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 language and editor. By virtue of its type-aware edit actions and execution semantics for incomplete programs, Hazel enforces the unique invariant that every program edit state is not only syntactically well-formed, but also statically [**?**] and dynamically [**?**] well-defined. Central to these guarantees is the use 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 structural invariants. For example, the Cornell Program Synthesizer enforces high-level tree structure but allows the programmer to construct expressions and assignment statements via text editing [**?**]. Hazel takes a different approach: it loosens its structural invariants by formally modeling incomplete programs. While this helps in some ways with designing for usability, Hazel's semantic guarantees require strict adherence to these invariants at all times. We present our attempt at designing an ergonomic structure editor under these constraints.

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 addresses this problem by automatically enclosing unfinished or type-inconsistent parts of the program in holes. By internalizing type inconsistencies into its semantics, Hazel allows for natural left-to-right construction of programs while maintaining well-typedness.

(2) ***Tree Flattening.*** While plain text is a linear sequence of characters, text editors present it in a two-dimensional interface by dividing the sequence into rows. Hazel's syntax explicitly encodes the vertical and horizontal linearity of text editing to which computer users are accustomed. Expressions in Hazel are broken up into sequences of line items; within each line item, sequences of infix operators are encoded sequentially. As we will see later, line items and operands serve as useful landmarks for complex node transformations.

(3) ***Explicit Tree Signifiers.*** Early structure editors were hard to use because they presented a textual notation but had editing interfaces requiring awareness of the underlying tree structure. Comtemporary tools have significantly improved usability, but issues remain. In a controlled user evaluation of MPS, a state-of-the-art structure editor, Berger *et al.* [**?**] 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. These results suggest that there remain discrepancies between the visual signifiers and editing affordances.

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. Not only do these markers faciliate understanding of the program's tree structure, but they also clearly indicate what would be removed by deletion.

(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 the same 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 present an example-driven overview of features **(2)**, **(3)**, and **(4)**, deferring to [**?**] for examples of **(1)**.

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

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

hit multiplier, and the returned `Num` is the damage points inflicted upon the current player.

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

To save space, we do not repeat the above and focus our listings on 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 `+` and `1`.

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

A naïve structure editor design would maintain a strict tree structure and apply edit commands as context-free transformations. Hazel avoids this issue via Feature **(2)**, while Hazel's edit actions re-parse operator precedence as needed for typchecking. This approach is similar to MPS's side transformations [**?**].

⋮

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 text cursor 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. Explicit 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 the variable `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 press `Backspace` again, they would be deleted as well.

We press `→` or `↓` to move the delimiter 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.