untyped or ill-formed syntax. Our contributions with Pantograph show that this goal is achievable, by placing a powerful typed editing system into an interface as simple, expressive, as language-generic as, and in direct correspondence with, those in text editing.

Data Availability Statement

An extended version of this paper is available [Prinz et al. 2024], containing full proofs of the theorems in Section 5.

An artifact [Blanchette and Prinz 2024] containing the source code for Pantograph, as well as the web application for Pantograph and the user study, is available. The web application can also be found at https://pantographeditor.github.io/Pantograph/.

Acknowledgments

We thank Sankha Narayan Guria, David Thrane Christiansen, our colleagues at the Programming Languages Research group at the University of Maryland, and the anonymous reviewers for their helpful comments.

We also thank all those who helped us test the user interface and the user study participants.

This work was supported by NSF award #2107206, Efficient and Trustworthy Proof Engineering (any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF).

References

Thomas Allgren. 1996. Alfa. https://cth.altocumulus.org/ hallgren/Alfa/index.html.

Thorsten Berger, Markus Völter, Hans Peter Jensen, Taweesap Dangprasert, and Janet Siegmund. 2016. Efficiency of projectional editing: a controlled experiment. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 763–774. https://doi.org/10.1145/2950290.2950315

Rick Bitter, Taqi Mohiuddin, and Matt Nawrocki. 2006. *LabVIEW: Advanced programming techniques*. Crc Press.

Alan Blackwell and Thomas Green. 2003. CHAPTER 5 - Notational Systems—The Cognitive Dimensions of Notations Framework. In *HCI Models, Theories, and Frameworks*, John M. Carroll (Ed.). Morgan Kaufmann, San Francisco, 103–133. https://doi.org/10.1016/B978-155860808-5/50005-8

Henry Blanchette and Jacob Prinz. 2024. Pantograph Implementation. https://doi.org/10.5281/zenodo.14199877

Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative Research in Psychology* 3, 2 (2006), 77–101. https://doi.org/10.1191/1478088706qp063oa arXiv:https://www.tandfonline.com/doi/pdf/10.1191/1478088706qp063oa

Virginia Braun and Victoria Clarke. 2019. Reflecting on reflexive thematic analysis. *Qualitative Research in Sport, Exercise and Health* 11, 4 (2019), 589–597. https://doi.org/10.1080/2159676X.2019.1628806 arXiv:https://doi.org/10.1080/2159676X.2019.1628806

Adam Chen, Thomas Porter, and Cyrus Omar. 2024. Polymorphism with Typed Holes. *TFP 2024*.

N.G de Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)* 75, 5 (1972), 381–392. https://doi.org/10.1016/1385-7258(72)90034-0

Nicola Dell, Vidya Vaidyanathan, Indrani Medhi, Edward Cutrell, and William Thies. 2012. "Yours is better!": participant response bias in HCI. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '12)*. ACM. https://doi.org/10.1145/2207676.2208589

Jana Dunfield and Neel Krishnaswami. 2021. Bidirectional Typing. *ACM Comput. Surv.* 54, 5, Article 98 (may 2021), 38 pages. https://doi.org/10.1145/3450952

Future of Programming Lab. 2017. Hazel, a live functional programming environment featuring typed holes. https://hazel.org

Gérard Huet. 1997. The Zipper. *J. Funct. Program.* 7, 5 (sep 1997), 549–554. https://doi.org/10.1017/S0956796897002864

Amy J. Ko, Htet Htet Aung, and Brad A. Myers. 2005. Design requirements for more flexible structured editors from a study of programmers' text editing. In *CHI '05 Extended Abstracts on Human Factors in Computing Systems (CHI05)*. ACM. https://doi.org/10.1145/1056808.1056965

Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (Jan. 2012), 54–72. https://doi.org/10.1109/tse.2011.104

C. Le Goues, M. Pradel, A. Roychoudhury, and S. Chandra. 2021. Automatic Program Repair. *IEEE Software* 38, 04 (jul 2021), 22–27. https://doi.org/10.1109/MS.2021.3072577

Eyal Lotem and Yair Chuchem. 2015. Project Lamdu. https://www.lamdu.org/.

John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The Scratch Programming Language and Environment. *ACM Trans. Comput. Educ.* 10, 4, Article 16 (nov 2010), 15 pages. https://doi.org/10.1145/1868358.1868363

Conor McBride. 2001. The Derivative of a Regular Type is its Type of One-Hole Contexts. https://api.semanticscholar.org/CorpusID:15235160

Conor McBride. 2010. Ornamental algebras, algebraic ornaments. (01 2010). https://personal.cis.strath.ac.uk/conor.mcbride/pub/OAAO/LitOrn.pdf

John McCarthy. 1960. Recursive functions of symbolic expressions and their computation by machine, Part I. *Commun. ACM* 3, 4 (apr 1960), 184–195. https://doi.org/10.1145/367177.367199

Microsoft. 2015. *Visual Studio Code*. https://github.com/microsoft/vscode

Microsoft. 2024. *monaco-editor: the fully featured code editor from VS Code*. https://www.npmjs.com/package/monaco-editor NPM package version 0.47.0.

David Moon, Andrew Blinn, and Cyrus Omar. 2022. tylr: a tiny tile-based structure editor. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Type-Driven Development* (Ljubljana, Slovenia) *(TyDe 2022)*. Association for Computing Machinery, New York, NY, USA, 28–37. https://doi.org/10.1145/3546196.3550164

David Moon, Andrew Blinn, and Cyrus Omar. 2023. Gradual Structure Editing with Obligations. In *2023 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE. https://doi.org/10.1109/vl-hcc57772.2023.00016

Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live functional programming with typed holes. *Proc. ACM Program. Lang.* 3, POPL, Article 14 (jan 2019), 32 pages. https://doi.org/10.1145/3290327

Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. 2017. Hazelnut: a bidirectionally typed structure editor calculus. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 86–99. https://doi.org/10.1145/3009837.3009900

Paredit Team. 2022. Paredit website. https://paredit.org/

Lena Pareto. 1995. The Implementation of ALF - a Proof Editor based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution. (03 1995).

Jacob Prinz, Henry Blanchette, and Leonidas Lampropoulos. 2024. Pantograph: A Fluid and Typed Structure Editor. (2024). https://doi.org/10.48550/arXiv.2411.16571 arXiv:arXiv:2411.16571

Thomas Schilling. 2012. *Constraint-Free Type Error Slicing*. Springer Berlin Heidelberg, 1–16. https://doi.org/10.1007/978-3-642-32037-8_1

Jeremy Siek and Walid Taha. 2006. Gradual typing for functional languages. *Scheme and Functional Programming*.

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPICS.SNAPL.2015.274

F. Tip and T. B. Dinesh. 2001. A slicing-based approach for locating type errors. *ACM Transactions on Software Engineering and Methodology* 10, 1 (Jan. 2001), 5–55. https://doi.org/10.1145/366378.366379

Markus Voelter. 2013. *Language and IDE Modularization and Composition with MPS*. Springer Berlin Heidelberg, Berlin, Heidelberg, 383–430. https://doi.org/10.1007/978-3-642-35992-7_11

Markus Voelter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. 2014. Towards User-Friendly Projectional Editors. In *Software Language Engineering*, Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju (Eds.). Springer International Publishing, Cham, 41–61.

Mitchell Wand. 1986. Finding the source of type errors. In *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '86 (POPL '86)*. ACM Press. https://doi.org/10.1145/512644.512648

Thomas Williams and Didier Rémy. 2017. A principled approach to ornamentation in ML. *Proceedings of the ACM on Programming Languages* 2, POPL (Dec. 2017), 1–30. https://doi.org/10.1145/3158109

Eric Zhao, Raef Maroof, Anand Dukkipati, Andrew Blinn, Zhiyi Pan, and Cyrus Omar. 2024. Total Type Error Localization and Recovery with Holes. *Proceedings of the ACM on Programming Languages* 8, POPL (Jan. 2024), 2041–2068. https://doi.org/10.1145/3632910

Received 2024-07-09; accepted 2024-11-07