Flexible Structure Editing of Well-Typed Expressions
(Extended Abstract)

David Moon
University of Colorado Boulder
dmoon1221@gmail.com

Cyrus Omar
University of Chicago
cyrus.omar@gmail.com

R. Benjamin Shapiro
University of Colorado Boulder
ben.shapiro@colorado.edu

1 Introduction
Structure editors allow the programmer to edit the tree structure of a program directly. They can provide cognitive benefits for novice programmers, simplify language composition, and improve the availability of editor services. They are also known for being difficult to use.

We present the structure editor design of Hazel, a live functional programming environment. Hazel’s type-aware edit actions enforce the invariant that every program edit state is not only syntactically well-formed, but also statically [4] and dynamically [3] well-defined. Central to these guarantees is automatic insertion of typed holes at and around incomplete and type-incorrect portions of the program.

Prior structure editors have attempted to improve usability by allowing tightly scoped violations of syntactic well-formedness. For example, the Cornell ProgramSynthesizer enforces high-level tree structure but allows the user to construct expressions and bindings via text editing [5]. We do not have the same leniency if we are to uphold Hazel’s robust semantic guarantees—every edit state must be a well-formed, well-typed expression. We present our attempt at designing an ergonomic structure editor under this constraint.

2 Structure Editor Design
We designed Hazel’s structure editor to maximize carryover of users’ text editing intuitions. We motivate four main features supporting Hazel’s ergonomics, then demonstrate them in a concrete example.

(1) Automatic Hole Insertion. Naïvely enforcing an injunction on ill-typed edit states would force programmers to construct programs in a rigid “outside-in” manner. Hazel’s type-aware edit actions address this issue by automatically enclosing unfinshed or type-inconsistent parts of the program in holes.

(2) Tree Linearization. Text editors present character sequences in a 2D interface by dividing the sequence into rows. Hazel’s syntax directly encodes the vertical and horizontal linearity of text editing to which computer users are accustomed. An expression in Hazel is encoded as a sequence of leading line items, which may introduce bindings (e.g., \( \text{let } x = 1 \text{ in} \)), followed by a concluding line item (e.g., \( x + 1 \)).

A concluding line item is encoded as an unassociated infix operator sequence—Hazel internally re-associates operator sequences as needed for typechecking, but the association does not affect how the user navigates and edits the sequence. This structure helps facilitate natural left-to-right program editing and free use of vertical space. As we will see below, line items and operators also serve as useful landmarks for staged node transformations.

(3) Explicit Tree Signifiers. Contemporary structure editors have made significant strides in usability, but issues remain. In a controlled user evaluation of MPS, a state-of-the-art structure editor, Berger et al. [1] report that novice users perceive selection as inaccurate relative to that in text editing, and that both novices and experts perceive deletion as relatively inaccurate. The issue is that MPS presents a linear textual notation but often requires awareness of the underlying tree structure in order to predict edit results.

To address this problem, Hazel features a novel cursor system that augments the familiar cursor of text editors with visual markers of the current node’s tree structure. These markers facilitate understanding of the program’s tree structure as well as indicate which node would be removed by deletion. At the same time, they require no additional screen real estate, a known issue with blocks-based interfaces [2].

(4) Node Staging Mode. Prior work on structure editing ergonomics has proposed a variety of solutions to constructing infix operator sequences in the manner of text editing, i.e., with similar keystrokes. There has been comparatively little attention paid to complex tree transformations involving other syntactic forms. Hazel features a novel node staging mode that facilitates exploration of valid node transformations. Whereas other structure editors require a “configure then invoke” flow, where child nodes must be selected before invoking construction of the new parent node, Hazel’s node staging mode enables a more natural “invoke then configure” flow, similar to that of text editing.

2.1 Example
We now give an example-driven overview of these features.

Suppose we are implementing a combat game and, specifically, defining a function \( \text{damage : Attack } \rightarrow \text{Num} \). An Attack is a tuple consisting of the attack type and a critical hit multiplier, and the returned Num is the damage points inflicted upon the current player.
All following listings should be interpreted as filling the body of the damage function.

Suppose we have so far implemented damage as follows.

```
let damage : Attack -> Num = λ(atyp, crit).( 
    | case atyp 
        | Magic => 5  
        | Melee => 2 * crit 
    end 
)
```

We press keys `...`

```
| case atyp 
    | Magic => 5  
    | Melee => 2 * crit + 1 
| case atyp end 
```

...and `...

```
| case atyp 
    | Magic => 5  
    | Melee => 2 * crit + 1 
| case atyp end 
```

A naïve structure editor design would apply edits as context-free transformations, leading to the result $2*(\text{crit}+1)$. Hazel avoids this issue via Features (1) and (2), while Hazel’s edit actions re-parse operator precedence as needed for type-checking. This approach is similar to MPS’s side transformations [6]. We defer to [4] for more examples of hole insertion.

We have in scope the player’s defenseScore and want to integrate it into the damage calculation. Our plan is to bind the current expression to a new variable attackScore and return a damage score in terms of attackScore and defenseScore.

```
let attackScore = in 
| case atyp ... end 
```

We have entered node staging mode [Feature (4)]. Just as a code completion menu facilitates exploration of valid token completions, node staging mode facilitates exploration of valid placements of a node’s syntactic delimiters. The in delimiter is highlighted in dark green to indicate that it is the delimiter to be placed, while the dark green guide on the left signifies possible positions. The two children nodes of the let line are highlighted to indicate that, if we were to press Backspace again, they would be deleted as well.

We press `→` or `↓` to move in to the next position.

```
let attackScore = 
    case atyp ... end 
in 
```

We press `Enter` to accept this position and return to normal editing mode.

```
let attackScore = 
    case atyp ... end 
in 
```

We have entered our final calculation but have forgotten to account for operator precedence. We press `(` at the start of the return expression.

```
let attackScore = 
    case atyp ... end 
{
    attackScore - defenseScore * 2 
}
```

In the case of parentheses, Hazel enters node staging mode automatically if it detects ambiguity in intent. We press `→` for the next position.

```
let attackScore = 
    case atyp ... end 
{
    attackScore - defenseScore * 2 
}
```

Finally we press `Enter` or `)` to accept and exit node staging.

```
let attackScore = 
    case atyp ... end 
{
    attackScore - defenseScore * 2 
}
```

Once again, note the similarity in keystrokes to the text editor experience.

Now that we have created attackScore, we want to bind it to the subsequent case expression. In a text editor, we would delete the delimeter in and retype it after the case expression. Similarly, we hit Backspace.

```
let attackScore = 
    case atyp ... end 
{
    attackScore - defenseScore * 2 
}
```

We have entered node staging mode [Feature (4)]. Just as a code completion menu facilitates exploration of valid token completions, node staging mode facilitates exploration of valid placements of a node’s syntactic delimiters. The in delimiter is highlighted in dark green to indicate that it is the delimiter to be placed, while the dark green guide on the left signifies possible positions. The two children nodes of the let line are highlighted to indicate that, if we were to press Backspace again, they would be deleted as well.

We press `→` or `↓` to move in to the next position.

```
let attackScore = 
    case atyp ... end 
{
    attackScore - defenseScore * 2 
}
```

We press `Enter` to accept this position and return to normal editing mode.

```
let attackScore = 
    case atyp ... end 
{
    attackScore - defenseScore * 2 
}
```

Note the similarity in keystrokes to the text editor experience.

```
let attackScore = 
    case atyp ... end 
{
    attackScore - defenseScore * 2 
}
```

We have entered node staging mode automatically if it detects ambiguity in intent. We press `→` for the next position.

```
let attackScore = 
    case atyp ... end 
{
    attackScore - defenseScore * 2 
}
```

Finally we press `Enter` or `)` to accept and exit node staging.

```
let attackScore = 
    case atyp ... end 
{
    attackScore - defenseScore * 2 
}
```

Once again, note the similarity in keystrokes to the text editor experience.
References


