Hazelnut: A Bidirectionally Typed Structure Editor Calculus

Cyrus Omar[1] Ian Voysey[1] Michael Hilton[2] Jonathan Aldrich[1] Matthew A. Hammer[3]

[1]Carnegie Mellon University, USA
{comar, iev, aldrich}@cs.cmu.edu

[2]Oregon State University, USA
hiltonm@eecs.oregonstate.edu

[3]University of Colorado Boulder, USA
matthew.hammer@colorado.edu

Abstract

Structure editors allow programmers to edit the tree structure of a program directly. This can have cognitive benefits, particularly for novice and end-user programmers. It also simplifies matters for tool designers, because they do not need to contend with malformed program text.

This paper introduces Hazelnut, a structure editor based on a small bidirectionally typed lambda calculus extended with *holes* and a *cursor*. Hazelnut goes one step beyond syntactic well-formedness: its edit actions operate over statically meaningful incomplete terms. Naïvely, this would force the programmer to construct terms in a rigid "outside-in" manner. To avoid this problem, the action semantics automatically places terms assigned a type that is inconsistent with the expected type *inside* a hole. This meaningfully defers the type consistency check until the term inside the hole is *finished*.

Hazelnut is not intended as an end-user tool itself. Instead, it serves as a foundational account of typed structure editing. To that end, we describe how Hazelnut's rich metatheory, which we have mechanized using the Agda proof assistant, serves as a guide when we extend the calculus to include binary sum types. We also discuss various interpretations of holes, and in so doing reveal connections with gradual typing and contextual modal type theory, the Curry-Howard interpretation of contextual modal logic. Finally, we discuss how Hazelnut's semantics lends itself to implementation as an event-based functional reactive program. Our simple reference implementation is written using `js_of_ocaml`.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.2.3 [*Software Engineering*]: Coding Tools and Techniques—Program Editors

Keywords structure editors, bidirectional type systems, gradual typing, mechanized metatheory

1. Introduction

Programmers typically interact with meaningful programs only indirectly, by editing text that is first parsed according to a textual syntax and then typechecked according to a static semantics. This indirection has practical benefits, to be sure – text editors and other text-based tools benefit from decades of development effort. However, this indirection through text also introduces some fundamental complexity into the programming process.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

POPL'17, January 15–21, 2017, Paris, France
ACM. 978-1-4503-4660-3/17/01...$15.00
http://dx.doi.org/10.1145/3009837.3009900

First, it requires that programmers learn the subtleties of the textual syntax (e.g. operator precedence.) This is particularly challenging for novices [2, 34, 56], and even experienced programmers frequently make mistakes [24, 26, 34].

Second, many sequences of characters do not correspond to meaningful programs. This complicates the design of program editors and other interactive programming tools. In a dataset gathered by Yoon and Myers consisting of 1460 hours of Eclipse edit logs [67], 44.2% of edit states were syntactically malformed. Some additional percentage of edit states were well-formed but ill-typed (the dataset was not complete enough to determine the exact percentage.) Collectively, we refer to these edit states as *meaningless edit states*, because they are not given static or dynamic meaning by the language definition. As a consequence, it is difficult to design useful language-aware editor services, e.g. syntax highlighting [53], type-aware code completion [39, 43], and refactoring services [36]. Editors must either disable these editor services when they encounter meaningless edit states or deploy *ad hoc* heuristics, e.g. by using whitespace to guess the intent [15, 27].

These complications have motivated a long line of research into *structure editors*, i.e. program editors where every edit state corresponds to a program structure [59].[1]

Most structure editors are *syntactic structure editors*, i.e. the edit state corresponds to a syntax tree with *holes* that stand for branches of the tree that have yet to be constructed, and the edit actions are context-free tree transformations. For example, Scratch is a syntactic structure editor that has achieved success as a tool for teaching children how to program [51].

Researchers have also designed syntactic structure editors for more sophisticated languages with non-trivial static type systems. Contemporary examples include `mbeddr`, a structure editor for a C-like language [63], TouchDevelop, a structure editor for an object-oriented language [60], and Lamdu, a structure editor for a functional language similar to Haskell [33]. Each of these editors presents an innovative user interface, but the non-trivial type and binding structure of the underlying language complicates its design. The problem is that syntactic structure editors do not assign static meaning to every edit state – they guarantee only that every edit state corresponds to a syntactically well-formed tree. These editors must also either selectively disable editor services that require an understanding of the semantics of the program being written, or deploy *ad hoc* heuristics.

This paper develops a principled solution to this problem. We introduce Hazelnut, a *typed structure editor* based on a bidirectionally typed lambda calculus extended to assign static meaning to expressions and types with holes, which we call **H-expressions** and **H-types**. Hazelnut's formal *action semantics* maintains the invariant that every edit state is a statically meaningful (i.e. well-H-

[1] Structure editors are also variously known as *structured editors*, *structural editors*, *syntax-directed editors* and *projectional editors*.

typed) H-expression with a single superimposed *cursor*. We call H-expressions and H-types with a cursor **Z-expressions** and **Z-types** (so prefixed because our encoding follows Huet's *zipper* pattern [25].)

Naïvely, enforcing an injunction on ill-H-typed edit states would force programmers to construct programs in a rigid "outside-in" manner. For example, the programmer would often need to construct the outer function application form before identifying the intended function. To address this problem, Hazelnut leaves newly constructed expressions *inside* a hole if the expression's type is inconsistent with the expected type. This meaningfully defers the type consistency check until the expression inside the hole is *finished*. In other words, holes appear both at the leaves and at the internal nodes of an H-expression that remain under construction.

The remainder of this paper is organized as follows:

- We begin in Sec. 2 with two examples of edit sequences to develop the reader's intuitions.
- We then give a detailed overview of Hazelnut's semantics and metatheory, which has been mechanized using the Agda proof assistant, in Sec. 3.
- Hazelnut is a foundational calculus, i.e. a calculus that language and editor designers are expected to extend with higher level constructs. We extend Hazelnut with binary sum types in Sec. 4 to demonstrate how Hazelnut's rich metatheory guides one such extension.
- In Sec. 5, we briefly describe how Hazelnut's action semantics lends itself to efficient implementation as a functional reactive program. Our reference implementation is written using the OCaml `React` library and `js_of_ocaml`.
- In Sec. 6, we summarize related work. In particular, much of the technical machinery needed to handle type holes coincides with machinery developed for gradual type systems. Similarly, expression holes can be interpreted as the closures of contextual modal type theory, which, by its correspondence with contextual modal logic, suggests logical foundations for the system.
- We conclude in Sec. 7 by summarizing our vision of a principled science of structure editor design rooted in type theory, and suggest a number of future directions.

The supplemental material can be accessed from:

`http://cs.cmu.edu/~comar/hazelnut-popl17/`

2. Programming in Hazelnut

2.1 Example 1: Constructing the Increment Function

Figure 1 shows an edit sequence that constructs the increment function, of type $(\texttt{num} \to \texttt{num})$, starting from the empty hole via the indicated sequence of actions. We will introduce Hazelnut's formal syntax and define the referenced rules in Sec. 3.[2] First, let us build some high-level intuitions.

The edit state in Hazelnut is a Z-expression, $\hat{e}$. Every Z-expression has a single H-expression, $\dot{e}$, or H-type, $\dot{\tau}$, under the cursor, typeset $\triangleright\dot{e}\triangleleft$ or $\triangleright\dot{\tau}\triangleleft$, respectively.[3] For example, on Line 1, the empty expression hole, $(\!|\,|\!)$, is under the cursor.

Actions act relative to the cursor. The first action, on Line 1, is `construct lam` x. This results in the Z-expression on Line 2, consisting of a lambda abstraction with argument x under an arrow type ascription. The cursor is placed (arbitrarily) on the argument type hole.

[2] For concision, the column labeled **Rule** in Figures 1 and 2 indicates only the relevant *non-zipper* rule (see Sec. 3.3.8.) The reader is encouraged to follow along with these examples using the reference implementation. The derivations for these examples are also available in the Agda mechanization.

[3] The reference implementation omits the triangles, while the Agda mechanization necessarily omits the colors.

#	Z-Expression	Next Action	Rule
1	$\triangleright(\!\|\,\|\!)\triangleleft$	`construct lam` x	(13e)
2	$(\lambda x.(\!\|\,\|\!)) : (\triangleright(\!\|\,\|\!)\triangleleft \to (\!\|\,\|\!))$	`construct num`	(12b)
3	$(\lambda x.(\!\|\,\|\!)) : (\triangleright\texttt{num}\triangleleft \to (\!\|\,\|\!))$	`move parent`	(6c)
4	$(\lambda x.(\!\|\,\|\!)) : \triangleright(\texttt{num} \to (\!\|\,\|\!))\triangleleft$	`move child 2`	(6b)
5	$(\lambda x.(\!\|\,\|\!)) : (\texttt{num} \to \triangleright(\!\|\,\|\!)\triangleleft)$	`construct num`	(12b)
6	$(\lambda x.(\!\|\,\|\!)) : (\texttt{num} \to \triangleright\texttt{num}\triangleleft)$	`move parent`	(6d)
7	$(\lambda x.(\!\|\,\|\!)) : \triangleright(\texttt{num} \to \texttt{num})\triangleleft$	`move parent`	(8d)
8	$\triangleright(\lambda x.(\!\|\,\|\!)) : (\texttt{num} \to \texttt{num})\triangleleft$	`move child 1`	(8a)
9	$\triangleright(\lambda x.(\!\|\,\|\!))\triangleleft : (\texttt{num} \to \texttt{num})$	`move child 1`	(8e)
10	$(\lambda x.\triangleright(\!\|\,\|\!)\triangleleft) : (\texttt{num} \to \texttt{num})$	`construct var` x	(13c)
11	$(\lambda x.\triangleright x\triangleleft) : (\texttt{num} \to \texttt{num})$	`construct plus`	(13l)
12	$(\lambda x.(x + \triangleright(\!\|\,\|\!)\triangleleft)) : (\texttt{num} \to \texttt{num})$	`construct lit 1`	(13j)
13	$(\lambda x.(x + \triangleright\underline{1}\triangleleft)) : (\texttt{num} \to \texttt{num})$	—	—

Figure 1. Constructing the increment function in Hazelnut.

now assume $\mathit{incr} : \texttt{num} \to \texttt{num}$

#	Z-Expression	Next Action	Rule
14	$\triangleright(\!\|\,\|\!)\triangleleft$	`construct var` incr	(13c)
15	$\triangleright\mathit{incr}\triangleleft$	`construct ap`	(13h)
16	$\mathit{incr}(\triangleright(\!\|\,\|\!)\triangleleft)$	`construct var` incr	(13d)
17	$\mathit{incr}((\!\|\triangleright\mathit{incr}\triangleleft\|\!))$	`construct ap`	(13h)
18	$\mathit{incr}((\!\|\mathit{incr}(\triangleright(\!\|\,\|\!)\triangleleft)\|\!))$	`construct lit 3`	(13j)
19	$\mathit{incr}((\!\|\mathit{incr}(\triangleright\underline{3}\triangleleft)\|\!))$	`move parent`	(8j)
20	$\mathit{incr}((\!\|\triangleright\mathit{incr}(\underline{3})\triangleleft\|\!))$	`move parent`	(8p)
21	$\mathit{incr}(\triangleright(\!\|\mathit{incr}(\underline{3})\|\!)\triangleleft)$	`finish`	(16b)
22	$\mathit{incr}(\triangleright\mathit{incr}(\underline{3})\triangleleft)$	—	—

Figure 2. Applying the increment function.

The actions on Lines 2-5 complete the type ascription. In particular, the `construct num` action constructs the `num` type at the cursor and the `move parent` and `move child 2` action sequence moves the cursor to the next hole. In practice, an editor would also define compound movement actions, e.g. an action that moves the cursor directly to the next hole, in terms of these primitive movement actions.

After moving to the function body, Lines 10-12 complete it by first constructing the variable x, then constructing the plus form, and finally constructing the number literal $\underline{1}$. Notice that we did not need to construct the function body in an "outside-in" manner, i.e. we constructed x before constructing the outer plus form inside which x ultimately appears. The transient function bodies, x and $(x + (\!|\,|\!))$, can be checked against the given return type, `num` (as we will detail in Sec. 3.1.)

2.2 Example 2: Applying the Increment Function

Figure 2 shows an edit sequence that constructs the expression $\mathit{incr}(\mathit{incr}(\underline{3}))$, where incr is assumed bound to the increment function from Figure 1.

We begin on Line 14 by constructing the variable incr. Line 15 then constructs the application form with incr in function position, leaving the cursor on a hole in the argument position. Notice again that we did not construct the outer application form before identifying the function being applied.

We now need to apply incr again, so we perform the same action on Line 16 as we did on Line 14, i.e. `construct var` incr. In a syntactic structure editor, performing such an action would result in the following edit state:

$$\mathit{incr}(\triangleright\mathit{incr}\triangleleft)$$

This edit state is ill-typed (after *cursor erasure*): the argument of incr must be of type `num` but here it is of type $(\texttt{num} \to \texttt{num})$. Hazelnut does not allow such an edit state to arise.

We could alternatively have performed the `construct ap` action on Line 16. This would result in the following edit state, which is well-typed according to the static semantics that we will define in the next section:

$$incr(\text{⦇⦈}(\triangleright\text{⦇⦈}\triangleleft))$$

The problem is that the programmer is not able to identify the intended function, *incr*, before constructing the function application form. This stands in contrast to Lines 14-15.

Hazelnut's action semantics addresses this problem: rather than disallowing the `construct var` *incr* action on Line 16, it leaves *incr* inside a hole:

$$incr(\text{⦇}\triangleright incr\triangleleft\text{⦈})$$

This defers the type consistency check, exactly as an empty hole in the same position does. One way to think about non-empty holes is as an internalization of the "squiggly underline" that text or syntactic structure editors display to indicate a type inconsistency. By internalizing this concept, the presence of a type inconsistency does not leave the entire program formally meaningless.

The expression inside a non-empty hole must itself be well-typed, so the programmer can continue to edit it. Lines 17-18 proceed to apply the inner mention of *incr* to a number literal, $\underline{3}$. Finally, Lines 18-19 move the cursor to the non-empty hole and Line 21 performs the `finish` action. The `finish` action removes the hole if the type of the expression inside the hole is consistent with the expected type, as it now is. This results in the final edit state on Line 22, as desired. In practice, the editor might automatically perform the `finish` action as soon as it becomes possible, but for simplicity, Hazelnut formally requires that it be performed explicitly.

3. Hazelnut, Formally

The previous section introduced Hazelnut by example. In this section, we systematically define the following structures:

- **H-types** and **H-expressions** (Sec. 3.1), which are types and expressions with holes. H-types classify H-expressions according to Hazelnut's **bidirectional static semantics**.
- **Z-types** and **Z-expressions** (Sec. 3.2), which superimpose a *cursor* onto H-types and H-expressions, respectively (following Huet's *zipper pattern* [25].) Every Z-type (resp. Z-expression) corresponds to an H-type (resp. H-expression) by *cursor erasure*.
- **Actions** (Sec. 3.3), which act relative to the cursor according to Hazelnut's **bidirectional action semantics**. The action semantics enjoys a rich metatheory. Of particular note, the *sensibility theorem* establishes that every edit state is well-typed after cursor erasure.

Our overview below omits certain "uninteresting" details. The supplement includes the complete collection of rules, in definitional order. These rules, along with the proofs of all of the metatheorems discussed in this section (and several omitted auxiliary lemmas), have been mechanized using the Agda proof assistant [41] (discussed in Sec. 3.5.)

3.1 H-types and H-expressions

Figure 3 defines the syntax of H-types, $\dot{\tau}$, and H-expressions, $\dot{e}$. Most forms correspond directly to those of the simply typed lambda calculus (STLC) extended with a single base type, `num`, of numbers (cf. [22].) The number expression corresponding to the mathematical number n is drawn $\underline{n}$, and for simplicity, we define only a single arithmetic operation, $(\dot{e} + \dot{e})$. The form $\dot{e} : \dot{\tau}$ is an explicit *type ascription*. In addition to these standard forms, *type holes* and *empty expression holes* are both drawn ⦇⦈ and *non-empty expression holes* are drawn ⦇$\dot{e}$⦈. Types and expressions that contain no holes are *complete types* and *complete expressions*, respectively.

Hazelnut's static semantics is organized as a *bidirectional type system* [9, 10, 14, 47] around the two mutually defined judgements in Figure 4. Derivations of the type analysis judgement, $\dot{\Gamma} \vdash \dot{e} \Leftarrow \dot{\tau}$,

$$\begin{array}{llll} \mathsf{HTyp} & \dot{\tau} & ::= & (\dot{\tau} \to \dot{\tau}) \mid \mathtt{num} \mid \text{⦇⦈} \\ \mathsf{HExp} & \dot{e} & ::= & x \mid (\lambda x.\dot{e}) \mid \dot{e}(\dot{e}) \mid \underline{n} \mid (\dot{e} + \dot{e}) \mid \dot{e} : \dot{\tau} \mid \text{⦇⦈} \mid \text{⦇}\dot{e}\text{⦈} \end{array}$$

Figure 3. Syntax of H-types and H-expressions. Metavariable x ranges over variables and n ranges over numerals.

$\boxed{\dot{\Gamma} \vdash \dot{e} \Rightarrow \dot{\tau}}$ $\dot{e}$ synthesizes $\dot{\tau}$

$$\frac{}{\dot{\Gamma}, x : \dot{\tau} \vdash x \Rightarrow \dot{\tau}} \tag{1a}$$

$$\frac{\dot{\Gamma} \vdash \dot{e}_1 \Rightarrow \dot{\tau}_1 \qquad \dot{\tau}_1 \blacktriangleright_{\to} (\dot{\tau}_2 \to \dot{\tau}) \qquad \dot{\Gamma} \vdash \dot{e}_2 \Leftarrow \dot{\tau}_2}{\dot{\Gamma} \vdash \dot{e}_1(\dot{e}_2) \Rightarrow \dot{\tau}} \tag{1b}$$

$$\frac{}{\dot{\Gamma} \vdash \underline{n} \Rightarrow \mathtt{num}} \tag{1c}$$

$$\frac{\dot{\Gamma} \vdash \dot{e}_1 \Leftarrow \mathtt{num} \qquad \dot{\Gamma} \vdash \dot{e}_2 \Leftarrow \mathtt{num}}{\dot{\Gamma} \vdash (\dot{e}_1 + \dot{e}_2) \Rightarrow \mathtt{num}} \tag{1d}$$

$$\frac{\dot{\Gamma} \vdash \dot{e} \Leftarrow \dot{\tau}}{\dot{\Gamma} \vdash \dot{e} : \dot{\tau} \Rightarrow \dot{\tau}} \tag{1e}$$

$$\frac{}{\dot{\Gamma} \vdash \text{⦇⦈} \Rightarrow \text{⦇⦈}} \tag{1f}$$

$$\frac{\dot{\Gamma} \vdash \dot{e} \Rightarrow \dot{\tau}}{\dot{\Gamma} \vdash \text{⦇}\dot{e}\text{⦈} \Rightarrow \text{⦇⦈}} \tag{1g}$$

$\boxed{\dot{\Gamma} \vdash \dot{e} \Leftarrow \dot{\tau}}$ $\dot{e}$ analyzes against $\dot{\tau}$

$$\frac{\dot{\tau} \blacktriangleright_{\to} (\dot{\tau}_1 \to \dot{\tau}_2) \qquad \dot{\Gamma}, x : \dot{\tau}_1 \vdash \dot{e} \Leftarrow \dot{\tau}_2}{\dot{\Gamma} \vdash (\lambda x.\dot{e}) \Leftarrow \dot{\tau}} \tag{2a}$$

$$\frac{\dot{\Gamma} \vdash \dot{e} \Rightarrow \dot{\tau}' \qquad \dot{\tau} \sim \dot{\tau}'}{\dot{\Gamma} \vdash \dot{e} \Leftarrow \dot{\tau}} \tag{2b}$$

Figure 4. H-type synthesis and analysis.

$\boxed{\dot{\tau} \sim \dot{\tau}'}$ $\dot{\tau}$ and $\dot{\tau}'$ are consistent

$$\frac{}{\text{⦇⦈} \sim \dot{\tau}} \qquad \frac{}{\dot{\tau} \sim \text{⦇⦈}} \qquad \frac{}{\dot{\tau} \sim \dot{\tau}} \qquad \frac{\dot{\tau}_1 \sim \dot{\tau}_1' \qquad \dot{\tau}_2 \sim \dot{\tau}_2'}{(\dot{\tau}_1 \to \dot{\tau}_2) \sim (\dot{\tau}_1' \to \dot{\tau}_2')} \tag{3a-d}$$

$\boxed{\dot{\tau} \blacktriangleright_{\to} (\dot{\tau}_1 \to \dot{\tau}_2)}$ $\dot{\tau}$ has matched arrow type $(\dot{\tau}_1 \to \dot{\tau}_2)$

$$\frac{}{\text{⦇⦈} \blacktriangleright_{\to} (\text{⦇⦈} \to \text{⦇⦈})} \tag{4a} \qquad \frac{}{(\dot{\tau}_1 \to \dot{\tau}_2) \blacktriangleright_{\to} (\dot{\tau}_1 \to \dot{\tau}_2)} \tag{4b}$$

Figure 5. H-type consistency and matched arrow types.

establish that $\dot{e}$ can appear where an expression of type $\dot{\tau}$ is expected. Derivations of the type synthesis judgement, $\dot{\Gamma} \vdash \dot{e} \Rightarrow \dot{\tau}$, synthesize (a.k.a. *locally infer* [47]) a type from $\dot{e}$. Type synthesis is necessary in positions where an expected type is not available (e.g. at the top level.) Algorithmically, the type is an "input" of the type analysis judgement, but an "output" of the type synthesis judgement. Making a judgemental distinction between these two notions will be essential in our action semantics (Sec. 3.3.)

If an expression is able to synthesize a type, it can also be analyzed against that type, or any other *consistent* type, according to the *subsumption rule*, Rule (2b).

The *H-type consistency judgement*, $\dot{\tau} \sim \dot{\tau}'$, that appears as a premise in the subsumption rule is a reflexive and symmetric (but not transitive) relation between H-types defined judgementally in Figure 5. This relation coincides with equality for complete H-types. Two incomplete H-types are consistent if they differ only

at positions where a hole appears in either type. The type hole is therefore consistent with every type. This notion of H-type consistency coincides with the notion of type consistency that Siek and Taha discovered in their foundational work on gradual type systems, if we interpret the type hole as the ? (i.e. unknown) type [54].

Typing contexts, $\dot{\Gamma}$, map each variable $x \in \mathrm{dom}(\dot{\Gamma})$ to an hypothesis $x : \dot{\tau}$. We identify contexts up to exchange and contraction and adopt the standard identification convention for structures that differ only up to alpha-renaming of bound variables. All hypothetical judgements obey a standard weakening lemma. Rule (1a) establishes that variable expressions synthesize the hypothesized H-type, in the standard manner.

Rule (2a) defines analysis for lambda abstractions, $(\lambda x.\dot{e})$. There is no type synthesis rule that applies to this form, so lambda abstractions can appear only in analytic position, i.e. where an expected type is known.[4] Rule (2a) is not quite the standard rule, as reproduced below:

$$\frac{\dot{\Gamma}, x : \dot{\tau}_1 \vdash \dot{e} \Leftarrow \dot{\tau}_2}{\dot{\Gamma} \vdash (\lambda x.\dot{e}) \Leftarrow (\dot{\tau}_1 \to \dot{\tau}_2)}$$

The problem is that this standard rule alone leaves us unable to analyze lambda abstractions against the type hole, because the type hole is not immediately of the form $(\dot{\tau}_1 \to \dot{\tau}_2)$. There are two plausible solutions to this problem. One solution would be to define a second rule specifically for this case:

$$\frac{\dot{\Gamma}, x : \langle\!|\,|\!\rangle \vdash \dot{e} \Leftarrow \langle\!|\,|\!\rangle}{\dot{\Gamma} \vdash (\lambda x.\dot{e}) \Leftarrow \langle\!|\,|\!\rangle}$$

Instead, we combine these two possible rules into a single rule through the simple auxiliary *matched arrow type* judgement, $\dot{\tau} \blacktriangleright_{\to} (\dot{\tau}_1 \to \dot{\tau}_2)$, defined in Figure 5. This judgement leaves arrow types alone and assigns the type hole the matched arrow type $(\langle\!|\,|\!\rangle \to \langle\!|\,|\!\rangle)$. It is easy to see that the two rules above are admissible by appeal to Rule (2a) and the matched arrow type judgement. Encouragingly, the matched arrow type judgement also arises in gradual type systems [11, 18, 49].

Rule (1b) is again nearly the standard rule for function application. It also makes use of the matched function type judgement to combine what would otherwise need to be two rules for function application – one for when e_1 synthesizes an arrow type, and another for when e_1 synthesizes $\langle\!|\,|\!\rangle$.

Rule (1c) states that numbers synthesize the `num` type. Rule (1d) states that $\dot{e}_1 + \dot{e}_2$ behaves like a function over numbers.

Rule (1e) defines type synthesis of expressions of ascription form, $\dot{e} : \dot{\tau}$. This allows the programmer to explicitly state a type for the ascribed expression to be analyzed against.

The rules described so far are sufficient to type complete H-expressions. The two remaining rules give H-expressions with holes a well-defined static semantics.

Rule (1f) states that the empty expression hole synthesizes the type hole. Non-empty holes, which contain an H-expression that is "under construction" as described in Sec. 2, also synthesize the hole type. According to Rule (1g), the enveloped expression must synthesize some (arbitrary) type. (We do not need non-empty type holes because every H-type is a valid classifier of H-expressions.)

Because the hole type is consistent with every type, expression holes can be analyzed against any type by subsumption. For example, it is instructive to derive the following:

$$\mathit{incr} : (\mathtt{num} \to \mathtt{num}) \vdash \langle\!|\mathit{incr}|\!\rangle \Leftarrow \mathtt{num}$$

$$\begin{array}{lll} \mathsf{ZTyp}\ \hat{\tau} & ::= & \triangleright\dot{\tau}\triangleleft \mid (\hat{\tau} \to \dot{\tau}) \mid (\dot{\tau} \to \hat{\tau}) \\ \mathsf{ZExp}\ \hat{e} & ::= & \triangleright\dot{e}\triangleleft \mid (\lambda x.\hat{e}) \mid \hat{e}(\dot{e}) \mid \dot{e}(\hat{e}) \mid (\hat{e} + \dot{e}) \mid (\dot{e} + \hat{e}) \\ & \mid & \hat{e} : \dot{\tau} \mid \dot{e} : \hat{\tau} \mid \langle\!|\hat{e}|\!\rangle \end{array}$$

Figure 6. Syntax of Z-types and Z-expressions.

3.2 Z-types and Z-expressions

Figure 6 defines the syntax of Z-types, $\hat{\tau}$, and Z-expressions, $\hat{e}$. A Z-type (resp. Z-expression) represents an H-type (resp. H-expression) with a single superimposed *cursor*.

The only base cases in these inductive grammars are $\triangleright\dot{\tau}\triangleleft$ and $\triangleright\dot{e}\triangleleft$, which identify the H-type or H-expression that the cursor is on. All of the other forms correspond to the recursive forms in the syntax of H-types and H-expressions, and contain exactly one "hatted" subterm that identifies the subtree where the cursor will be found. Any other sub-term is "dotted", i.e. it is either an H-type or H-expression. Taken together, every Z-type and Z-expression contains exactly one selected H-type or H-expression by construction. This can be understood as an instance of Huet's *zipper pattern* [25] (which, coincidentally, Huet encountered while implementing a structure editor.)

We write $\hat{\tau}^{\diamond}$ for the H-type constructed by erasing the cursor from $\hat{\tau}$, which we refer to as the *cursor erasure* of $\hat{\tau}$. This straightforward metafunction is defined as follows:

$$\begin{aligned} \triangleright\dot{\tau}\triangleleft^{\diamond} &= \dot{\tau} \\ (\hat{\tau} \to \dot{\tau})^{\diamond} &= (\hat{\tau}^{\diamond} \to \dot{\tau}) \\ (\dot{\tau} \to \hat{\tau})^{\diamond} &= (\dot{\tau} \to \hat{\tau}^{\diamond}) \end{aligned}$$

Similarly, we write $\hat{e}^{\diamond}$ for cursor erasure of $\hat{e}$. The definition of this metafunction is analogous, so we omit it for concision.

This zipper structure is not the only way to model a cursor, though we have found it to be the most elegant for our present purposes. Another plausible strategy would be to formalize the notion of a relative path into an H-expression. This would then require defining the notion of consistency between a relative path and an H-expression, so we avoid it.

3.3 Actions

We now arrive at the heart of Hazelnut: its *bidirectional action semantics*. Figure 7 defines the syntax of *actions*, α, some of which involve *directions*, δ, and *shapes*, ψ.

Expression actions are governed by two mutually defined judgements, 1) the *synthetic action judgement*:

$$\dot{\Gamma} \vdash \hat{e} \Rightarrow \dot{\tau} \xrightarrow{\alpha} \hat{e}' \Rightarrow \dot{\tau}'$$

and 2) *the analytic action judgement*:

$$\dot{\Gamma} \vdash \hat{e} \xrightarrow{\alpha} \hat{e}' \Leftarrow \dot{\tau}$$

In some Z-expressions, the cursor is in a type ascription, so we also need a *type action judgement*:

$$\hat{\tau} \xrightarrow{\alpha} \hat{\tau}'$$

3.3.1 Sensibility

These judgements are governed by a critical metatheorem, *action sensibility* (or simply *sensibility*):

Theorem 1 (Action Sensibility)**.**

1. *If* $\dot{\Gamma} \vdash \hat{e}^{\diamond} \Rightarrow \dot{\tau}$ *and* $\dot{\Gamma} \vdash \hat{e} \Rightarrow \dot{\tau} \xrightarrow{\alpha} \hat{e}' \Rightarrow \dot{\tau}'$ *then* $\dot{\Gamma} \vdash \hat{e}'^{\diamond} \Rightarrow \dot{\tau}'$.
2. *If* $\dot{\Gamma} \vdash \hat{e}^{\diamond} \Leftarrow \dot{\tau}$ *and* $\dot{\Gamma} \vdash \hat{e} \xrightarrow{\alpha} \hat{e}' \Leftarrow \dot{\tau}$ *then* $\dot{\Gamma} \vdash \hat{e}'^{\diamond} \Leftarrow \dot{\tau}$.

In other words, if a Z-expression is statically meaningful, i.e. its cursor erasure is well-typed, then performing an action on it leaves the resulting Z-expression statically meaningful. More specifically, the first clause of Theorem 1 establishes that when an action is performed on a Z-expression whose cursor erasure synthesizes

[4] It is possible to also define a "half-annotated" synthetic lambda form, $\lambda x{:}\tau.e$, but for simplicity, we leave it out [9].

$$
\begin{array}{llll}
\text{Action} & \alpha & ::= & \texttt{move}\ \delta \mid \texttt{construct}\ \psi \mid \texttt{del} \mid \texttt{finish} \\
\text{Dir} & \delta & ::= & \texttt{child}\ n \mid \texttt{parent} \\
\text{Shape} & \psi & ::= & \texttt{arrow} \mid \texttt{num} \\
 & & \mid & \texttt{asc} \mid \texttt{var}\ x \mid \texttt{lam}\ x \mid \texttt{ap} \mid \texttt{lit}\ n \mid \texttt{plus} \\
 & & \mid & \texttt{nehole}
\end{array}
$$

Figure 7. Syntax of actions.

an H-type, the result is a Z-expression whose cursor erasure also synthesizes some (possibly different) H-type. The second clause establishes that when an action is performed using the analytic action judgement on an edit state whose cursor erasure analyzes against some H-type, the result is a Z-expression whose cursor erasure also analyzes against the same H-type.

This metatheorem deeply informs the design of the rules, given starting in Sec. 3.3.3. Its proof is by straightforward induction, so the reader is encouraged to think about the relevant proof case when considering each action rule below.

No sensibility theorem is needed for the type action judgement because every syntactically well-formed type is meaningful in Hazelnut. (Adding type variables to the language would require defining both a type-level sensibility theorem and type-level non-empty holes.)

3.3.2 Type Inconsistency

In some of the rules below, we will need to supplement our definition of type consistency from Figure 5 with a definition of *type inconsistency*, written $\dot{\tau} \nsim \dot{\tau}'$. One can define this notion either directly as the constructive negation of type consistency, or as a separate inductively defined judgement with the following key rule, which establishes that arrow types are inconsistent with `num`:

$$\frac{}{\texttt{num} \nsim (\dot{\tau}_1 \to \dot{\tau}_2)}$$

The mechanization proves that the judgemental definition of type inconsistency is indeed the negation of type consistency.

3.3.3 Action Subsumption

The action semantics includes a subsumption rule similar to the subsumption rule, Rule (2b), in the statics:

$$\frac{\dot{\Gamma} \vdash \hat{e}^{\diamond} \Rightarrow \dot{\tau}' \qquad \dot{\Gamma} \vdash \hat{e} \Rightarrow \dot{\tau}' \xrightarrow{\alpha} \hat{e}' \Rightarrow \dot{\tau}'' \qquad \dot{\tau} \sim \dot{\tau}''}{\dot{\Gamma} \vdash \hat{e} \xrightarrow{\alpha} \hat{e}' \Leftarrow \dot{\tau}} \tag{5}$$

In other words, if the cursor erasure of the edit state synthesizes a type, $\dot{\tau}'$, then we defer to the synthetic action judgement. The cursor erasure of the Z-expression resulting from performing the action α synthetically could have a different type, $\dot{\tau}''$, so we must check that it is consistent with the type provided for analysis, $\dot{\tau}$. The case for Rule (5) in the proof of Theorem 1 goes through by induction and static subsumption, i.e. Rule (2b). Algorithmically, subsumption should be the rule of last resort (see Sec. 3.4 for further discussion.)

3.3.4 Relative Movement

The rules below define relative movement within Z-types. They should be self-explanatory:

$$\frac{}{\triangleright(\dot{\tau}_1 \to \dot{\tau}_2)\triangleleft \xrightarrow{\texttt{move child 1}} (\triangleright\dot{\tau}_1\triangleleft \to \dot{\tau}_2)} \tag{6a}$$

$$\frac{}{\triangleright(\dot{\tau}_1 \to \dot{\tau}_2)\triangleleft \xrightarrow{\texttt{move child 2}} (\dot{\tau}_1 \to \triangleright\dot{\tau}_2\triangleleft)} \tag{6b}$$

$$\frac{}{(\triangleright\dot{\tau}_1\triangleleft \to \dot{\tau}_2) \xrightarrow{\texttt{move parent}} \triangleright(\dot{\tau}_1 \to \dot{\tau}_2)\triangleleft} \tag{6c}$$

$$\frac{}{(\dot{\tau}_1 \to \triangleright\dot{\tau}_2\triangleleft) \xrightarrow{\texttt{move parent}} \triangleright(\dot{\tau}_1 \to \dot{\tau}_2)\triangleleft} \tag{6d}$$

Two more rules are needed to recurse into the zipper structure. We define these zipper rules in an action-independent manner in Sec. 3.3.8.

The rules for relative movement within Z-expressions are similarly straightforward. Movement is type-independent, so we defer to an auxiliary expression movement judgement in both the analytic and synthetic case:

$$\frac{\hat{e} \xrightarrow{\texttt{move}\ \delta} \hat{e}'}{\dot{\Gamma} \vdash \hat{e} \Rightarrow \dot{\tau} \xrightarrow{\texttt{move}\ \delta} \hat{e}' \Rightarrow \dot{\tau}} \tag{7a}$$

$$\frac{\hat{e} \xrightarrow{\texttt{move}\ \delta} \hat{e}'}{\dot{\Gamma} \vdash \hat{e} \xrightarrow{\texttt{move}\ \delta} \hat{e}' \Leftarrow \dot{\tau}} \tag{7b}$$

The expression movement judgement is defined as follows.

Ascription

$$\frac{}{\triangleright\dot{e} : \dot{\tau}\triangleleft \xrightarrow{\texttt{move child 1}} \triangleright\dot{e}\triangleleft : \dot{\tau}} \tag{8a}$$

$$\frac{}{\triangleright\dot{e} : \dot{\tau}\triangleleft \xrightarrow{\texttt{move child 2}} \dot{e} : \triangleright\dot{\tau}\triangleleft} \tag{8b}$$

$$\frac{}{\triangleright\dot{e}\triangleleft : \dot{\tau} \xrightarrow{\texttt{move parent}} \triangleright\dot{e} : \dot{\tau}\triangleleft} \tag{8c}$$

$$\frac{}{\dot{e} : \triangleright\dot{\tau}\triangleleft \xrightarrow{\texttt{move parent}} \triangleright\dot{e} : \dot{\tau}\triangleleft} \tag{8d}$$

Lambda

$$\frac{}{\triangleright(\lambda x.\dot{e})\triangleleft \xrightarrow{\texttt{move child 1}} (\lambda x.\triangleright\dot{e}\triangleleft)} \tag{8e}$$

$$\frac{}{(\lambda x.\triangleright\dot{e}\triangleleft) \xrightarrow{\texttt{move parent}} \triangleright(\lambda x.\dot{e})\triangleleft} \tag{8f}$$

Application

$$\frac{}{\triangleright\dot{e}_1(\dot{e}_2)\triangleleft \xrightarrow{\texttt{move child 1}} \triangleright\dot{e}_1\triangleleft(\dot{e}_2)} \tag{8g}$$

$$\frac{}{\triangleright\dot{e}_1(\dot{e}_2)\triangleleft \xrightarrow{\texttt{move child 2}} \dot{e}_1(\triangleright\dot{e}_2\triangleleft)} \tag{8h}$$

$$\frac{}{\triangleright\dot{e}_1\triangleleft(\dot{e}_2) \xrightarrow{\texttt{move parent}} \triangleright\dot{e}_1(\dot{e}_2)\triangleleft} \tag{8i}$$

$$\frac{}{\dot{e}_1(\triangleright\dot{e}_2\triangleleft) \xrightarrow{\texttt{move parent}} \triangleright\dot{e}_1(\dot{e}_2)\triangleleft} \tag{8j}$$

Plus

$$\frac{}{\triangleright(\dot{e}_1 + \dot{e}_2)\triangleleft \xrightarrow{\texttt{move child 1}} (\triangleright\dot{e}_1\triangleleft + \dot{e}_2)} \tag{8k}$$

$$\frac{}{\triangleright(\dot{e}_1 + \dot{e}_2)\triangleleft \xrightarrow{\texttt{move child 2}} (\dot{e}_1 + \triangleright\dot{e}_2\triangleleft)} \tag{8l}$$

$$\frac{}{(\triangleright\dot{e}_1\triangleleft + \dot{e}_2) \xrightarrow{\texttt{move parent}} \triangleright(\dot{e}_1 + \dot{e}_2)\triangleleft} \tag{8m}$$

$$\frac{}{(\dot{e}_1 + \triangleright\dot{e}_2\triangleleft) \xrightarrow{\texttt{move parent}} \triangleright(\dot{e}_1 + \dot{e}_2)\triangleleft} \tag{8n}$$

Non-Empty Hole

$$\frac{}{\triangleright(\!|\dot{e}|\!)\triangleleft \xrightarrow{\texttt{move child 1}} (\!|\triangleright\dot{e}\triangleleft|\!)} \tag{8o}$$

$$\frac{}{(\!|\triangleright\dot{e}\triangleleft|\!) \xrightarrow{\texttt{move parent}} \triangleright(\!|\dot{e}|\!)\triangleleft} \tag{8p}$$

Again, additional rules are needed to recurse into the zipper structure, but we will define these zipper rules in an action-independent manner in Sec. 3.3.8.

The rules above are numerous and fairly uninteresting. That makes them quite hazardous – we might make a typo or forget a rule absent-mindedly. One check against this is to establish that movement actions do not change the cursor erasure, as in Theorem 2.

Theorem 2 (Movement Erasure Invariance)**.**

1. *If* $\hat{\tau} \xrightarrow{\texttt{move}\ \delta} \hat{\tau}'$ *then* $\hat{\tau}^{\diamond} = \hat{\tau}'^{\diamond}$.
2. *If* $\dot{\Gamma} \vdash \hat{e}^{\diamond} \Rightarrow \dot{\tau}$ *and* $\dot{\Gamma} \vdash \hat{e} \Rightarrow \dot{\tau} \xrightarrow{\texttt{move}\ \delta} \hat{e}' \Rightarrow \dot{\tau}'$ *then* $\hat{e}^{\diamond} = \hat{e}'^{\diamond}$ *and* $\dot{\tau} = \dot{\tau}'$.
3. *If* $\dot{\Gamma} \vdash \hat{e}^{\diamond} \Leftarrow \dot{\tau}$ *and* $\dot{\Gamma} \vdash \hat{e} \xrightarrow{\texttt{move}\ \delta} \hat{e}' \Leftarrow \dot{\tau}$ *then* $\hat{e}^{\diamond} = \hat{e}'^{\diamond}$.

Theorem 2 is useful also in that the relevant cases of Theorem 1 are straightforward by its application.

Another useful check is to establish *reachability*, i.e. that it is possible, through a sequence of movement actions, to move the cursor from any position to any other position within a well-typed H-expression.

This requires developing machinery for reasoning about sequences of actions. There are two possibilities: we can either add a sequencing action, $\alpha;\alpha$, directly to the syntax of actions, or we can define a syntax for lists of actions, $\bar{\alpha}$, together with iterated action judgements. To keep the core of the action semantics small, we take the latter approach in Figure 8.

A simple auxiliary judgement, $\bar{\alpha}$ movements (not shown) establishes that $\bar{\alpha}$ consists only of actions of the form `move` δ.

With these definitions, we can state reachability as follows:

Theorem 3 (Reachability)**.**

1. *If* $\hat{\tau}^{\diamond} = \hat{\tau}'^{\diamond}$ *then there exists some* $\bar{\alpha}$ *such that* $\bar{\alpha}$ movements *and* $\hat{\tau} \xrightarrow{\bar{\alpha}}{}^{*} \hat{\tau}'$.
2. *If* $\dot{\Gamma} \vdash \hat{e}^{\diamond} \Rightarrow \dot{\tau}$ *and* $\hat{e}^{\diamond} = \hat{e}'^{\diamond}$ *then there exists some* $\bar{\alpha}$ *such that* $\bar{\alpha}$ movements *and* $\dot{\Gamma} \vdash \hat{e} \Rightarrow \dot{\tau} \xrightarrow{\bar{\alpha}}{}^{*} \hat{e}' \Rightarrow \dot{\tau}$.
3. *If* $\dot{\Gamma} \vdash \hat{e}^{\diamond} \Leftarrow \dot{\tau}$ *and* $\hat{e}^{\diamond} = \hat{e}'^{\diamond}$ *then there exists some* $\bar{\alpha}$ *such that* $\bar{\alpha}$ movements *and* $\dot{\Gamma} \vdash \hat{e} \xrightarrow{\bar{\alpha}}{}^{*} \hat{e}' \Leftarrow \dot{\tau}$.

The simplest way to prove Theorem 3 is to break it into two lemmas. Lemma 4 establishes that you can always move the cursor to the outermost position in an expression. This serves as a check on our `move parent` rules.

Lemma 4 (Reach Up)**.**

1. *If* $\hat{\tau}^{\diamond} = \dot{\tau}$ *then there exists some* $\bar{\alpha}$ *such that* $\bar{\alpha}$ movements *and* $\hat{\tau} \xrightarrow{\bar{\alpha}}{}^{*} \triangleright\dot{\tau}\triangleleft$.
2. *If* $\dot{\Gamma} \vdash \dot{e} \Rightarrow \dot{\tau}$ *and* $\hat{e}^{\diamond} = \dot{e}$ *then there exists some* $\bar{\alpha}$ *such that* $\bar{\alpha}$ movements *and* $\dot{\Gamma} \vdash \hat{e} \Rightarrow \dot{\tau} \xrightarrow{\bar{\alpha}}{}^{*} \triangleright\dot{e}\triangleleft \Rightarrow \dot{\tau}$.
3. *If* $\dot{\Gamma} \vdash \dot{e} \Leftarrow \dot{\tau}$ *and* $\hat{e}^{\diamond} = \dot{e}$ *then there exists some* $\bar{\alpha}$ *such that* $\bar{\alpha}$ movements *and* $\dot{\Gamma} \vdash \hat{e} \xrightarrow{\bar{\alpha}}{}^{*} \triangleright\dot{e}\triangleleft \Leftarrow \dot{\tau}$.

Lemma 5 establishes that you can always move the cursor from the outermost position to any other position. This serves as a check on our `move child` n rules.

Lemma 5 (Reach Down)**.**

1. *If* $\hat{\tau}^{\diamond} = \dot{\tau}$ *then there exists some* $\bar{\alpha}$ *such that* $\bar{\alpha}$ movements *and* $\triangleright\dot{\tau}\triangleleft \xrightarrow{\bar{\alpha}}{}^{*} \hat{\tau}$.
2. *If* $\dot{\Gamma} \vdash \dot{e} \Rightarrow \dot{\tau}$ *and* $\hat{e}^{\diamond} = \dot{e}$ *then there exists some* $\bar{\alpha}$ *such that* $\bar{\alpha}$ movements *and* $\dot{\Gamma} \vdash \triangleright\dot{e}\triangleleft \Rightarrow \dot{\tau} \xrightarrow{\bar{\alpha}}{}^{*} \hat{e} \Rightarrow \dot{\tau}$.
3. *If* $\dot{\Gamma} \vdash \dot{e} \Leftarrow \dot{\tau}$ *and* $\hat{e}^{\diamond} = \dot{e}$ *then there exists some* $\bar{\alpha}$ *such that* $\bar{\alpha}$ movements *and* $\dot{\Gamma} \vdash \triangleright\dot{e}\triangleleft \xrightarrow{\bar{\alpha}}{}^{*} \hat{e} \Leftarrow \dot{\tau}$.

Theorem 3 follows by straightforward composition of these two lemmas. The proofs we give of these three theorems in the mechanization do not produce the shortest sequence of actions to witness reachability, which would resemble something like a lowest common ancestor computation. In future versions of Hazelnut that use the produced witnesses for automatic tool support it may make sense to engineer these proofs differently; here we are only concerned with whether the theorems are true.

3.3.5 Construction

The construction actions, `construct` ψ, are used to construct terms of a shape indicated by ψ at the cursor.

Types The `construct arrow` action constructs an arrow type. The H-type under the cursor becomes the argument type, and the cursor is placed on an empty return type hole:

$$\frac{}{\triangleright\dot{\tau}\triangleleft \xrightarrow{\texttt{construct arrow}} (\dot{\tau} \to \triangleright(\!|\!|\triangleleft)} \tag{12a}$$

ActionList $\bar{\alpha} ::= \cdot \mid \alpha;\bar{\alpha}$

$$\boxed{\hat{\tau} \xrightarrow{\bar{\alpha}}{}^{*} \hat{\tau}'} \qquad \frac{}{\hat{\tau} \xrightarrow{\cdot}{}^{*} \hat{\tau}} \quad (9a) \qquad \frac{\hat{\tau} \xrightarrow{\alpha} \hat{\tau}' \qquad \hat{\tau}' \xrightarrow{\bar{\alpha}}{}^{*} \hat{\tau}''}{\hat{\tau} \xrightarrow{\alpha;\bar{\alpha}}{}^{*} \hat{\tau}''} \quad (9b)$$

$$\boxed{\dot{\Gamma} \vdash \hat{e} \Rightarrow \dot{\tau} \xrightarrow{\bar{\alpha}}{}^{*} \hat{e}' \Rightarrow \dot{\tau}'} \qquad \frac{}{\dot{\Gamma} \vdash \hat{e} \Rightarrow \dot{\tau} \xrightarrow{\cdot}{}^{*} \hat{e} \Rightarrow \dot{\tau}} \quad \frac{\dot{\Gamma} \vdash \hat{e} \Rightarrow \dot{\tau} \xrightarrow{\alpha} \hat{e}' \Rightarrow \dot{\tau}' \qquad \dot{\Gamma} \vdash \hat{e}' \Rightarrow \dot{\tau}' \xrightarrow{\bar{\alpha}}{}^{*} \hat{e}'' \Rightarrow \dot{\tau}''}{\dot{\Gamma} \vdash \hat{e} \Rightarrow \dot{\tau} \xrightarrow{\alpha;\bar{\alpha}}{}^{*} \hat{e}'' \Rightarrow \dot{\tau}''} \quad (10a\text{-}b)$$

$$\boxed{\dot{\Gamma} \vdash \hat{e} \xrightarrow{\bar{\alpha}} \hat{e}' \Leftarrow \dot{\tau}} \qquad \frac{}{\dot{\Gamma} \vdash \hat{e} \xrightarrow{\cdot}{}^{*} \hat{e} \Leftarrow \dot{\tau}} \quad \frac{\dot{\Gamma} \vdash \hat{e} \xrightarrow{\alpha} \hat{e}' \Leftarrow \dot{\tau} \qquad \dot{\Gamma} \vdash \hat{e}' \xrightarrow{\bar{\alpha}}{}^{*} \hat{e}'' \Leftarrow \dot{\tau}}{\dot{\Gamma} \vdash \hat{e} \xrightarrow{\alpha;\bar{\alpha}}{}^{*} \hat{e}'' \Leftarrow \dot{\tau}} \quad (11a\text{-}b)$$

Figure 8. Iterated Action Judgements

This choice is formally arbitrary – it would have also been sensible to use the type under the cursor as the return type, for example. Indeed, we could consider defining both of these using different shapes. We avoid this for the sake of simplicity.

The `construct num` action replaces an empty type hole under the cursor with the `num` type:

$$\frac{}{\triangleright(\!|\!|\triangleleft \xrightarrow{\texttt{construct num}} \triangleright\texttt{num}\triangleleft} \tag{12b}$$

Ascription The `construct asc` action operates differently depending on whether the H-expression under the cursor synthesizes a type or is being analyzed against a type. In the first case, the synthesized type appears in the ascription:

$$\frac{}{\dot{\Gamma} \vdash \triangleright\dot{e}\triangleleft \Rightarrow \dot{\tau} \xrightarrow{\texttt{construct asc}} \dot{e} : \triangleright\dot{\tau}\triangleleft \Rightarrow \dot{\tau}} \tag{13a}$$

In the second case, the type provided for analysis appears in the ascription:

$$\frac{}{\dot{\Gamma} \vdash \triangleright\dot{e}\triangleleft \xrightarrow{\texttt{construct asc}} \dot{e} : \triangleright\dot{\tau}\triangleleft \Leftarrow \dot{\tau}} \tag{13b}$$

Variables The `construct var` x action places the variable x into an empty hole. If that hole is being asked to synthesize a type, then the result synthesizes the hypothesized type:

$$\frac{}{\dot{\Gamma}, x : \dot{\tau} \vdash \triangleright(\!|\!|\triangleleft \Rightarrow (\!|\!|) \xrightarrow{\texttt{construct var } x} \triangleright x\triangleleft \Rightarrow \dot{\tau}} \tag{13c}$$

If the hole is being analyzed against a type that is consistent with the hypothesized type, then the action semantics goes through the action subsumption rule described in Sec. 3.3.3. If the hole is being analyzed against a type that is inconsistent with the hypothesized type, x is placed inside a hole:

$$\frac{\dot{\tau} \nsim \dot{\tau}'}{\dot{\Gamma}, x : \dot{\tau}' \vdash \triangleright(\!|\!|\triangleleft \xrightarrow{\texttt{construct var } x} (\!|\triangleright x\triangleleft|\!) \Leftarrow \dot{\tau}} \tag{13d}$$

The rule above featured on Line 16 of Figure 2.

Lambdas The `construct lam` x action places a lambda abstraction binding x into an empty hole. If the empty hole is being asked to synthesize a type, then the result of the action is a lambda ascribed the type $((\!|\!|) \to (\!|\!|))$, with the cursor on the argument type hole (again, arbitrarily):

$$\frac{\alpha = \texttt{construct lam } x}{\dot{\Gamma} \vdash \triangleright(\!|\!|\triangleleft \Rightarrow (\!|\!|) \xrightarrow{\alpha} (\lambda x.(\!|\!|)) : (\triangleright(\!|\!|\triangleleft \to (\!|\!|)) \Rightarrow ((\!|\!|) \to (\!|\!|))} \tag{13e}$$

The type ascription is necessary because lambda expressions do not synthesize a type. (Type-annotated function definitions often arise as a single syntactic construct in full-scale languages like ML.)

If the empty hole is being analyzed against a type with matched arrow type, then no ascription is necessary:

$$\frac{\dot{\tau} \blacktriangleright_{\rightarrow} (\dot{\tau}_1 \rightarrow \dot{\tau}_2)}{\dot{\Gamma} \vdash \triangleright (\!|\,|\!) \triangleleft \xrightarrow{\texttt{construct lam } x} (\lambda x.\triangleright (\!|\,|\!) \triangleleft) \Leftarrow \dot{\tau}} \tag{13f}$$

Finally, if the empty hole is being analyzed against a type that has no matched arrow type, expressed in the premise as inconsistency with $(\!|\,|\!) \rightarrow (\!|\,|\!)$, then a lambda ascribed the type $(\!|\,|\!) \rightarrow (\!|\,|\!)$ is inserted inside a hole, which defers the type inconsistency as previously discussed:

$$\frac{\dot{\tau} \nsim (\!|\,|\!) \rightarrow (\!|\,|\!)}{\dot{\Gamma} \vdash \triangleright (\!|\,|\!) \triangleleft \xrightarrow{\texttt{construct lam } x} (\!|(\lambda x.(\!|\,|\!)) : (\triangleright (\!|\,|\!) \triangleleft \rightarrow (\!|\,|\!))|\!) \Leftarrow \dot{\tau}} \tag{13g}$$

Application The `construct ap` action applies the expression under the cursor. The following rule handles the case where the synthesized type has matched function type:

$$\frac{\dot{\tau} \blacktriangleright_{\rightarrow} (\dot{\tau}_1 \rightarrow \dot{\tau}_2)}{\dot{\Gamma} \vdash \triangleright \dot{e} \triangleleft \Rightarrow \dot{\tau} \xrightarrow{\texttt{construct ap}} \dot{e}(\triangleright (\!|\,|\!) \triangleleft) \Rightarrow \dot{\tau}_2} \tag{13h}$$

If the expression under the cursor synthesizes a type that is inconsistent with an arrow type, then we must place that expression inside a hole to maintain Theorem 3.1:

$$\frac{\dot{\tau} \nsim (\!|\,|\!) \rightarrow (\!|\,|\!)}{\dot{\Gamma} \vdash \triangleright \dot{e} \triangleleft \Rightarrow \dot{\tau} \xrightarrow{\texttt{construct ap}} (\!|\dot{e}|\!)(\triangleright (\!|\,|\!) \triangleleft) \Rightarrow (\!|\,|\!)} \tag{13i}$$

Numbers The `construct lit` n action replaces an empty hole with the number expression $\underline{n}$. If the empty hole is being asked to synthesize a type, then the rule is straightforward:

$$\frac{}{\dot{\Gamma} \vdash \triangleright (\!|\,|\!) \triangleleft \Rightarrow (\!|\,|\!) \xrightarrow{\texttt{construct lit } n} \triangleright \underline{n} \triangleleft \Rightarrow \texttt{num}} \tag{13j}$$

If the empty hole is being analyzed against a type that is inconsistent with `num`, then we must place the number expression inside the hole:

$$\frac{\dot{\tau} \nsim \texttt{num}}{\dot{\Gamma} \vdash \triangleright (\!|\,|\!) \triangleleft \xrightarrow{\texttt{construct lit } n} (\!|\triangleright \underline{n} \triangleleft|\!) \Leftarrow \dot{\tau}} \tag{13k}$$

The `construct plus` action constructs a plus expression with the expression under the cursor as its first argument (again, arbitrarily.) If that expression synthesizes a type consistent with `num`, then the rule is straightforward:

$$\frac{\dot{\tau} \sim \texttt{num}}{\dot{\Gamma} \vdash \triangleright \dot{e} \triangleleft \Rightarrow \dot{\tau} \xrightarrow{\texttt{construct plus}} (\dot{e} + \triangleright (\!|\,|\!) \triangleleft) \Rightarrow \texttt{num}} \tag{13l}$$

Otherwise, we must place that expression inside a hole:

$$\frac{\dot{\tau} \nsim \texttt{num}}{\dot{\Gamma} \vdash \triangleright \dot{e} \triangleleft \Rightarrow \dot{\tau} \xrightarrow{\texttt{construct plus}} (\dot{e}|\!) + \triangleright (\!|\,|\!) \triangleleft) \Rightarrow \texttt{num}} \tag{13m}$$

Non-Empty Holes The final shape is `nehole`. This explicitly places the expression under the cursor inside a hole:

$$\frac{}{\dot{\Gamma} \vdash \triangleright \dot{e} \triangleleft \Rightarrow \dot{\tau} \xrightarrow{\texttt{construct nehole}} (\!|\triangleright \dot{e} \triangleleft|\!) \Rightarrow (\!|\,|\!)} \tag{13n}$$

The `nehole` shape is grayed out in Figure 7 because we do not generally expect the programmer to perform it explicitly – other actions automatically insert holes when a type inconsistency would arise. The inclusion of this rule simplifies the statement of the constructability theorem, discussed next.

Constructability To check that we have defined "enough" construct actions, we need to establish that we can start from an empty hole and arrive at any well-typed expression with the cursor on the outside. This simpler statement is sufficient because Lemma 5 allows us to then move the cursor anywhere else inside the constructed term. As with reachability, we rely on the iterated action judgements defined in Figure 8.

Theorem 6 (Constructability)**.**

1. *For every $\dot{\tau}$ there exists $\bar{\alpha}$ such that $\triangleright (\!|\,|\!) \triangleleft \xrightarrow{\bar{\alpha}}{}^{*} \triangleright \dot{\tau} \triangleleft$.*
2. *If $\dot{\Gamma} \vdash \dot{e} \Rightarrow \dot{\tau}$ then there exists $\bar{\alpha}$ such that:*
$$\dot{\Gamma} \vdash \triangleright (\!|\,|\!) \triangleleft \Rightarrow (\!|\,|\!) \xrightarrow{\bar{\alpha}}{}^{*} \triangleright \dot{e} \triangleleft \Rightarrow \dot{\tau}$$
3. *If $\dot{\Gamma} \vdash \dot{e} \Leftarrow \dot{\tau}$ then there exists $\bar{\alpha}$ such that:*
$$\dot{\Gamma} \vdash \triangleright (\!|\,|\!) \triangleleft \xrightarrow{\bar{\alpha}}{}^{*} \triangleright \dot{e} \triangleleft \Leftarrow \dot{\tau}$$

Without the `nehole` shape, this theorem as stated would not hold. For example, it is not possible to construct well-typed H-expressions where non-empty holes appear superfluously without the `nehole` shape. Note also that although none of the shapes that we have defined can be dropped without losing this theorem, some construction rules could be dropped. In particular, rules that insert non-empty holes automatically could be dropped because the `construct nehole` action can always be used instead. We included them because we are interested in the mechanics of automatic non-empty hole insertion.

3.3.6 Deletion

The `del` action inserts an empty hole at the cursor, deleting what was there before.

The type action rule for `del` is self-explanatory:

$$\frac{}{\triangleright \dot{\tau} \triangleleft \xrightarrow{\texttt{del}} \triangleright (\!|\,|\!) \triangleleft} \tag{14}$$

Deletion within a Z-expression is similarly straightforward:

$$\frac{}{\dot{\Gamma} \vdash \triangleright \dot{e} \triangleleft \Rightarrow \dot{\tau} \xrightarrow{\texttt{del}} \triangleright (\!|\,|\!) \triangleleft \Rightarrow (\!|\,|\!)} \tag{15a}$$

$$\frac{}{\dot{\Gamma} \vdash \triangleright \dot{e} \triangleleft \xrightarrow{\texttt{del}} \triangleright (\!|\,|\!) \triangleleft \Leftarrow \dot{\tau}} \tag{15b}$$

Unlike the relative movement and construction actions, there is no "checksum" theorem for deletion. The rules do not inspect the structure of the expression in the cursor, so they both match our intuition and will be correct in any extension of the language without modification.

3.3.7 Finishing

The final action we need to consider is `finish`, which finishes the non-empty hole under the cursor.

If the non-empty hole appears in synthetic position, then it can always be finished:

$$\frac{\dot{\Gamma} \vdash \dot{e} \Rightarrow \dot{\tau}'}{\dot{\Gamma} \vdash \triangleright (\!|\dot{e}|\!) \triangleleft \Rightarrow (\!|\,|\!) \xrightarrow{\texttt{finish}} \triangleright \dot{e} \triangleleft \Rightarrow \dot{\tau}'} \tag{16a}$$

If the non-empty hole appears in analytic position, then it can only be finished if the type synthesized for the enveloped expression is consistent with the type that the hole is being analyzed against. This amounts to analyzing the enveloped expression against the provided type (by subsumption):

$$\frac{\dot{\Gamma} \vdash \dot{e} \Leftarrow \dot{\tau}}{\dot{\Gamma} \vdash \triangleright (\!|\dot{e}|\!) \triangleleft \xrightarrow{\texttt{finish}} \triangleright \dot{e} \triangleleft \Leftarrow \dot{\tau}} \tag{16b}$$

Like deletion, there is no need for a "checksum" theorem for the finishing action.

3.3.8 Zipper Cases

The rules defined so far handle the base cases, i.e. the cases where the action has "reached" the expression under the cursor. We also need to define the recursive cases, which propagate the action into the subtree where the cursor appears, as encoded by the zipper structure. For types, the zipper rules are straightforward:

$$\frac{\hat{\tau} \xrightarrow{\alpha} \hat{\tau}'}{(\hat{\tau} \to \dot{\tau}) \xrightarrow{\alpha} (\hat{\tau}' \to \dot{\tau})} \tag{17a}$$

$$\frac{\hat{\tau} \xrightarrow{\alpha} \hat{\tau}'}{(\dot{\tau} \to \hat{\tau}) \xrightarrow{\alpha} (\dot{\tau} \to \hat{\tau}')} \tag{17b}$$

For expressions, the zipper rules essentially follow the structure of the corresponding rules in the statics.

In particular, when the cursor is in the body of a lambda expression, the zipper case mirrors Rule (2a):

$$\frac{\dot{\tau} \blacktriangleright_{\to} (\dot{\tau}_1 \to \dot{\tau}_2) \qquad \dot{\Gamma}, x : \dot{\tau}_1 \vdash \hat{e} \xrightarrow{\alpha} \hat{e}' \Leftarrow \dot{\tau}_2}{\dot{\Gamma} \vdash (\lambda x.\hat{e}) \xrightarrow{\alpha} (\lambda x.\hat{e}') \Leftarrow \dot{\tau}} \tag{18a}$$

When the cursor is in the function position of an application, the rule mirrors Rule (1b):

$$\frac{\dot{\Gamma} \vdash \hat{e}^{\diamond} \Rightarrow \dot{\tau}_2 \qquad \dot{\Gamma} \vdash \hat{e} \Rightarrow \dot{\tau}_2 \xrightarrow{\alpha} \hat{e}' \Rightarrow \dot{\tau}_3 \qquad \dot{\tau}_3 \blacktriangleright_{\to} (\dot{\tau}_4 \to \dot{\tau}_5) \qquad \dot{\Gamma} \vdash \dot{e} \Leftarrow \dot{\tau}_4}{\dot{\Gamma} \vdash \hat{e}(\dot{e}) \Rightarrow \dot{\tau}_1 \xrightarrow{\alpha} \hat{e}'(\dot{e}) \Rightarrow \dot{\tau}_5} \tag{18b}$$

The situation is similar when the cursor is in argument position:

$$\frac{\dot{\Gamma} \vdash \dot{e} \Rightarrow \dot{\tau}_2 \qquad \dot{\tau}_2 \blacktriangleright_{\to} (\dot{\tau}_3 \to \dot{\tau}_4) \qquad \dot{\Gamma} \vdash \hat{e} \xrightarrow{\alpha} \hat{e}' \Leftarrow \dot{\tau}_3}{\dot{\Gamma} \vdash \dot{e}(\hat{e}) \Rightarrow \dot{\tau}_1 \xrightarrow{\alpha} \dot{e}(\hat{e}') \Rightarrow \dot{\tau}_4} \tag{18c}$$

The rules for the addition operator mirror Rule (1d):

$$\frac{\dot{\Gamma} \vdash \hat{e} \xrightarrow{\alpha} \hat{e}' \Leftarrow \texttt{num}}{\dot{\Gamma} \vdash (\hat{e} + \dot{e}) \Rightarrow \texttt{num} \xrightarrow{\alpha} (\hat{e}' + \dot{e}) \Rightarrow \texttt{num}} \tag{18d}$$

$$\frac{\dot{\Gamma} \vdash \hat{e} \xrightarrow{\alpha} \hat{e}' \Leftarrow \texttt{num}}{\dot{\Gamma} \vdash (\dot{e} + \hat{e}) \Rightarrow \texttt{num} \xrightarrow{\alpha} (\dot{e} + \hat{e}') \Rightarrow \texttt{num}} \tag{18e}$$

When the cursor is in the expression position of an ascription, we use the analytic judgement, mirroring Rule (1e):

$$\frac{\dot{\Gamma} \vdash \hat{e} \xrightarrow{\alpha} \hat{e}' \Leftarrow \dot{\tau}}{\dot{\Gamma} \vdash \hat{e} : \dot{\tau} \Rightarrow \dot{\tau} \xrightarrow{\alpha} \hat{e}' : \dot{\tau} \Rightarrow \dot{\tau}} \tag{18f}$$

When the cursor is in the type position of an ascription, we must re-check the ascribed expression because the cursor erasure might have changed (in practice, one would optimize this check to only occur if the cursor erasure did change):

$$\frac{\hat{\tau} \xrightarrow{\alpha} \hat{\tau}' \qquad \dot{\Gamma} \vdash \dot{e} \Leftarrow \hat{\tau}'^{\diamond}}{\dot{\Gamma} \vdash \dot{e} : \hat{\tau} \Rightarrow \hat{\tau}^{\diamond} \xrightarrow{\alpha} \dot{e} : \hat{\tau}' \Rightarrow \hat{\tau}'^{\diamond}} \tag{18g}$$

Finally, if the cursor is inside a non-empty hole, the relevant zipper rule mirrors Rule (1f):

$$\frac{\dot{\Gamma} \vdash \hat{e}^{\diamond} \Rightarrow \dot{\tau} \qquad \dot{\Gamma} \vdash \hat{e} \Rightarrow \dot{\tau} \xrightarrow{\alpha} \hat{e}' \Rightarrow \dot{\tau}'}{\dot{\Gamma} \vdash \langle\!|\hat{e}|\!\rangle \Rightarrow \langle\!|\,|\!\rangle \xrightarrow{\alpha} \langle\!|\hat{e}'|\!\rangle \Rightarrow \langle\!|\,|\!\rangle} \tag{18h}$$

Theorem 1 directly checks the correctness of these rules. Moreover, the zipper rules arise ubiquitously in derivations of edit steps, so the proofs of the other "check" theorems, e.g. Reachability and Constructability, serve as a check that none of these rules have been missed.

3.4 Determinism

A last useful property to consider is *action determinism*, i.e. that performing an action produces a unique result. The following theorem establishes determinism for type actions:

Theorem 7 (Type Action Determinism)**.** *If* $\hat{\tau} \xrightarrow{\alpha} \hat{\tau}'$ *and* $\hat{\tau} \xrightarrow{\alpha} \hat{\tau}''$ *then* $\hat{\tau}' = \hat{\tau}''$.

The corresponding theorem for expression actions would be stated as follows:

1. If $\dot{\Gamma} \vdash \hat{e}^{\diamond} \Rightarrow \dot{\tau}$ and $\dot{\Gamma} \vdash \hat{e} \Rightarrow \dot{\tau} \xrightarrow{\alpha} \hat{e}' \Rightarrow \dot{\tau}'$ and $\dot{\Gamma} \vdash \hat{e} \Rightarrow \dot{\tau} \xrightarrow{\alpha} \hat{e}'' \Rightarrow \dot{\tau}''$ then $\hat{e}' = \hat{e}''$ and $\dot{\tau}' = \dot{\tau}''$.
2. If $\dot{\Gamma} \vdash \hat{e}^{\diamond} \Leftarrow \dot{\tau}$ and $\dot{\Gamma} \vdash \hat{e} \xrightarrow{\alpha} \hat{e}' \Leftarrow \dot{\tau}$ and $\dot{\Gamma} \vdash \hat{e} \xrightarrow{\alpha} \hat{e}'' \Leftarrow \dot{\tau}$ then $\hat{e}' = \hat{e}''$.

This is not a theorem of the system as described so far. The reason is somewhat subtle: two construction actions, `construct asc` and `construct lam` x, behave differently in the analytic case than they do in the synthetic case. The problem is that both rules can "fire" when considering a Z-expression in analytic position due to action subsumption. This is a technically valid but "morally" invalid use of action subsumption: subsumption is included in the system to be used as a rule of last resort, i.e. it should only be applied when no other analytic action rule can fire.

There are several possible ways to address this problem. One approach would be to modify the judgement forms to internalize this notion of "rule of last resort". This approach is related to focusing from proof theory [55]. However, this approach would substantially complicate our presentation of the system.

The approach that we take leaves the system unchanged. Instead, we define predicates over *derivations* of the expression action judgements that exclude those derivations that apply subsumption prematurely, i.e. when another rule could have been applied. We call such derivations *subsumption-minimal derivations*. We can establish determinism for subsumption-minimal derivations.

Theorem 8 (Expression Action Determinism)**.**

1. *If* $\dot{\Gamma} \vdash \hat{e}^{\diamond} \Rightarrow \dot{\tau}$ *and* $\mathcal{D}_1 :: \dot{\Gamma} \vdash \hat{e} \Rightarrow \dot{\tau} \xrightarrow{\alpha} \hat{e}' \Rightarrow \dot{\tau}'$ *and* $\mathcal{D}_2 :: \dot{\Gamma} \vdash \hat{e} \Rightarrow \dot{\tau} \xrightarrow{\alpha} \hat{e}'' \Rightarrow \dot{\tau}''$ *and* $\mathsf{submin}_{\Rightarrow}(\mathcal{D}_1)$ *and* $\mathsf{submin}_{\Rightarrow}(\mathcal{D}_2)$ *then* $\hat{e}' = \hat{e}''$ *and* $\dot{\tau}' = \dot{\tau}''$.
2. *If* $\dot{\Gamma} \vdash \hat{e}^{\diamond} \Leftarrow \dot{\tau}$ *and* $\mathcal{D}_1 :: \dot{\Gamma} \vdash \hat{e} \xrightarrow{\alpha} \hat{e}' \Leftarrow \dot{\tau}$ *and* $\mathcal{D}_2 :: \dot{\Gamma} \vdash \hat{e} \xrightarrow{\alpha} \hat{e}'' \Leftarrow \dot{\tau}$ *and* $\mathsf{submin}_{\Leftarrow}(\mathcal{D}_1)$ *and* $\mathsf{submin}_{\Leftarrow}(\mathcal{D}_2)$ *then* $\hat{e}' = \hat{e}''$.

The mechanization, discussed next, defines the predicates $\mathsf{submin}_{\Rightarrow}(\mathcal{D})$ and $\mathsf{submin}_{\Leftarrow}(\mathcal{D})$. In addition, it defines a mapping from any derivation into a corresponding subsumption-minimal derivation. Implementations of Hazelnut need only implement this subsumption-minimal fragment.

3.5 Mechanization

In order to formally establish that our design meets our stated objectives, we have mechanized the semantics and metatheory of Hazelnut as described above using the Agda proof assistant [41] (also see the Agda Wiki, hosted at `http://wiki.portal.chalmers.se/agda/`.) This development is available in the supplemental material. The mechanization also includes the extension to Hazelnut described in Sec. 4.

The documentation includes a more detailed discussion of the technical representation decisions that we made. The main idea is standard: we encode each judgement as a dependent type. The rules defining the judgements become the constructors of this type, and derivations are terms of these type. This is a rich setting that allows proofs to take advantage of pattern matching on the shape of derivations, closely matching standard on-paper proofs. No proof automation was used, because the proof structure itself is likely to be interesting to researchers who plan to build upon our work.

We adopt Barendregt's convention for bound variables [61]. Hazelnut's semantics does not need substitution, so we do not need to adopt more sophisticated encodings (e.g. [32, 48].)

4. Extending Hazelnut

In this section, we will conservatively extend Hazelnut with binary sum types to demonstrate how the rules and the rich metatheory developed in the previous section serve to guide and constrain this and other such efforts.

Syntax. The first step is to extend the syntax of H-types and H-expressions with the familiar forms [22]:

$$\begin{array}{lrcl} \mathsf{HTyp} & \dot{\tau} & ::= & \cdots \mid (\dot{\tau} + \dot{\tau}) \\ \mathsf{HExp} & \dot{e} & ::= & \cdots \mid \mathtt{inj}_i(\dot{e}) \mid \mathtt{case}(\dot{e}, x.\dot{e}, y.\dot{e}) \end{array}$$

Recall that binary sum types introduce a new type-level connective, $(\dot{\tau}_1 + \dot{\tau}_2)$. The introductory forms are the *injections*, $\mathtt{inj}_i(\dot{e})$; here, we consider only binary sums, so $i \in \{\mathsf{L}, \mathsf{R}\}$. The elimination form is case analysis, $\mathtt{case}(\dot{e}, x.\dot{e}_1, y.\dot{e}_2)$.

Next, we must correspondingly extend the syntax of Z-types and Z-expressions, following Huet's zipper pattern [25]:

$$\begin{array}{lrcl} \mathsf{ZTyp} & \hat{\tau} & ::= & \cdots \mid (\hat{\tau} + \dot{\tau}) \mid (\dot{\tau} + \hat{\tau}) \\ \mathsf{ZExp} & \hat{e} & ::= & \cdots \mid \mathtt{inj}_i(\hat{e}) \mid \mathtt{case}(\hat{e}, x.\dot{e}, y.\dot{e}) \\ & & & \mid \mathtt{case}(\dot{e}, x.\hat{e}, y.\dot{e}) \mid \mathtt{case}(\dot{e}, x.\dot{e}, y.\hat{e}) \end{array}$$

Notice that for each H-type or H-expression form of arity n, there are n corresponding Z-type or Z-expression forms, each of which has a single "hatted" subterm. The remaining subterms are "dotted". We must also extend the definition of cursor erasure, e.g. for types:

$$(\hat{\tau} + \dot{\tau})^\diamond = (\hat{\tau}^\diamond + \dot{\tau})$$
$$(\dot{\tau} + \hat{\tau})^\diamond = (\dot{\tau} + \hat{\tau}^\diamond)$$

The rules for Z-expressions are analogous:

$$\mathtt{inj}_i(\hat{e})^\diamond = \mathtt{inj}_i(\hat{e}^\diamond)$$
$$\mathtt{case}(\hat{e}, x.\dot{e}_1, y.\dot{e}_2)^\diamond = \mathtt{case}(\hat{e}^\diamond, x.\dot{e}_1, y.\dot{e}_2)$$
$$\mathtt{case}(\dot{e}, x.\hat{e}_1, y.\dot{e}_2)^\diamond = \mathtt{case}(\dot{e}, x.\hat{e}_1^\diamond, y.\dot{e}_2)$$
$$\mathtt{case}(\dot{e}, x.\dot{e}_1, y.\hat{e}_2)^\diamond = \mathtt{case}(\dot{e}, x.\dot{e}_1, y.\hat{e}_2^\diamond)$$

Finally, we must extend the syntax of shapes:

$$\mathsf{Shape} \quad \psi ::= \cdots \mid \mathtt{sum} \mid \mathtt{inj}\ i \mid \mathtt{case}\ x\ y$$

Notice that for each H-type or H-expression form, there is a corresponding shape. The injection form had a formal parameter, i, so the corresponding shape takes a corresponding argument (like `lit` n.) The case form included two variable binders, so the corresponding shape takes two variable arguments (like `lam` x.)

Statics. We can now move on to the static semantics.

First, we must extend the type consistency relation as shown in Figure 9, following the example of covariant type consistency rule for arrow types in Figure 5. Similarly, we need a notion of a *matched sum type* analogous to the notion of a matched arrow type defined in Figure 5.

The type analysis rules shown in Figure 9 are essentially standard, differing only in that instead of immediately requiring that a type be of the form $(\dot{\tau}_1 + \dot{\tau}_2)$, we use the matched sum type judgement. We combine the two injection rules for concision and define only a type analysis rule for the case form for simplicity (see [11] for additional machinery that would be necessary for a synthetic rule.)

Action Semantics. Figures 10 and 11 extend Hazelnut's action semantics to support bidirectionally typed structure editing with sums.

Rule (22a), the construction rule for sum types, and Rules (22b)-(22c), the zipper rules for sum types, follow the corresponding rules for arrow types. Were we to have missed any of these, the first clause of Theorem 6, i.e. Constructability, would not be conserved.

$\boxed{\dot{\tau}_1 \sim \dot{\tau}_2}$

$$\frac{\dot{\tau}_1 \sim \dot{\tau}_1' \qquad \dot{\tau}_2 \sim \dot{\tau}_2'}{(\dot{\tau}_1 + \dot{\tau}_2) \sim \dot{\tau}_1' + \dot{\tau}_2'} \tag{19}$$

$\boxed{\dot{\tau} \blacktriangleright_+ \dot{\tau}_1 + \dot{\tau}_2}$ $\dot{\tau}$ has matched sum type $\dot{\tau}_1 + \dot{\tau}_2$

$$\frac{}{(\!|\,|\!) \blacktriangleright_+ (\!|\,|\!) + (\!|\,|\!)} \tag{20a} \qquad \frac{}{(\dot{\tau}_1 + \dot{\tau}_2) \blacktriangleright_+ (\dot{\tau}_1 + \dot{\tau}_2)} \tag{20b}$$

$\boxed{\dot{\Gamma} \vdash \dot{e} \Leftarrow \dot{\tau}}$

$$\frac{\dot{\tau}_+ \blacktriangleright_+ (\dot{\tau}_\mathsf{L} + \dot{\tau}_\mathsf{R}) \qquad \dot{\Gamma} \vdash \dot{e} \Leftarrow \dot{\tau}_i}{\dot{\Gamma} \vdash \mathtt{inj}_i(\dot{e}) \Leftarrow \dot{\tau}_+} \ (i \in \{\mathsf{L}, \mathsf{R}\}) \tag{21a}$$

$$\frac{\dot{\Gamma} \vdash \dot{e} \Rightarrow \dot{\tau}_+ \qquad \dot{\tau}_+ \blacktriangleright_+ (\dot{\tau}_1 + \dot{\tau}_2) \qquad \dot{\Gamma}, x : \dot{\tau}_1 \vdash \dot{e}_1 \Leftarrow \dot{\tau} \qquad \dot{\Gamma}, y : \dot{\tau}_2 \vdash \dot{e}_2 \Leftarrow \dot{\tau}}{\dot{\Gamma} \vdash \mathtt{case}(\dot{e}, x.\dot{e}_1, y.\dot{e}_2) \Leftarrow \dot{\tau}} \tag{21b}$$

Figure 9. The statics of sums.

$\boxed{\hat{\tau} \xrightarrow{\alpha} \hat{\tau}'}$

$$\frac{}{\triangleright \dot{\tau} \triangleleft \xrightarrow{\mathtt{construct\ sum}} (\dot{\tau} + \triangleright (\!|\,|\!) \triangleleft)} \tag{22a}$$

$$\frac{\hat{\tau} \xrightarrow{\alpha} \hat{\tau}'}{(\hat{\tau} + \dot{\tau}) \xrightarrow{\alpha} (\hat{\tau}' + \dot{\tau})} \tag{22b} \qquad \frac{\hat{\tau} \xrightarrow{\alpha} \hat{\tau}'}{(\dot{\tau} + \hat{\tau}) \xrightarrow{\alpha} (\dot{\tau} + \hat{\tau}')} \tag{22c}$$

$\boxed{\dot{\Gamma} \vdash \hat{e} \xrightarrow{\alpha} \hat{e}' \Leftarrow \dot{\tau}}$

$$\frac{\dot{\tau}_+ \blacktriangleright_+ (\dot{\tau}_1 + \dot{\tau}_2)}{\dot{\Gamma} \vdash \triangleright (\!|\,|\!) \triangleleft \xrightarrow{\mathtt{construct\ inj}\ i} \mathtt{inj}_i(\triangleright (\!|\,|\!) \triangleleft) \Leftarrow \dot{\tau}_+} \tag{23a}$$

$$\frac{\dot{\tau} \nsim (\!|\,|\!) + (\!|\,|\!))}{\dot{\Gamma} \vdash \triangleright (\!|\,|\!) \triangleleft \xrightarrow{\mathtt{construct\ inj}\ i} (\!|\mathtt{inj}_i((\!|\,|\!)) : (\triangleright (\!|\,|\!) \triangleleft + (\!|\,|\!))|\!) \Leftarrow \dot{\tau}} \tag{23b}$$

$$\frac{}{\dot{\Gamma} \vdash \triangleright (\!|\,|\!) \triangleleft \xrightarrow{\mathtt{construct\ case}\ x\ y} \mathtt{case}(\triangleright (\!|\,|\!) \triangleleft, x.(\!|\,|\!), y.(\!|\,|\!)) \Leftarrow \dot{\tau}} \tag{23c}$$

$$\frac{\dot{\tau}_+ \blacktriangleright_+ (\dot{\tau}_\mathsf{L} + \dot{\tau}_\mathsf{R}) \qquad \dot{\Gamma} \vdash \hat{e} \xrightarrow{\alpha} \hat{e}' \Leftarrow \dot{\tau}_i}{\dot{\Gamma} \vdash \mathtt{inj}_i(\hat{e}) \xrightarrow{\alpha} \mathtt{inj}_i(\hat{e}') \Leftarrow \dot{\tau}_+} \ (i \in \{\mathsf{L}, \mathsf{R}\}) \tag{23d}$$

$$\frac{\dot{\Gamma} \vdash \hat{e}^\diamond \Rightarrow \dot{\tau}_0 \qquad \dot{\Gamma} \vdash \hat{e} \Rightarrow \dot{\tau}_0 \xrightarrow{\alpha} \hat{e}' \Rightarrow \dot{\tau}_+ \qquad \dot{\tau}_+ \blacktriangleright_+ (\dot{\tau}_1 + \dot{\tau}_2) \qquad \dot{\Gamma}, x : \dot{\tau}_1 \vdash \dot{e}_1 \Leftarrow \dot{\tau} \qquad \dot{\Gamma}, y : \dot{\tau}_2 \vdash \dot{e}_2 \Leftarrow \dot{\tau}}{\dot{\Gamma} \vdash \mathtt{case}(\hat{e}, x.\dot{e}_1, y.\dot{e}_2) \xrightarrow{\alpha} \mathtt{case}(\hat{e}', x.\dot{e}_1, y.\dot{e}_2) \Leftarrow \dot{\tau}} \tag{23e}$$

$$\frac{\dot{\Gamma} \vdash \dot{e} \Rightarrow \dot{\tau}_+ \qquad \dot{\tau}_+ \blacktriangleright_+ (\dot{\tau}_1 + \dot{\tau}_2) \qquad \dot{\Gamma}, x : \dot{\tau}_1 \vdash \hat{e}_1 \xrightarrow{\alpha} \hat{e}_1' \Leftarrow \dot{\tau}}{\dot{\Gamma} \vdash \mathtt{case}(\dot{e}, x.\hat{e}_1, y.\dot{e}_2) \xrightarrow{\alpha} \mathtt{case}(\dot{e}, x.\hat{e}_1', y.\dot{e}_2) \Leftarrow \dot{\tau}} \tag{23f}$$

$$\frac{\dot{\Gamma} \vdash \dot{e} \Rightarrow \dot{\tau}_+ \qquad \dot{\tau}_+ \blacktriangleright_+ (\dot{\tau}_1 + \dot{\tau}_2) \qquad \dot{\Gamma}, y : \dot{\tau}_2 \vdash \hat{e}_2 \xrightarrow{\alpha} \hat{e}_2' \Leftarrow \dot{\tau}}{\dot{\Gamma} \vdash \mathtt{case}(\dot{e}, x.\dot{e}_1, y.\hat{e}_2) \xrightarrow{\alpha} \mathtt{case}(\dot{e}, x.\dot{e}_1, y.\hat{e}_2') \Leftarrow \dot{\tau}} \tag{23g}$$

$\boxed{\dot{\Gamma} \vdash \hat{e} \Rightarrow \dot{\tau} \xrightarrow{\alpha} \hat{e}' \Rightarrow \dot{\tau}'}$

$$\frac{\alpha = \mathtt{construct\ inj}\ i}{\dot{\Gamma} \vdash \triangleright (\!|\,|\!) \triangleleft \Rightarrow \dot{\tau} \xrightarrow{\alpha} \mathtt{inj}_i((\!|\,|\!)) : (\triangleright (\!|\,|\!) \triangleleft + (\!|\,|\!)) \Rightarrow (\!|\,|\!) + (\!|\,|\!)} \tag{24a}$$

$$\frac{\alpha = \mathtt{construct\ case}\ x\ y \qquad \dot{\tau} \blacktriangleright_+ (\dot{\tau}_1 + \dot{\tau}_2)}{\dot{\Gamma} \vdash \triangleright \dot{e} \triangleleft \Rightarrow \dot{\tau} \xrightarrow{\alpha} \mathtt{case}(\dot{e}, x.\triangleright (\!|\,|\!) \triangleleft, y.(\!|\,|\!)) : (\!|\,|\!) \Rightarrow (\!|\,|\!)} \tag{24b}$$

$$\frac{\alpha = \mathtt{construct\ case}\ x\ y \qquad \dot{\tau} \nsim (\!|\,|\!) + (\!|\,|\!))}{\dot{\Gamma} \vdash \triangleright \dot{e} \triangleleft \Rightarrow \dot{\tau} \xrightarrow{\alpha} \mathtt{case}((\!|\triangleright \dot{e} \triangleleft|\!), x.(\!|\,|\!), y.(\!|\,|\!)) : (\!|\,|\!) \Rightarrow (\!|\,|\!)} \tag{24c}$$

Figure 10. The construction & zipper action rules for sums.

$\hat{\tau} \xrightarrow{\text{move } \delta} \hat{\tau}'$ (25a-d)

$\triangleright(\dot{\tau}_1 + \dot{\tau}_2)\triangleleft \xrightarrow{\text{move child 1}} (\triangleright\dot{\tau}_1\triangleleft + \dot{\tau}_2)$

$\triangleright(\dot{\tau}_1 + \dot{\tau}_2)\triangleleft \xrightarrow{\text{move child 2}} (\dot{\tau}_1 + \triangleright\dot{\tau}_2\triangleleft)$

$(\triangleright\dot{\tau}_1\triangleleft + \dot{\tau}_2) \xrightarrow{\text{move parent}} \triangleright(\dot{\tau}_1 + \dot{\tau}_2)\triangleleft$

$(\dot{\tau}_1 + \triangleright\dot{\tau}_2\triangleleft) \xrightarrow{\text{move parent}} \triangleright(\dot{\tau}_1 + \dot{\tau}_2)\triangleleft$

$\hat{e} \xrightarrow{\text{move } \delta} \hat{e}'$ (26a-h)

$\triangleright\texttt{inj}_i(\dot{e})\triangleleft \xrightarrow{\text{move child 1}} \texttt{inj}_i(\triangleright\dot{e}\triangleleft)$

$\texttt{inj}_i(\triangleright\dot{e}\triangleleft) \xrightarrow{\text{move parent}} \triangleright\texttt{inj}_i(\dot{e})\triangleleft$

$\triangleright\texttt{case}(\dot{e}, x.\dot{e}_1, y.\dot{e}_2)\triangleleft \xrightarrow{\text{move child 1}} \texttt{case}(\triangleright\dot{e}\triangleleft, x.\dot{e}_1, y.\dot{e}_2)$

$\triangleright\texttt{case}(\dot{e}, x.\dot{e}_1, y.\dot{e}_2)\triangleleft \xrightarrow{\text{move child 2}} \texttt{case}(\dot{e}, x.\triangleright\dot{e}_1\triangleleft, y.\dot{e}_2)$

$\triangleright\texttt{case}(\dot{e}, x.\dot{e}_1, y.\dot{e}_2)\triangleleft \xrightarrow{\text{move child 3}} \texttt{case}(\dot{e}, x.\dot{e}_1, y.\triangleright\dot{e}_2\triangleleft)$

$\texttt{case}(\triangleright\dot{e}\triangleleft, x.\dot{e}_1, y.\dot{e}_2) \xrightarrow{\text{move parent}} \triangleright\texttt{case}(\dot{e}, x.\dot{e}_1, y.\dot{e}_2)\triangleleft$

$\texttt{case}(\dot{e}, x.\triangleright\dot{e}_1\triangleleft, y.\dot{e}_2) \xrightarrow{\text{move parent}} \triangleright\texttt{case}(\dot{e}, x.\dot{e}_1, y.\dot{e}_2)\triangleleft$

$\texttt{case}(\dot{e}, x.\dot{e}_1, y.\triangleright\dot{e}_2\triangleleft) \xrightarrow{\text{move parent}} \triangleright\texttt{case}(\dot{e}, x.\dot{e}_1, y.\dot{e}_2)\triangleleft$

Figure 11. Movement actions for sums.

Rule (23a) constructs an injection when the type provided for analysis has a matched sum type. This is analogous to Rule (13f) for lambdas. Rule (23b) constructs an injection when the type provided for analysis is not consistent with sum types. This is analogous to Rule (13g) for lambdas. Rule (23c) is a straightforward rule for constructing case expressions in empty holes. Rules (23d)-(23g) are the zipper cases, which follow the structure of the statics. Finally, we also define a single new synthetic action rule, Rule (24a), which allows for the construction of an injection in synthetic position, with automatic insertion of an ascription. This is analogous to Rule (13e). If we had defined any of these rules incorrectly, the Sensibility Theorem (Theorem 1) would not be conserved. Had we forgotten the analytic rules, the Constructability Theorem (Theorem 6) would not be conserved.

Figure 11 gives the relevant movement axioms. For concision and clarity, we write these axioms in tabular form. Had we made a mistake in any of these rules, the Movement Erasure Invariance theorem (Theorem 2) would not be conserved. Had we forgotten any of these rules, the Reachability Theorem (Theorem 3) would not be conserved.

5. Implementation

5.1 Implementation Concepts

Central to any implementation of Hazelnut is a stream of edit states whose cursor erasures synthesize types under an empty context according to the synthetic action judgement, $\emptyset \vdash \hat{e} \Rightarrow \dot{\tau} \xrightarrow{\alpha} \hat{e}' \Rightarrow \dot{\tau}'$. The middle row of Figure 12 diagrams this stream of edit states. For example, the reader is encouraged to re-examine the examples in Figure 1 and 2 – the cursor erasure of each edit state synthesizes a type.

Because Theorem 1 expresses an invariant, the editor does not need to typecheck the edit state anew on each action (though in some of the rules, e.g. Rule (18g) which handles the situation where the type in a type ascription changes, portions of the program would need to be typechecked again.) In other words, many of the problems related to incrementality are simply not relevant. Storing the types of subtrees would allow for further optimizations (e.g. of the zipper rules.)

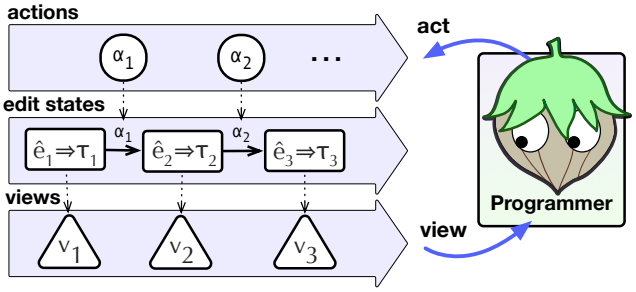

Figure 12. Implementation Concepts

The programmer examines a view generated from each edit state and produces actions in some implementation-defined manner (e.g. using a keyboard, mouse, touchscreen, voice interface, or neural interface), as diagrammed in Figure 12. Each new action causes a new abstract edit state to arise according to an implementation of the action semantics. This then causes a new view to arise. This is a simple event-based functional reactive programming model [66].

If an action is not well-defined according to Hazelnut's action semantics, the implementation must reject it. In fact, the implementation is encouraged to present an "action palette" that either hides or visibly disables actions that are not well-defined (see below.)

5.2 HZ

We have developed a simple reference implementation, HZ, of Hazelnut extended with sum types as described in Sec. 4. In order to reach a wide audience, we decided to implement HZ in the web browser. To take advantage of a mature implementation of the FRP model, we chose to implement HZ using OCaml[5], the OCaml React library[6], and the `js_of_ocaml` compiler and associated libraries [4][7].

The core semantics have been implemented in a functional style that follows the presentation in this paper closely.

The view computation renders the model (i.e. a Z-expression paired with a type) as nested HTML `div` elements matching the tree structure of the corresponding Z-expression. This tree is stylized separately using CSS. The action palette is a collection of buttons and text boxes which are disabled when the corresponding action cannot be performed. We determine this by simply attempting to perform the action internally and handling the exception that is raised when an action is undefined. (An optimization that is not in HZ would be to implement a version of the action semantics that simply computes a boolean, rather than the resulting edit state, for the purposes of action validation.) Each action form also has a corresponding keyboard shortcut. For actions that take arguments, the keyboard shortcut moves the cursor into a text box. The action validation occurs on every change to the text box.

This implementation is not, of course, meant to score marks for usability or performance (though both are surprisingly good for such a simple system.) Instead, as a simple reference implementation, it allows the reader to interact with the system presented in this paper, to aid in their understanding. Additionally, we expect its source code to be of use to others who are interested in layering a more fluid user interface atop the core semantics, or extensions thereof.

6. Related Work and Discussion

6.1 Structure Editors

Syntactic structure editors have a long history – the Cornell Program Synthesizer [59] was first introduced in 1981. Novice pro-

[5] `https://ocaml.org/`

[6] `http://erratique.ch/software/react`

[7] `http://ocsigen.org/js_of_ocaml/`

grammers have been a common target for structure editors. For example, GNOME [20] was developed to teach programming to undergraduates. Alice [12] is a 3-D programming language with an integrated structure editor for teaching novice CS undergraduate students. Scratch [51] is a structure editor targeted at children ages 8 to 16. TouchDevelop [60] incorporates a structure editor for programming on touch-based devices, and is used to teach high school students. An implementation of Hazelnut might be useful in teaching students about the typed lambda calculus, though that has not been our explicit aim with this work.

Not all structure editors are for educational purposes. For example, mbeddr [63] is a structure editor for a C-based programming language (nominally, for programming embedded systems.) Lamdu [33], like Hazelnut, is a structure editor for a statically typed functional language. It is designed for use by professional programmers.

The examples given so far either do not attempt to reason statically about types and binding, or do not attempt to maintain well-typedness as an edit invariant. This can pose problems, for reasons discussed in Sec. 1. One apparent exception is Unison [8], a structure editor for a typed functional language similar to Haskell. Like Hazelnut, it seems to define some notion of well-typedness for expressions with holes (though there is no technical documentation on virtually any aspect of its design.) Unlike Hazelnut, it does not have a notion analogous to Hazelnut's notion of a non-empty hole. As such, programmers must construct programs in a rigid outside-in manner, as discussed in Sec. 2. Another system with the same problem is Polymorphic Blocks, a block-based user interface where the structure of block connectors encodes a type [31].

We fundamentally differ from these projects in our design philosophy: we consider it essential to start by building type theoretic foundations, which are independent of nearly all decisions about the user interface (other than our choice to use an explicit cursor.) In contrast, these editors have developed innovative user interfaces (e.g. see the discussion in [64]) but lack a principled foundational calculus. In this respect, we follow the philosophical approach taken by languages that are rooted in the type theoretic tradition and have gone to great effort to develop a clear metatheory, like Standard ML [23, 29, 37]. In the future, we hope that these lines of research will merge to produce a human-usable typed structure editor with sound formal foundations. Our contribution, then, is in defining and analyzing the theoretical properties of a small foundational calculus that could be extended to achieve this vision. Our implementation resembles the minimal structure editor defined in Haskell by Sufrin and De Moor [58].

Some structure editor *generators* do operate on formal or semi-formal definitions of an underlying language. For example, the Synthesizer Generator [50] allows the user to define an attribute grammar-based language implementation that then can be used to generate a structured editor. CENTAUR [5] produces a language specific environment from a user defined formal specification of a language. Barista is a programmatic toolkit for building structure editors [28]. mbeddr is built on top of the commercial JetBrains MPS framework for constructing structure editors [62, 65]. These systems do not give a semantics to the edit states of the structure editor itself, or maintain non-trivial edit invariants, as Hazelnut does.

Related to structure editors are value editors, which operate directly on simple values (but not generally expressions or functions) of a programming language. For example, Eros is a typed value editor based in Haskell [17].

Other work has attempted to integrate structure editing features into text editors. For example, recent work has used syntactic placeholders reminiscent of our expression holes to decrease the percentage of edit states that are malformed [3]. This work does not consider the semantics of placeholders.

Prior work has also explored formal definitions of text editor commands, e.g. using functional combinators [57].

6.2 Gradual Type Systems

A significant contribution of this paper is the discovery of a clear technical relationship between typed structure editing and gradual typing. In particular, the machinery necessary to give a reasonable semantics to type holes – i.e. type consistency and type matching – coincides with that developed in the study of gradual type systems for functional languages. The pioneering work of Siek and Taha [54] introduced type consistency. Subsequent work developed the concept of type matching [18, 49] and has further studied the notion of type consistency [19]. In retrospect, this relationship is perhaps unsurprising: gradual typing is, notionally, also motivated by the idea of iterated development of a program where every intermediate state is well-defined in some sense, albeit at different granularity.

Recent work has discovered a systematic procedure for generating a "gradual version" of a standard type system [11]. This system, called the Gradualizer, operates on a logic program that specifies a simple type assignment system with some additional annotations to generate a corresponding specification of a gradual type system. The authors leave support for working with bidirectional type systems as future work. This suggests the possibility of an analogous "Editorializer" that generates a specification of a typed structure editor from a simple language definition. Our exposition in Sec. 4 certainly suggests that many of the necessary definitions follow seemingly mechanically from the definition of the static semantics, and the relationship with gradual typing suggests that many of the technical details of this transformation may already exist in the Gradualizer. One possibility we have explored informally is to use Agda's reflection features to implement such a system.

An aspect of gradual typing that we did not touch on directly here is its concern with assigning a dynamics to programs where type information is not known, by inserting dynamic type casts [54] or deducing evidence for consistency during evaluation [19]. This would correspond to assigning a dynamics to Hazelnut expressions with type holes such that a run-time error occurs when a type hole is found to be unfillable through evaluation. This may be useful as an exploratory programming tool.

6.3 Bidirectional Type Systems

Hazelnut is bidirectionally typed [9, 10, 14, 42, 47]. Bidirectional type systems are notable in that they are easy to define, easy to implement, produce simple error messages and support advanced language features [16]. For example, Scala [42] and Agda [41] are both fundamentally bidirectionally typed languages.

6.4 Type Reconstruction

An alternative approach to type inference is to use a unification-based type reconstruction system, as in functional languages like ML and Haskell [13]. This is difficult to reconcile with the approach presented in this paper, because edit actions could introduce new unification constraints that would require placing non-empty holes around terms far from the cursor. A whole-program hole insertion pass after each edit action could perhaps be used to recover invariants similar to those presented here, but we leave the details as future work. Our contention is that a bidirectional approach is a sweet spot in the design space of interactive systems like Hazelnut because it precludes "spooky errors at a distance". Instead, the interaction is a sort of local dialog between the programmer and the system involving simple, familiar concepts – types with holes – rather than sets of constraints.

An intermediate approach would be to layer unification-based type generation features atop the bidirectional system. This would amount to interpreting type holes as unification variables. For a simple calculus, e.g. the STLC upon which Hazelnut is based, type

inference for complete expressions is known to be decidable, so type holes could be instantiated automatically once the expression that they appear within has been constructed. It would also be possible to flag expressions for which there does not exist any way to fill the type holes. In more complex settings, e.g. in a dependently typed language, a partial decision procedure may still be useful in this regard, both at edit-time and (just prior to) run-time. Indeed, text editor modes for dependently typed proof assistants, e.g. for Agda, attempt to do exactly this for indicated "type holes" (and do not always succeed.)

6.5 Exceptions

Expression holes can be interpreted in several ways. One straightforward interpretation is to treat them like expressions that raise exceptions. Indeed, placing `raise Unimplemented` or similar in portions of an expression that are under construction is a common practice across programming languages today. The GHC dialect of Haskell recently introduced an explicit notion of a typed hole that behaves similarly [1].

6.6 Type-Directed Program Synthesis

Some text editor modes, e.g. those for proof assistants like Agda [41] and Idris [6], support a more explicit hole-based programming model where indicated expression holes are treated as sites where the system can be asked to automatically generate an expression of an appropriate type.

The Graphite system borrowed Eclipse's heuristic model of typed holes for Java to allow library providers to associate interactive code generation interfaces with types [43].

More generally, the topic of type-directed program synthesis an active area of research, e.g. [45]. By maintaining static well-definedness throughout the editing process, Hazelnut provides researchers interested in editor-integrated type-directed program synthesis with a formal foundation upon which to build.

6.7 Tactics

Interactive proof refinement systems, e.g. those in LCF [21], and more recent typed tactic systems, e.g. Mtac for Coq [68], support an explicit model of a "current" typed hole that serves as the target of program synthesis. Hazelnut differs in that edits can occur anywhere within a term.

A related approach is to interpret expression holes as the *metavariables* of contextual modal type theory (CMTT) [40]. In particular, expression holes have types and are surrounded by contexts, just as metavariables in CMTT are associated with types and contexts. This begins to clarify the logical meaning of a typing derivation in Hazelnut – it conveys well-typedness relative to an (implicit) modal context that extracts each expression hole's type and context. The modal context must be emptied – i.e. the expression holes must be instantiated with expressions of the proper type in the proper context – before the expression can be considered complete. This corresponds to the notion of modal necessity in contextual modal logic.

We did not make the modal context explicit in our semantics because interactive program editing is not merely hole filling in Hazelnut (i.e. the cursor need not be on a hole.) Moreover, the hole's type and context become apparent as our action semantics traverses the zipper structure on each step. For interactive proof assistants that support a tactic model based directly on hole filling, as just discussed, the connection to CMTT and similar systems is more useful. For example, Beluga [46] is based on dependent CMTT and aspects of Idris' editor support [6] are based on McBride's OLEG [35] and Lee and Friedman have explored a lambda calculus with contexts for a similar purpose [30].

One interesting avenue of future work is to elaborate expression holes to CMTT's closures, i.e. CMTT terms of the form $\mathsf{clo}(u; \mathrm{id}(\Gamma))$ where u is a unique metavariable associated with each hole and $\mathrm{id}(\Gamma)$ is the explicit identity substitution. This would allow us to evaluate expressions with holes such that the closure "accumulates" substitutions explicitly. When evaluation gets "stuck" (as it can, for CMTT does not define a dynamics equipped with a notion of progress under a non-empty modal context), it would then be possible for the programmer to choose a hole from the visible holes (which may have been duplicated) to edit in their original context. Once finished, the CMTT hole instantiation operation, together with a metatheorem that establishes that reduction commutes with instantiation, would enable an "edit and resume" feature with a clear formal basis. This notion of reduction commuting with instantiation has also been studied in other calculi [52]. Being able to edit a running program also has connections to less formal work on "live programming" interfaces [7, 33].

7. Conclusion

This paper presented Hazelnut, a type theoretic structure editor calculus. Our aim was to take a principled approach to its design by formally defining its static semantics as well as its action semantics and developing a rich metatheory. Moreover, we have mechanized substantial portions of the metatheory, including the crucial Sensibility theorem that establishes that every edit state is statically meaningful.

In addition to simplifying the job of an editor designer, typed structure editors also promise to increase the speed of development by eliminating redundant syntax and supporting higher-level primitive actions. However, we did not discuss such "edit costs" here, because they depend on particular implementation details, e.g. whether a keyboard or a mouse is in use. Indeed, we consider it a virtue of this work that such implementation details do not enter into our design.

7.1 Future Work

7.1.1 Richer Languages

Hazelnut is, obviously, a very limited language at its core. So the most obvious avenue for future work is to increase the expressive power of this language by extension. Our plan is to simultaneously maintain a mechanization and implementation (following, for example, Standard ML [29]) as we proceed, ultimately producing the first large-scale, formally verified bidirectionally typed structure editor.

It is interesting to note that the demarcation between the language and the editor is fuzzy (indeed, non-existent) in Hazelnut. There may well be interesting opportunities in language design when the language is being codesigned with a typed structure editor. It may be that certain language features are unnecessary given a sufficiently advanced type-aware structure editor (e.g. SML's `open`?), while other features may only be practical with editor support. We intend to use Hazelnut and derivative systems thereof as a platform for rigorously exploring such questions.

7.1.2 Evaluation Strategies: A High-Dimensional Space

The related work brought up in the previous section suggests three different evaluation strategies in the presence of type holes:

1. ...as preventing evaluation (the standard approach.)
2. ...as unknown types, in the gradual sense.
3. ...as unification variables.

In addition, we have discussed four different evaluation strategies in the presence of expression holes:

1. ...as preventing evaluation (the standard approach.)
2. ...as causing exceptions.
3. ...as sites for automatic program synthesis.
4. ...as the closures of CMTT.

Every combination of these choices could well be considered in the design of a full-scale programming system in the spirit of Hazelnut. Indeed, the user might be given several options from among these combinations, depending on their usage scenario. Many of these warrant further inquiry.

7.1.3 Editor Services

There are various aspects of the editor model that we have not yet formalized. For example, our action model does not consider how actions are actually entered using, for example, key combinations or chords. In practice, we would want also to suggest sensible compound actions, and to rank these suggested actions in some reasonable manner (perhaps based on usage data gathered from other users or code repositories.) Designing a action suggestion semantics and a rigorous typed probability model over actions is one avenue of research that we have started to explore, with intriguing initial results.

7.1.4 Programmable Actions

Our language of actions is intentionally primitive. However, even now it acts much like a simple imperative command language. This suggests future expansion to, for example, a true *action macro* language, whereby functional programs could themselves be lifted to the level of actions to compute non-trivial compound actions. Such compound actions would give a uniform description of transformations ranging from the simple – like "move the cursor to the next hole to the right" – to quite complex whole program refactoring, while remaining subject to the core Hazelnut metatheory. Techniques like those in advanced tactic systems, e.g. Mtac, might be useful in proving these action macros correct [68].

7.1.5 Views

Another research direction is in exploring how types can be used to control the presentation of expressions in the editor. For example, following an approach developed in a textual setting of developing *type-specific languages* (TSLs), it should be possible to have the type that an expression is being analyzed against define alternative display forms and interaction modes [44].

It should also be possible to develop the concept of semantic comments, i.e. comments that mention semantic structures or even contain values. These would be subject to the same operations, e.g. renaming, as other structures, helping to address the problem of comments becoming inconsistent with code. This system, generalized sufficiently, could one day help unify document editing with program editing.

7.1.6 Collaborative Programming

Finally, we did not consider any aspects of *collaborative programming*, such as a packaging system, a differencing algorithm for use in a source control system, support for multiple simultaneous cursors for different users, and so on. These are all interesting avenues for future work.

7.1.7 Empirical Evaluation

Although we make few empirical claims in this paper, it is ultimately an empirical question as to whether structure editors, and typed structure editors, are practical. We hope to conduct user studies once a richer semantics and a practical implementation thereof has been developed.

7.1.8 More Theory

Connections with gradual type systems and CMTT, discussed in the previous section, seem likely to continue to be revealing.

The notion of having one of many possible locations within a term under a cursor has a very strong intuitive connection to the proof theoretic notion of focusing [55]. Building closer connections with proof theory (and category theory) is likely to be a fruitful avenue of further inquiry.

> *In any case, these are but steps toward more graphical program-description systems, for we will not forever stay confined to mere strings of symbols.*
>
> — Marvin Minsky, Turing Award lecture [38]

Acknowledgments

The authors would like to thank the anonymous referees at POPL and TFP 2016 for useful feedback on earlier drafts of this paper; Ed Morehouse and Carlo Angiuli for counsel on mechanization and Barendrecht's Convention; Vincent Zeng for *pro bono* artistic services; and YoungSeok Yoon for reanalyzing the data from [67]. This work was partially funded through a gift from Mozilla; the NSF grant #CCF-1619282, 1553741 and 1439957; by AFRL and DARPA under agreement #FA8750-16-2-0042; and by NSA lablet contract #H98230-14-C-0140.

References

[1] Typed holes in GHC. `https://wiki.haskell.org/GHC/Typed_holes`. Retrieved Nov. 7, 2016.

[2] A. Altadmri and N. C. Brown. 37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data. In *ACM Technical Symposium on Computer Science Education (SIGCSE)*, 2015.

[3] L. E. d. S. Amorim, S. Erdweg, G. Wachsmuth, and E. Visser. Principled syntactic code completion using placeholders. In *SLE*, 2016.

[4] V. Balat. Ocsigen: Typing Web Interaction with Objective Caml. In *ACM Workshop on ML*, 2006.

[5] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: The System. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 14–24, 1988.

[6] E. Brady. Idris, A General-Purpose Dependently Typed Programming Language: Design and Implementation. *Journal of Functional Programming*, 23(05):552–593, 2013.

[7] S. Burckhardt, M. Fähndrich, P. de Halleux, S. McDirmid, M. Moskal, N. Tillmann, and J. Kato. It's alive! continuous feedback in UI programming. In *PLDI*, 2013.

[8] P. Chiusano. Unison. `http://www.unisonweb.org/`. Accessed: 2016-04-25.

[9] A. Chlipala, L. Petersen, and R. Harper. Strict bidirectional type checking. In *ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI)*, 2005.

[10] D. R. Christiansen. Bidirectional Typing Rules: A Tutorial. `http://davidchristiansen.dk/tutorials/bidirectional.pdf`, 2013.

[11] M. Cimini and J. G. Siek. The gradualizer: a methodology and algorithm for generating gradual type systems. In *POPL*, 2016.

[12] M. Conway, S. Audia, T. Burnette, D. Cosgrove, and K. Christiansen. Alice: Lessons Learned from Building a 3D System for Novices. In *SIGCHI Conference on Human Factors in Computing Systems (CHI)*, 2000.

[13] L. Damas and R. Milner. Principal type-schemes for functional programs. In *POPL*, 1982.

[14] R. Davies and F. Pfenning. Intersection types and computational effects. In *ICFP*, 2000.

[15] M. de Jonge, E. Nilsson-Nyman, L. C. L. Kats, and E. Visser. Natural and flexible error recovery for generated parsers. In *SLE*, 2009.

[16] J. Dunfield and N. R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *ICFP*, 2013.

[17] C. Elliott. Tangible Functional Programming. In *ICFP*, 2007.

[18] R. Garcia and M. Cimini. Principal Type Schemes for Gradual Programs. In *POPL*, 2015.

[19] R. Garcia, A. M. Clark, and E. Tanter. Abstracting gradual typing. In *POPL*, 2016.

[20] D. B. Garlan and P. L. Miller. GNOME: An introductory programming environment based on a family of structure editors. In *First ACM*

SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 1984.

[21] M. Gordon, R. Milner, L. Morris, M. Newey, and C. Wadsworth. A metalanguage for interactive proof in LCF. In *POPL*, 1978.

[22] R. Harper. *Practical Foundations for Programming Languages*. 2nd edition, 2016. URL `https://www.cs.cmu.edu/~rwh/plbook/2nded.pdf`.

[23] R. Harper and C. Stone. A Type-Theoretic Interpretation of Standard ML. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[24] G. J. Holzmann. Brace yourself. *IEEE Software*, 33(5):34–37, Sept 2016. ISSN 0740-7459. doi: 10.1109/MS.2016.123.

[25] G. Huet. The Zipper. *Journal of Functional Programming*, 7(5), Sept. 1997. Functional Pearl.

[26] D. Jones. Developer beliefs about binary operator precedence. *C Vu*, 18(4):14–21, 2006.

[27] L. C. L. Kats, M. de Jonge, E. Nilsson-Nyman, and E. Visser. Providing rapid feedback in generated modular language environments: adding error recovery to scannerless generalized-LR parsing. In *OOPSLA*, 2009.

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

[29] D. K. Lee, K. Crary, and R. Harper. Towards a mechanized metatheory of Standard ML. In *POPL*, 2007.

[30] S. Lee and D. P. Friedman. Enriching the Lambda Calculus with Contexts: Toward a Theory of Incremental Program Construction. In *ICFP*, 1996.

[31] S. Lerner, S. R. Foster, and W. G. Griswold. Polymorphic blocks: Formalism-inspired UI for structured connectors. In *ACM Conference on Human Factors in Computing Systems (CHI)*, 2015.

[32] D. R. Licata and R. Harper. A Universe of Binding and Computation. In *ICFP*, 2009.

[33] E. Lotem and Y. Chuchem. Project Lamdu. `http://www.lamdu.org/`. Accessed: 2016-04-08.

[34] G. Marceau, K. Fisler, and S. Krishnamurthi. Do values grow on trees?: Expression integrity in functional programming. In *Seventh International Workshop on Computing Education Research (ICER)*, 2011.

[35] C. McBride. *Dependently typed functional programs and their proofs*. PhD thesis, University of Edinburgh. College of Science and Engineering. School of Informatics., 2000.

[36] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

[37] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[38] M. Minsky. Form and content in computer science (1970 ACM Turing Lecture). *J. ACM*, 17(2):197–215, 1970.

[39] M. Mooty, A. Faulring, J. Stylos, and B. A. Myers. Calcite: Completing code completion for constructors using crowds. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2010.

[40] A. Nanevski, F. Pfenning, and B. Pientka. Contextual modal type theory. *ACM Trans. Comput. Log.*, 9(3), 2008.

[41] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[42] M. Odersky, C. Zenger, and M. Zenger. Colored local type inference. In *POPL*, 2001.

[43] C. Omar, Y. Yoon, T. D. LaToza, and B. A. Myers. Active code completion. In *ICSE*, 2012.

[44] C. Omar, D. Kurilova, L. Nistor, B. Chung, A. Potanin, and J. Aldrich. Safely composable type-specific languages. In *ECOOP*, 2014.

[45] P. Osera and S. Zdancewic. Type-and-example-directed program synthesis. In *PLDI*, 2015.

[46] B. Pientka. Beluga: Programming with dependent types, contextual data, and contexts. In *International Symposium on Functional and Logic Programming (FLOPS)*, 2010.

[47] B. C. Pierce and D. N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, Jan. 2000.

[48] N. Pouillard. Nameless, painless. In *ICFP*, 2011.

[49] A. Rastogi, A. Chaudhuri, and B. Hosmer. The ins and outs of gradual type inference. In *POPL*, 2012.

[50] T. Reps and T. Teitelbaum. The synthesizer generator. *SIGSOFT Softw. Eng. Notes*, 9(3):42–48, Apr. 1984. ISSN 0163-5948.

[51] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai. Scratch: Programming for All. *Commun. ACM*, 52(11):60–67, Nov. 2009.

[52] D. Sands. Computing with contexts: A simple approach. *Electr. Notes Theor. Comput. Sci.*, 10:134–149, 1997.

[53] A. Sarkar. The impact of syntax colouring on program comprehension. In *Annual Conference of the Psychology of Programming Interest Group (PPIG)*, 2015.

[54] J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, 2006.

[55] R. J. Simmons and F. Pfenning. Weak Focusing for Ordered Linear Logic. Technical Report CMU-CS-10-147, Carnegie Mellon University, 2011. Revision of April 2011.

[56] A. Stefik and S. Siebert. An empirical investigation into programming language syntax. *ACM Transactions on Computing Education (TOCE)*, 13(4):19, 2013.

[57] B. Sufrin. Formal specification of a display-oriented text editor. *Sci. Comput. Program.*, 1(3):157–202, 1982.

[58] B. Sufrin and O. De Moor. Modeless structure editing. In *Proceedings of the Oxford-Microsoft Symposium in Celebration of the work of Tony Hoare*, 1999.

[59] T. Teitelbaum and T. Reps. The Cornell Program Synthesizer: A Syntax-directed Programming Environment. *Commun. ACM*, 24(9):563–573, 1981.

[60] N. Tillmann, M. Moskal, J. de Halleux, and M. Fahndrich. TouchDevelop: Programming Cloud-connected Mobile Devices via Touchscreen. In *SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2011.

[61] C. Urban, S. Berghofer, and M. Norrish. Barendregt's variable convention in rule inductions. In *Conference on Automated Deduction (CADE)*, 2007.

[62] M. Voelter. Language and IDE Modularization and Composition with MPS. In *International Summer School on Generative and Transformational Techniques in Software Engineering*, pages 383–430. Springer, 2011.

[63] M. Voelter, D. Ratiu, B. Schaetz, and B. Kolb. Mbeddr: An Extensible C-based Programming Language and IDE for Embedded Systems. In *SPLASH*, 2012.

[64] M. Voelter, J. Siegmund, T. Berger, and B. Kolb. Towards User-Friendly Projectional Editors. In *International Conference on Software Language Engineering (SLE)*, 2014.

[65] M. Voelter, J. Warmer, and B. Kolb. Projecting a Modular Future. *IEEE Software*, 32(5):46–52, 2015.

[66] Z. Wan and P. Hudak. Functional Reactive Programming from First Principles. In *PLDI*, 2000.

[67] Y. S. Yoon and B. A. Myers. A longitudinal study of programmers' backtracking. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2014.

[68] B. Ziliani, D. Dreyer, N. R. Krishnaswami, A. Nanevski, and V. Vafeiadis. Mtac: A monad for typed tactic programming in Coq. *Journal of Functional Programming*, 25:e12, 2015.