Extended Abstract: Flexible Structure Editing of Well-Typed Expressions

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 programmers 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 hard to use.

We present the structure editor design of Hazel, a live functional programming environment. By virtue of its type-aware edit actions and execution semantics for incomplete programs, Hazel enforces 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* to enclose incomplete and type-inconsistent parts of the program.

Prior structure editors have attempted to improve usability by allowing tightly scoped violations of syntactic well-formedness. For example, the Cornell Program Synthesizer 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 unfinished or type-inconsistent parts of the program in holes. Together with Feature **(2)**, Hazel's semantic internalization of type inconsistencies enables natural left-to-right construction while maintaining well-typedness.

(2) ***Tree Flattening.*** 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 non-terminal line items (e.g., `let x = 1 in`) followed by a nonbinding terminal line item (e.g., `x + 1`). A terminal line item is encoded at the top level as an unassociated infix operator sequence. This structure helps facilitate natural left-to-right program construction and free use of vertical space. As we will see later, line items and operands also serve as useful landmarks for complex 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 no similar attention paid to complex tree transformations involving other syntactic forms. Hazel features a novel *node staging mode* that faciliates 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 `damage : Attack -> 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.

TyDe'19, August 18, 2019, Berlin, Germany
2019.

```
let damage : Attack -> Num = λ(atyp, crit).{
  |_
}
```

All following listings should be interpreted as filling the body of the damage function.

⋮

Suppose we have so far implemented damage as follows.

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

We press keys [+] …

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

…and [1].

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

A naïve structure editor design would apply edit commands as context-free transformations, leading to the unintended result `2 * (crit + 1)`. Hazel avoids this issue via Features **(1)** and **(2)**, while Hazel's edit actions re- parse operator precedence as needed for typechecking. This approach is similar to MPS's side transformations [6].

⋮

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`.

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

We have moved the cursor to the start of the `case` expression. Note that the dark green caret is augmented with a light green shading of the the `case` node and outlines of its child nodes [Feature **(3)**].

We press [Enter] to create a new empty line. Direct encoding of line items allows us to create space around existing nodes in preparation for a complex node construction, just like in a text editor.

```
|
case atyp … end
```

We type the keys [l] [e] [t] [Space] to construct a new let line.

```
let |_ = _ in
case atyp … end
```

By recognizing keywords that prefix syntactic forms, Hazel eliminates the requirement to learn keyboard shortcuts.

⋮

```
let attackScore = _ in|
case atyp … end
```

Now that we have created `attackScore`, we want to bind it to the `case` expression on the following line. In a text editor, we would delete the delimiter `in` and retype it after the `case` expression. Similarly, we hit [Backspace].

```
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|
_
```

Note the similarity in keystrokes to the text editor experience.

⋮

```
let attackScore =
  case atyp … end
in
|attackScore - defenseScore * 2
```

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
in
(|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
in
(attackScore |- defenseScore) * 2
```

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

```
let attackScore =
  case atyp … end
in
(attackScore - defenseScore)| * 2
```

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

References

[1] 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 (FSE 2016)*. ACM, New York, NY, USA, 763–774. https://doi.org/10.1145/2950290.2950315

[2] Jens Monig, Yoshiki Ohshima, and John Maloney. 2015. Blocks at Your Fingertips: Blurring the Line Between Blocks and Text in GP. In *Proceedings of the 2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond) (BLOCKS AND BEYOND '15)*. IEEE Computer Society, Washington, DC, USA, 51–53. https://doi.org/10.1109/BLOCKS.2015.7369001

[3] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live functional programming with typed holes. *PACMPL* 3, POPL (2019), 14:1–14:32. https://doi.org/10.1145/3290327

[4] 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, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 86–99. https://doi.org/10.1145/3009837

[5] Tim Teitelbaum and Thomas Reps. 1981. The Cornell Program Synthesizer: A Syntax-directed Programming Environment. *Commun. ACM* 24, 9 (Sept. 1981), 563–573. https://doi.org/10.1145/358746.358755

[6] Markus Voelter, Tamás Szabó, Sascha Lisson, Bernd Kolb, Sebastian Erdweg, and Thorsten Berger. 2016. Efficient Development of Consistent Projectional Editors Using Grammar Cells. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering (SLE 2016)*. ACM, New York, NY, USA, 28–40. https://doi.org/10.1145/2997364.2997365