Hazelnut: A Bidirectionally Typed Structure Editor Calculus

Abstract

Programs are rich inductive structures, but programmers typically construct and manipulate them indirectly, through flat textual representations. This indirection comes at a cost: programmers must understand the subtleties of parsing, and it can require several text editor actions to make a single syntactically and semantically well-defined change. During these sequences of edit actions, or when the programmer makes a mistake, programmers and programming tools must contend with malformed text or semantically ill-defined programs.

Structure editors promise to alleviate these burdens by exposing only edit actions that produce sensible changes to the program structure. Existing designs for structure editors, however, are complex and somewhat *ad hoc*. They also focus primarily on syntactic well-formedness, so programs can still be left semantically ill-defined as they are being constructed. In this paper, we report on our ongoing efforts to develop Hazelnut, a minimal structure editor defined in a principled type-theoretic style where all edit actions leave the program in both a syntactically and semantically well-defined state. Uniquely, Hazelnut does not force the programmer to construct the program in a strictly "outside-in" fashion. Formally, Hazelnut is a bidirectionally typed lambda calculus extended with 1) *holes* (which mark subterms that are being constructed from the inside out); 2) a *focus model*; and 3) a bidirectional *action model* equipped with a useful *action sensibility* theorem.

1. Introduction

Programmers and the tools that they use are often presented with text that does not correspond to a well-defined program. This may be because the programmer is in the midst of a sequence of text edit actions that leave the program "temporarily" malformed or ill-defined, or because the programmer has made a mistake. This complicates the programming process, both because the language definition provides no reasoning principles relevant to such text, and, relatedly, because the editor can no longer provide useful services (e.g. syntax highlighting, or semantics-aware code completion.) Source code editors sometimes develop *ad hoc* workarounds for this problem, e.g. they might use whitespace to guess where a construct is likely to end, or recover from a type error by pretending that the type was as expected (if, indeed, an expected type can be determined.)

Structure editors have made some progress toward addressing this problem by allowing programmers to edit the tree structure of a program directly. Each edit action leaves the program being constructed in a structurally well-formed state.[1] We give examples of notable structure editors in Sec. 6.

The problem is that it is still possible to leave the program in a semantically meaningless (i.e. undefined) state, because language definitions generally only give meaning to complete, well-typed terms. This makes it difficult for humans and tools to reason about types and binding during the development process.

In this paper, we present our ongoing work on a minimal structure editor, Hazelnut, defined in the type-theoretic style (i.e. Hazelnut is a *structure editor calculus*.) In Hazelnut, expressions and types with *holes* have a well-defined static semantics. Edit actions are type-aware and leave the program in both a structurally and semantically well-defined state. In fact, edit actions maintain an even stronger invariant – when acting on an expression whose type is determined by its surroundings (e.g. an expression in function argument position), only edit actions consistent with that type are permitted (we will formally state this invariant in Sec. 3.3.) This does not imply that programs need to be constructed in a strictly outside-in manner, however, because an expression that has a type that is not yet consistent with its surroundings will automatically be put into a hole that defers consistency analysis until the hole is *finished*.

The remainder of the paper is organized as follows:

- We begin with an example to develop the reader's intuitions in Section 2.
- We then formally define Hazelnut in Section 3 and state the important metatheoretic properties.
- In Section 4, we describe our ongoing effort to formalize the semantics and metatheory of Hazelnut in Agda.
- In Section 5, we describe our ongoing effort to implement Hazelnut in a web browser, using a functional reactive model for user interaction.
- In Section 6, we give an overview of related work.
- We conclude in Section 7 by describing our vision for this work going forward.

[1] In addition to eliminating malformed edit states, it is also generally the case that programs can be written more quickly using a structure editor. However, we will not talk about "edit costs" here, because we would like to abstract away from details like whether a keyboard or a mouse (or some other input device) is being used.

#	H-Expression	Z-Expression	Next Action	Semantics
1	$\llparenthesis\rrparenthesis$	$\triangleright\llparenthesis\rrparenthesis\triangleleft$	`construct lam` x	Rule (19e)
2	$\lambda x.\llparenthesis\rrparenthesis : \llparenthesis\rrparenthesis \to \llparenthesis\rrparenthesis$	$\lambda x.\llparenthesis\rrparenthesis : \triangleright\llparenthesis\rrparenthesis\triangleleft \to \llparenthesis\rrparenthesis$	`construct num`	Rule (18b)
3	$\lambda x.\llparenthesis\rrparenthesis : \texttt{num} \to \llparenthesis\rrparenthesis$	$\lambda x.\llparenthesis\rrparenthesis : \triangleright\texttt{num}\triangleleft \to \llparenthesis\rrparenthesis$	`move nextSib`	Rule (15d)
4		$\lambda x.\llparenthesis\rrparenthesis : \texttt{num} \to \triangleright\llparenthesis\rrparenthesis\triangleleft$	`construct num`	Rule (18b)
5	$\lambda x.\llparenthesis\rrparenthesis : \texttt{num} \to \texttt{num}$	$\lambda x.\llparenthesis\rrparenthesis : \texttt{num} \to \triangleright\texttt{num}\triangleleft$	`move parent`	Rule (15b)
6		$\lambda x.\llparenthesis\rrparenthesis : \triangleright\texttt{num} \to \texttt{num}\triangleleft$	`move prevSib`	Rule (15e)
7		$\triangleright\lambda x.\llparenthesis\rrparenthesis\triangleleft : \texttt{num} \to \texttt{num}$	`move firstChild`	Rule (**??**)
8		$\lambda x.\triangleright\llparenthesis\rrparenthesis\triangleleft : \texttt{num} \to \texttt{num}$	`construct var` x	Rule (19c)
9	$\lambda x.x : \texttt{num} \to \texttt{num}$	$\lambda x.\triangleright x\triangleleft : \texttt{num} \to \texttt{num}$	——	——
... now assume a context where $id : \texttt{num} \to \texttt{num}$ ...				
10	$\llparenthesis\rrparenthesis$	$\triangleright\llparenthesis\rrparenthesis\triangleleft$	`construct asc`	Rule (19a)
11	$\llparenthesis\rrparenthesis : \llparenthesis\rrparenthesis$	$\llparenthesis\rrparenthesis : \triangleright\llparenthesis\rrparenthesis\triangleleft$	`construct num`	Rule (18b)
12	$\llparenthesis\rrparenthesis : \texttt{num}$	$\llparenthesis\rrparenthesis : \triangleright\texttt{num}\triangleleft$	`move prevSib`	Rule (15e)
13		$\triangleright\llparenthesis\rrparenthesis\triangleleft : \texttt{num}$	`construct var` id	Rule (19d)
14	$\llparenthesis id\rrparenthesis : \texttt{num}$	$\llparenthesis\triangleright id\triangleleft\rrparenthesis : \texttt{num}$	`construct ap`	Rule (19h)
15	$\llparenthesis id(\llparenthesis\rrparenthesis)\rrparenthesis : \texttt{num}$	$\llparenthesis id(\triangleright\llparenthesis\rrparenthesis\triangleleft)\rrparenthesis : \texttt{num}$	`construct numlit 3`	Rule (19l)
16	$\llparenthesis id(\underline{3})\rrparenthesis : \texttt{num}$	$\llparenthesis id(\triangleright\underline{3}\triangleleft)\rrparenthesis : \texttt{num}$	`move parent`	Rule (**??**)
17		$\llparenthesis\triangleright id(\underline{3})\triangleleft\rrparenthesis : \texttt{num}$	`move parent`	Rule (**??**)
18		$\triangleright\llparenthesis id(\underline{3})\rrparenthesis\triangleleft : \texttt{num}$	`finish`	Rule (20a)
19	$id(\underline{3}) : \texttt{num}$	$\triangleright id(\underline{3})\triangleleft : \texttt{num}$	——	——

Figure 1. Constructing an identity function in Hazelnut (Lines 1–9), then applying this function (assumed bound to *id*, not shown) to an argument (Lines 10–19). The formal syntax and referenced rules in the final column are described in Section 3.

2. Programming in Hazelnut

Figure 1 gives an example of the Hazelnut user performing two simple programming tasks. The syntactic forms in this figure will be formally defined in Sec. 3. For now, we will develop only the necessary intuitions. In the first task (Lines 1-9), the user constructs the identity function over numbers. In the second task (Lines 10-19), the user applies this function (assumed to be bound to a variable, *id*), to the number expression $\underline{3}$. Each of these tasks is carried out interactively, through the sequence of *actions* shown in the column labeled **Next Action**. For reference, we cite the relevant rules from Sec. 3 in the final column.

The second and third columns of the table show the program as it is being constructed in two forms. The second column shows it as an **H-expression**, which is an expression that can contain *holes*, delimited by $\llparenthesis$ and $\rrparenthesis$. The third column shows a corresponding **Z-expression**. Z-expressions are H-expressions with a single focus on some sub-term, delimited by $\triangleright$ and $\triangleleft$. The focus need not be on a hole. Each action produces a new Z-expression, but this may or may not correspond to a new H-expression (in particular, some actions only move the focus, without changing the structure of the term.)

Line 1 begins with the simplest initial expression: an H-expression consisting of a single hole. The corresponding Z-Expression has that hole in focus, indicated by the syntax $\triangleright\llparenthesis\rrparenthesis\triangleleft$. Focus determines the locus of action. The first action the user performs is `construct lam` x, which replaces the hole with a lambda abstraction binding the variable x. This results in the program on line 2, consisting of a lambda abstraction ascribed an arrow type with holes in all positions. The argument type hole is in focus. The user proceeds to fill these holes using construction and movement actions, resulting in the final expression on Line 9. With no holes remaining, this expression is *complete*.

So far, editing has proceeded in an essentially type-directed, outside-in fashion – the user first specified the type of the function, then produced a body of that type by the action on Line 8. Lines 10-12 similarly begin in a type-directed manner with the user giving an explicit type ascription, indicating that the expression that they are constructing will have type `num`.

However, on Line 13, the user performs the `construct var` *id* action. Notice that *id* has type $\texttt{num} \to \texttt{num}$, which is not consistent with the type `num` given in the ascription. Naïvely, this would produce a type error, leaving the program in a well-formed but semantically undefined state. One way to avoid this state is to simply not make this action available in the program configuration on Line 12. This is inflexible, forcing an outside-in approach to program construction (i.e. the user would need to construct the function application form before constructing the variable *id*.) Instead, Hazelnut permits this action, but places the variable *id* inside a hole. This defers the consistency check that would normally occur: a hole can be checked against any type, as long as its contents have some type. The cursor is placed inside the hole. The user then proceeds to apply *id* to the number expression $\underline{3}$. At this point, the expression inside the hole has a type consistent with the ascription, so the user can *finish* the hole. In our simple formalism, this requires moving the cursor to the hole (in practice, the system might find the nearest parent of hole form.) The result is the complete, well-typed program shown on Line 19 (notice that *complete* is distinct from *closed* – the variable *id* is free on Line 19, so this is not a closed program.)

3. Hazelnut, Formally

Hazelnut is based on the simply-typed lambda calculus extended with a single base type, `num`. Its major constituents, introduced by example in the previous section, are:

- **H-types** and **H-expressions** (Sec. 3.1), which are terms with *holes*. Holes mark subterms that are "under construction." H-types classify H-expressions according to a bidirectionally typed static semantics.

$$\begin{array}{llcl} \textsf{HTyp} & \tau, \dot\tau & ::= & \dot\tau \to \dot\tau \mid \texttt{num} \mid (\!|\,|\!) \\ \textsf{HExp} & e, \dot e & ::= & \dot e : \dot\tau \mid x \mid \lambda x.\dot e \mid \dot e(\dot e) \mid \underline{n} \mid \dot e + \dot e \mid (\!|\,|\!) \mid (\!|\dot e|\!) \end{array}$$

Figure 2. Syntax of H-types and H-expressions. Metavariable x ranges over variables and n ranges over numerals.

- **Z-types** and **Z-expressions** (Sec. 3.2), which superimpose a single *focus* onto H-types and H-expressions (using Huet's *zipper pattern* [?].)
- **Actions** (Sec. 3.3), which move the focus or modify the subterm in focus.

  Whenever an action is performed on a well-typed expression, it produces another well-typed expression in a *sensible* manner. More specifically, the action semantics satisfies a crucial *sensibility theorem*, stated in Sec. 3.3.

In our overview of the semantics below, we will reproduce only the most interesting rules, and in some cases we will do so "out of order." The appendix (and our Agda formalization, see Sec. 4) defines the complete collection of rules in their dependency order.

3.1 Holes

The syntax of H-types and H-expressions is given in Figure 2. Most of the forms correspond directly to those of the simply-typed lambda calculus extended with type `num`. The number expression corresponding to the number n is drawn $\underline{n}$, and for simplicity, we define only a single arithmetic operation, $\dot e + \dot e$. In addition to these standard forms, *empty holes* are drawn $(\!|\,|\!)$ and *non-empty H-expression holes* are drawn $(\!|\dot e|\!)$. In our simple calculus, all well-formed type expressions are valid types, so we do not need non-empty H-type holes.

We refer to terms that do not contain subterms of hole form as *complete*. Informally, we will use metavariables τ and e rather than $\dot\tau$ and $\dot e$ for complete H-types and H-expressions, respectively. Formally, we can derive τ complete when τ is a complete H-type, and e complete when e is a complete H-expression. We omit the straightforward definitions of these judgements for concision. The dynamics of Hazelnut, which we need not detail here, is defined only over complete H-expressions (i.e. we can only "run" a complete program, though see Sec. 7.)

The statics of Hazelnut is organized as a *bidirectional type system* [?], i.e. around the following mutually defined typing judgements:

$$\begin{array}{ll} \dot\Gamma \vdash \dot e \Leftarrow \dot\tau & \dot e \text{ analyzes against } \dot\tau \\ \dot\Gamma \vdash \dot e \Rightarrow \dot\tau & \dot e \text{ synthesizes } \dot\tau \end{array}$$

where typing contexts, $\dot\Gamma$, map each variable $x \in \text{dom}(\dot\Gamma)$ to a hypothesis $x : \dot\tau$. Derivations of the type analysis judgement establish that $\dot e$ can appear where an expression of type $\dot\tau$ is expected. Derivations of the type synthesis judgement determine a type that can be assigned to $\dot e$ even in positions where an expected type is not known (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 for giving a sensible action semantics to our system (Sec. 3.3.)

Type synthesis is stronger than type analysis in that if an expression is able to synthesize a type, it can also be analyzed against that type, or any *compatible* type. This is expressed by the *subsumption rule*:

$$\frac{\dot\Gamma \vdash \dot e \Rightarrow \dot\tau' \qquad \dot\tau \sim \dot\tau'}{\dot\Gamma \vdash \dot e \Leftarrow \dot\tau} \tag{1a}$$

The *H-type compatibility judgement*, $\dot\tau \sim \dot\tau'$, reduces to syntactic equality for complete H-types. For incomplete H-types, the rules are given after we discuss the semantics of holes below.

First, let us briefly review the standard constructs. Type ascription allows the user to state a type for the ascribed expression to be analyzed against:

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

A variable synthesizes the type that the context assigns to it:

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

Functions are not themselves annotated with types, so they can only appear in analytic position:

$$\frac{\dot\Gamma, x : \dot\tau_1 \vdash \dot e \Leftarrow \dot\tau_2}{\dot\Gamma \vdash \lambda x.\dot e \Leftarrow \dot\tau_1 \to \dot\tau_2} \tag{1d}$$

(It would be straightforward to also add a "half-annotated" lambda form, $\lambda x{:}\tau.e$, but for simplicity, we leave it out of our calculus [?].)

For function application, if the expression in function position synthesizes an arrow type, the argument is analyzed against the synthesized argument type:

$$\frac{\dot\Gamma \vdash \dot e_1 \Rightarrow \dot\tau_2 \to \dot\tau \qquad \dot\Gamma \vdash \dot e_2 \Leftarrow \dot\tau_2}{\dot\Gamma \vdash \dot e_1(\dot e_2) \Rightarrow \dot\tau} \tag{1e}$$

Numbers synthesize type `num`:

$$\frac{}{\dot\Gamma \vdash \underline{n} \Rightarrow \texttt{num}} \tag{1f}$$

Addition operates like a function over numbers:

$$\frac{\dot\Gamma \vdash \dot e_1 \Leftarrow \texttt{num} \qquad \dot\Gamma \vdash \dot e_2 \Leftarrow \texttt{num}}{\dot\Gamma \vdash \dot e_1 + \dot e_2 \Rightarrow \texttt{num}} \tag{1g}$$

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

The empty hole synthesizes the hole type:

$$\frac{}{\dot\Gamma \vdash (\!|\,|\!) \Rightarrow (\!|\,|\!)} \tag{1h}$$

A non-empty hole contains an H-expression that is "under construction". The inner expression must synthesize some type, but the non-empty hole synthesizes only the hole type:

$$\frac{\dot\Gamma \vdash \dot e \Rightarrow \dot\tau}{\dot\Gamma \vdash (\!|\dot e|\!) \Rightarrow (\!|\,|\!)} \tag{1i}$$

The type compatibility judgement $\dot\tau \sim \dot\tau'$, which appeared as a premise in the subsumption rule, makes the hole type compatible with any other type:

$$\frac{}{\dot\tau \sim (\!|\,|\!)} \tag{1ja}$$

The remaining rules, given in the appendix, establish that type compatibility is symmetric and reflexive (but not transitive.) Consequently, by subsumption, we can derive that $id : \texttt{num} \to \texttt{num} \vdash (\!|id|\!) \Leftarrow \texttt{num}$, as is necessary to synthesize a type for the H-expression on Line 14 of Fig. 1.

The final rule handles function applications where the expression in function position synthesizes a hole type, rather than an arrow type. We treat it as if it had instead synthesized $(\!|\,|\!) \to (\!|\,|\!)$:

$$\frac{\dot\Gamma \vdash \dot e_1 \Rightarrow (\!|\,|\!) \qquad \dot\Gamma \vdash \dot e_2 \Leftarrow (\!|\,|\!)}{\dot\Gamma \vdash \dot e_1(\dot e_2) \Rightarrow (\!|\,|\!)} \tag{1k}$$

$$\begin{array}{llll}
\mathsf{ZTyp} & \hat{\tau} & ::= & \triangleright\dot{\tau}\triangleleft \mid \hat{\tau} \to \dot{\tau} \mid \dot{\tau} \to \hat{\tau} \\
\mathsf{ZExp} & \hat{e} & ::= & \triangleright\dot{e}\triangleleft \mid \hat{e} : \dot{\tau} \mid \dot{e} : \hat{\tau} \mid \lambda x.\hat{e} \mid \hat{e}(\dot{e}) \mid \dot{e}(\hat{e}) \mid \hat{e} + \dot{e} \mid \dot{e} + \hat{e} \mid \llparenthesis\hat{e}\rrparenthesis
\end{array}$$

Figure 3. Syntax of Z-types and Z-expressions, i.e. types and expressions with holes and a single cursor.

$$\begin{array}{llll}
\mathsf{Action} & \alpha & ::= & \texttt{move}\ \delta \mid \texttt{del} \mid \texttt{construct}\ \varphi \mid \texttt{finish} \\
\mathsf{Direction} & \delta & ::= & \texttt{firstChild} \mid \texttt{parent} \mid \texttt{nextSib} \mid \texttt{prevSib} \\
\mathsf{Shape} & \varphi & ::= & \texttt{arrow} \mid \texttt{num} \\
 & & \mid & \texttt{asc} \mid \texttt{var}\ x \mid \texttt{lam}\ x \mid \texttt{ap} \mid \texttt{arg} \mid \texttt{numlit}\ n \mid \texttt{plus}
\end{array}$$

Figure 4. Syntax of actions.

The hole type behaves much like the type ? in prior work by Siek and Taha on gradual types for functional languages [**?**]. Their system (which was not bidirectionally typed nor an editor model) also needed to define two rules for function application. In general, when a premise requires that a synthesized type be of a particular form, we need a special case where the synthesized hole type is treated instead as if it were the "holey-est" type of that form.[2]

3.2 Focus Model

In order to identify a single subtree of an H-type or H-expression as the current focus of action, we apply Huet's *zipper pattern* [**?**]. The syntax of Z-types, $\hat{\tau}$, and Z-expressions, $\hat{e}$, is given in Figure 3. 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 is the current focus. All 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 focus will be found. All other sub-terms are H-types or H-expressions. Taken together, every syntactically well-formed Z-type and Z-expression contains exactly one focused H-type or H-expression.

We write $\hat{\tau}^\diamond$ for the H-type constructed by removing the focus marker from the Z-type $\hat{\tau}$. This straightforward metafunction is defined as follows:

$$(\triangleright\dot{\tau}\triangleleft)^\diamond = \dot{\tau}$$
$$(\hat{\tau} \to \dot{\tau})^\diamond = \hat{\tau}^\diamond \to \dot{\tau}$$
$$(\dot{\tau} \to \hat{\tau})^\diamond = \dot{\tau} \to \hat{\tau}^\diamond$$

Similarly, we write $\hat{e}^\diamond$ for the H-expression constructed by removing the focus marker from the Z-expression $\hat{e}$. The definition of this metafunction is analagous, so we leave it in the appendix for concision.

3.3 Action Semantics

The syntax of *actions*, α, some of which involve *directions*, δ, or *shapes*, φ, is given in Figure 4. Actions are performed on Z-types and Z-expressions according to the *action semantics* of Hazelnut, which is organized around three judgements:

$\hat{\tau} \xrightarrow{\alpha} \hat{\tau}'$	Performing α on $\hat{\tau}$ produces $\hat{\tau}'$
$\dot{\Gamma} \vdash \hat{e} \Rightarrow \dot{\tau} \xrightarrow{\alpha} \hat{e}' \Rightarrow \dot{\tau}'$	Performing α on $\hat{e}$ when $\hat{e}^\diamond$ synthesizes type $\dot{\tau}$ produces $\hat{e}'$ such that $\hat{e}'^\diamond$ synthesizes type $\dot{\tau}'$
$\dot{\Gamma} \vdash \hat{e} \xrightarrow{\alpha} \hat{e}' \Leftarrow \dot{\tau}$	Performing α on $\hat{e}$ when analyzing $\hat{e}^\diamond$ against $\dot{\tau}$ produces $\hat{e}'$, such that $\hat{e}'^\diamond$ can also be analyzed against $\dot{\tau}$

As suggested by the descriptions above, the action semantics maintains the following *action sensibility* theorem:

Theorem 1 (Action Sensibility)**.** *Both of the following hold:*

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

In words, every action leaves the program in a semantically well-defined state. More specifically, the first clause of Theorem 1 establishes that actions performed on expressions that synthesize a type can only produce expressions that also synthesize some (possibly different) type. The second clause establishes that actions performed on expressions in analytic position (e.g. those under type ascriptions or in argument position, see above) can only produce expressions that can also be analyzed against the expected type.

It is also useful to maintain a *deterministic* action semantics, i.e. every well-defined action should produce a unique Z-type or Z-expression. Formally, this is stated as follows:

Theorem 2 (Action Determinism)**.** *All of the following hold:*

1. *If* $\hat{\tau} \xrightarrow{\alpha} \hat{\tau}'$ *and* $\hat{\tau} \xrightarrow{\alpha} \hat{\tau}''$ *then* $\hat{\tau}' = \hat{\tau}''$.
2. *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* $\dot{\tau}' = \dot{\tau}''$.
3. *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* $\hat{e}' = \hat{e}''$.

In order to maintain determinism, we will need to supplement the definition of type compatibility above with a definition for *type incompatibility*, $\dot{\tau} \nsim \dot{\tau}'$. The key rule establishes that arrow types are incompatible with the `num` type:

$$\frac{}{\texttt{num} \nsim \dot{\tau}_1 \to \dot{\tau}_2} \tag{11a}$$

The remaining rules, given in the appendix, establish that type incompatibility is symmetric and covariant.

3.3.1 Subsumption

The action semantics includes a subsumption rule much like the one from the underlying semantics of H-expressions:

$$\frac{\dot{\Gamma} \vdash \hat{e}^\diamond \Rightarrow \dot{\tau}' \qquad \dot{\Gamma} \vdash \hat{e} \Rightarrow \dot{\tau}' \xrightarrow{\alpha} \hat{e}' \Rightarrow \dot{\tau}'' \qquad \dot{\tau} \sim \dot{\tau}''}{\dot{\Gamma} \vdash \hat{e} \xrightarrow{\alpha} \hat{e}' \Leftarrow \dot{\tau}} \tag{12}$$

In other words, if the expression synthesizes a type, then we defer to the synthetic action performance judgement, as long as it produces an expression that synthesizes a type compatible with the type provided for analysis. It is easy to see that this satisfies Theorem 1 by applying the IH and subsumption.

3.3.2 Relative Movement

Movement actions change the focus but do not change the underlying H-type or H-expression (so action sensibility is easy to show for these rules as well.)

The rules for relative movement within Z-types are given below and should be self-explanatory:

$$\frac{}{\triangleright\dot{\tau}_1 \to \dot{\tau}_2\triangleleft \xrightarrow{\texttt{move firstChild}} \triangleright\dot{\tau}_1\triangleleft \to \dot{\tau}_2} \tag{13a}$$

$$\frac{}{\triangleright\dot{\tau}_1\triangleleft \to \dot{\tau}_2 \xrightarrow{\texttt{move parent}} \triangleright\dot{\tau}_1 \to \dot{\tau}_2\triangleleft} \tag{13b}$$

$$\frac{}{\dot{\tau}_1 \to \triangleright\dot{\tau}_2\triangleleft \xrightarrow{\texttt{move parent}} \triangleright\dot{\tau}_1 \to \dot{\tau}_2\triangleleft} \tag{13c}$$

$$\frac{}{\triangleright\dot{\tau}_1\triangleleft \to \dot{\tau}_2 \xrightarrow{\texttt{move nextSib}} \dot{\tau}_1 \to \triangleright\dot{\tau}_2\triangleleft} \tag{13d}$$

$$\frac{}{\dot{\tau}_1 \to \triangleright\dot{\tau}_2\triangleleft \xrightarrow{\texttt{move prevSib}} \triangleright\dot{\tau}_1\triangleleft \to \dot{\tau}_2} \tag{13e}$$

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

[2] Alternatively, we might add a rule that allows expressions that synthesize hole type to then non-deterministically synthesize any other type, but maintaining determinism is useful in practice, so we avoid this approach.

judgements:

$$\frac{\hat{e} \xrightarrow{\texttt{move}\ \delta} \hat{e}'}{\dot{\Gamma} \vdash \hat{e} \Rightarrow \dot{\tau} \xrightarrow{\texttt{move}\ \delta} \hat{e}' \Rightarrow \dot{\tau}} \tag{14a}$$

$$\frac{\hat{e} \xrightarrow{\texttt{move}\ \delta} \hat{e}'}{\dot{\Gamma} \vdash \hat{e} \xrightarrow{\texttt{move}\ \delta} \hat{e}' \Leftarrow \dot{\tau}} \tag{14b}$$

For concision, we show only the rules for ascription here:

$$\frac{}{\triangleright\dot{e} : \dot{\tau}\triangleleft \xrightarrow{\texttt{move firstChild}} \triangleright\dot{e}\triangleleft : \dot{\tau}} \tag{15a}$$

$$\frac{}{\triangleright\dot{e}\triangleleft : \dot{\tau} \xrightarrow{\texttt{move parent}} \triangleright\dot{e} : \dot{\tau}\triangleleft} \tag{15b}$$

$$\frac{}{\dot{e} : \triangleright\dot{\tau}\triangleleft \xrightarrow{\texttt{move parent}} \triangleright\dot{e} : \dot{\tau}\triangleleft} \tag{15c}$$

$$\frac{}{\triangleright\dot{e}\triangleleft : \dot{\tau} \xrightarrow{\texttt{move nextSib}} \dot{e} : \triangleright\dot{\tau}\triangleleft} \tag{15d}$$

$$\frac{}{\dot{e} : \triangleright\dot{\tau}\triangleleft \xrightarrow{\texttt{move prevSib}} \triangleright\dot{e}\triangleleft : \dot{\tau}} \tag{15e}$$

$$\frac{\hat{e} \xrightarrow{\texttt{move}\ \delta} \hat{e}'}{\hat{e} : \dot{\tau} \xrightarrow{\texttt{move}\ \delta} \hat{e}' : \dot{\tau}} \tag{15f}$$

$$\frac{\hat{\tau} \xrightarrow{\texttt{move}\ \delta} \hat{\tau}'}{\dot{e} : \hat{\tau} \xrightarrow{\texttt{move}\ \delta} \dot{e} : \hat{\tau}'} \tag{15g}$$

3.3.3 Deletion

The `del` action replaces the selected subterm with an empty hole.

Again, the rule for Z-types is self-explanatory:

$$\frac{}{\triangleright\dot{\tau}\triangleleft \xrightarrow{\texttt{del}} \triangleright\llparenthesis\rrparenthesis\triangleleft} \tag{16a}$$

Deletion within a Z-expression is similarly straightforward:

$$\frac{}{\dot{\Gamma} \vdash \triangleright\dot{e}\triangleleft \Rightarrow \dot{\tau} \xrightarrow{\texttt{del}} \triangleright\llparenthesis\rrparenthesis\triangleleft \Rightarrow \llparenthesis\rrparenthesis} \tag{17a}$$

$$\frac{}{\dot{\Gamma} \vdash \triangleright\dot{e}\triangleleft \xrightarrow{\texttt{del}} \triangleright\llparenthesis\rrparenthesis\triangleleft \Leftarrow \dot{\tau}} \tag{17b}$$

3.3.4 Construction

The construction actions, `construct` φ, are used to construct terms of a shape indicated by φ into the program at or around the focus.

Again, let us begin with type actions. The `construct arrow` action constructs an arrow type. The focused H-type becomes the argument type, and the focus is placed on an empty return type hole:

$$\frac{}{\triangleright\dot{\tau}\triangleleft \xrightarrow{\texttt{construct arrow}} \dot{\tau} \rightarrow \triangleright\llparenthesis\rrparenthesis\triangleleft} \tag{18a}$$

The `construct num` action replaces an empty Z-type hole with the `num` type:

$$\frac{}{\triangleright\llparenthesis\rrparenthesis\triangleleft \xrightarrow{\texttt{construct num}} \triangleright\texttt{num}\triangleleft} \tag{18b}$$

Moving on to expression actions, we start to see more interesting rules. The `construct asc` action operates differently depending on whether the focused expression synthesizes a type or is being analyzed against a type. In the first case, the ascribed type is the synthesized type:

$$\frac{}{\dot{\Gamma} \vdash \triangleright\dot{e}\triangleleft \Rightarrow \dot{\tau} \xrightarrow{\texttt{construct asc}} \dot{e} : \triangleright\dot{\tau}\triangleleft \Rightarrow \dot{\tau}} \tag{19a}$$

In the second case, the ascribed type is the type provided for analysis:

$$\frac{}{\dot{\Gamma} \vdash \triangleright\dot{e}\triangleleft \xrightarrow{\texttt{construct asc}} \dot{e} : \triangleright\dot{\tau}\triangleleft \Leftarrow \dot{\tau}} \tag{19b}$$

The `construct var` x action places the variable x into the focused empty hole. If that hole is being asked to synthesize a type, then the result of the action synthesizes the type assigned to x in the context:

$$\frac{}{\dot{\Gamma}, x : \dot{\tau} \vdash \triangleright\llparenthesis\rrparenthesis\triangleleft \Rightarrow \llparenthesis\rrparenthesis \xrightarrow{\texttt{construct var}\ x} \triangleright x\triangleleft \Rightarrow \dot{\tau}} \tag{19c}$$

If the focused empty hole is being analyzed against a type that is inconsistent with the type assigned to x by the context, x is placed inside a hole:

$$\frac{\dot{\tau} \nsim \dot{\tau}'}{\dot{\Gamma}, x : \dot{\tau}' \vdash \triangleright\llparenthesis\rrparenthesis\triangleleft \xrightarrow{\texttt{construct var}\ x} \llparenthesis\triangleright x\triangleleft\rrparenthesis \Leftarrow \dot{\tau}} \tag{19d}$$

The rule above featured in the example in Section 2.

Notice that no rule was necessary for the case where the hole was being analyzed against a type compatible with the variable's type, because this case is handled by the action subsumption rule.

The `construct lam` x action places a lambda term binding x into an empty hole. If the focused empty hole is being asked to synthesize a type, then the result of the action is a lambda ascribed the type $\llparenthesis\rrparenthesis \rightarrow \llparenthesis\rrparenthesis$, with the focus in the argument type position:

$$\frac{}{\dot{\Gamma} \vdash \triangleright\llparenthesis\rrparenthesis\triangleleft \Rightarrow \llparenthesis\rrparenthesis \xrightarrow{\texttt{construct lam}\ x} \lambda x.\llparenthesis\rrparenthesis : \triangleright\llparenthesis\rrparenthesis\triangleleft \rightarrow \llparenthesis\rrparenthesis \Rightarrow \llparenthesis\rrparenthesis \rightarrow \llparenthesis\rrparenthesis} \tag{19e}$$

The type ascription is necessary because lambda expressions do not synthesize a type. If the focused empty hole is being analyzed against an arrow type, then no ascription is necessary:

$$\frac{}{\dot{\Gamma} \vdash \triangleright\llparenthesis\rrparenthesis\triangleleft \xrightarrow{\texttt{construct lam}\ x} \lambda x.\triangleright\llparenthesis\rrparenthesis\triangleleft \Leftarrow \dot{\tau}_1 \rightarrow \dot{\tau}_2} \tag{19f}$$

If the focused empty hole is being analyzed against a type that is incompatible with an arrow type, then a lambda ascribed the type $\llparenthesis\rrparenthesis \rightarrow \llparenthesis\rrparenthesis$ is inserted inside a hole, to maintain Theorem 1:

$$\frac{\dot{\tau} \nsim \llparenthesis\rrparenthesis \rightarrow \llparenthesis\rrparenthesis}{\dot{\Gamma} \vdash \triangleright\llparenthesis\rrparenthesis\triangleleft \xrightarrow{\texttt{construct lam}\ x} \llparenthesis\lambda x.\llparenthesis\rrparenthesis : \triangleright\llparenthesis\rrparenthesis\triangleleft \rightarrow \llparenthesis\rrparenthesis\rrparenthesis \Leftarrow \dot{\tau}} \tag{19g}$$

The `construct ap` action applies the expression in focus to a hole. If the focused expression synthesizes a function type, then the rule is straightforward:

$$\frac{}{\dot{\Gamma} \vdash \triangleright\dot{e}\triangleleft \Rightarrow \dot{\tau}_1 \rightarrow \dot{\tau}_2 \xrightarrow{\texttt{construct ap}} \dot{e}(\triangleright\llparenthesis\rrparenthesis\triangleleft) \Rightarrow \dot{\tau}_2} \tag{19h}$$

If the focused expression synthesizes a hole type, then we can treat it as if it synthesized the $\llparenthesis\rrparenthesis \rightarrow \llparenthesis\rrparenthesis$ type, exactly as described in Sec. 3.1:

$$\frac{}{\dot{\Gamma} \vdash \triangleright\dot{e}\triangleleft \Rightarrow \llparenthesis\rrparenthesis \xrightarrow{\texttt{construct ap}} \dot{e}(\triangleright\llparenthesis\rrparenthesis\triangleleft) \Rightarrow \llparenthesis\rrparenthesis} \tag{19i}$$

Finally, if the focused expression synthesizes a type that is incompatible with an arrow type, then we must place that expression inside a hole to maintain Theorem 3.1:

$$\frac{\dot{\tau} \nsim \llparenthesis\rrparenthesis \rightarrow \llparenthesis\rrparenthesis}{\dot{\Gamma} \vdash \triangleright\dot{e}\triangleleft \Rightarrow \dot{\tau} \xrightarrow{\texttt{construct ap}} \llparenthesis\dot{e}\rrparenthesis(\triangleright\llparenthesis\rrparenthesis\triangleleft) \Rightarrow \llparenthesis\rrparenthesis} \tag{19j}$$

The `construct arg` action places the focused expression instead in the argument position of an application. Because the function position is always an empty hole in this situation, we only need a single rule:

$$\frac{}{\dot{\Gamma} \vdash \triangleright\dot{e}\triangleleft \Rightarrow \dot{\tau} \xrightarrow{\texttt{construct arg}} \triangleright\llparenthesis\rrparenthesis\triangleleft(\dot{e}) \Rightarrow \llparenthesis\rrparenthesis} \tag{19k}$$

The `construct numlit` n action places the number expression $\underline{n}$ into an empty hole. If the focused hole is being asked to synthesize a type, then the rule is straightforward:

$$\frac{}{\dot\Gamma \vdash \triangleright(\!|\,|\!)\triangleleft \Rightarrow (\!|\,|\!) \xrightarrow{\texttt{construct numlit } n} \triangleright\underline{n}\triangleleft \Rightarrow \texttt{num}} \quad (19l)$$

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

$$\frac{\dot\tau \nsim \texttt{num}}{\dot\Gamma \vdash \triangleright(\!|\,|\!)\triangleleft \xrightarrow{\texttt{construct numlit } n} (\!|\triangleright\underline{n}\triangleleft|\!) \Leftarrow \dot\tau} \quad (19m)$$

Finally, the `construct plus` action constructs a plus expression with the focused expression as its first argument. If the focused expression synthesizes a type consistent with `num`, then the rule is straightforward:

$$\frac{\dot\tau \sim \texttt{num}}{\dot\Gamma \vdash \triangleright\dot e\triangleleft \Rightarrow \dot\tau \xrightarrow{\texttt{construct plus}} \dot e + \triangleright(\!|\,|\!)\triangleleft \Rightarrow \texttt{num}} \quad (19n)$$

Otherwise, we must place the focused expression inside a hole:

$$\frac{\dot\tau \nsim \texttt{num}}{\dot\Gamma \vdash \triangleright\dot e\triangleleft \Rightarrow \dot\tau \xrightarrow{\texttt{construct plus}} (\!|\dot e|\!) + \triangleright(\!|\,|\!)\triangleleft \Rightarrow \texttt{num}} \quad (19o)$$

Notice that we do not have an action that explicitly wraps an expression in a non-empty hole. These arise implicitly when an action that would not naïvely satisfy Theorem 1 is performed (see Figure 1.)

3.3.5 Finishing

The final action we will consider in Hazelnut is `finish`, which finishes the focused non-empty hole.

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

$$\frac{\dot\Gamma \vdash \dot e \Rightarrow \dot\tau'}{\dot\Gamma \vdash \triangleright(\!|\dot e|\!)\triangleleft \Rightarrow (\!|\,|\!) \xrightarrow{\texttt{finish}} \triangleright\dot e\triangleleft \Rightarrow \dot\tau'} \quad (20a)$$

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

$$\frac{\dot\Gamma \vdash \dot e \Leftarrow \dot\tau}{\dot\Gamma \vdash \triangleright(\!|\dot e|\!)\triangleleft \xrightarrow{\texttt{finish}} \triangleright\dot e\triangleleft \Leftarrow \dot\tau} \quad (20b)$$

3.3.6 Zipper Cases

The rules given so far handle the base cases, where the action has "reached" the focused expression. We also need to define the recursive cases, which propagate the action into the subtree where the focus appears. These rules follow the structure of the corresponding rules in the statics of H-expressions.

For example, when the focus is in the expression position of an ascription, we use the analytic action performance judgement:

$$\frac{\dot\Gamma \vdash \hat e \xrightarrow{\alpha} \hat e' \Leftarrow \dot\tau}{\dot\Gamma \vdash \hat e : \dot\tau \Rightarrow \dot\tau \xrightarrow{\alpha} \hat e' : \dot\tau \Rightarrow \dot\tau} \quad (21a)$$

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

$$\frac{\hat\tau \xrightarrow{\alpha} \hat\tau' \qquad \dot\Gamma \vdash \dot e \Leftarrow \hat\tau'^{\diamond}}{\dot\Gamma \vdash \dot e : \hat\tau \Rightarrow \hat\tau^{\diamond} \xrightarrow{\alpha} \dot e : \hat\tau' \Rightarrow \hat\tau'^{\diamond}} \quad (21b)$$

If the focus is in the body of a lambda expression, then we must use the analytic action performance rule:

$$\frac{\dot\Gamma, x : \dot\tau_1 \vdash \hat e \xrightarrow{\alpha} \hat e' \Leftarrow \dot\tau_2}{\dot\Gamma \vdash \lambda x.\hat e \xrightarrow{\alpha} \lambda x.\hat e' \Leftarrow \dot\tau_1 \to \dot\tau_2} \quad (21c)$$

There are two rules that handle the case where the focus is in the function position of an application, corresponding to the two application rules in the statics. Each involves rechecking the argument against the new function type:

$$\frac{\begin{array}{c}\dot\Gamma \vdash \hat e^{\diamond} \Rightarrow \dot\tau_2 \\ \dot\Gamma \vdash \hat e \Rightarrow \dot\tau_2 \xrightarrow{\alpha} \hat e' \Rightarrow \dot\tau_3 \to \dot\tau_4 \qquad \dot\Gamma \vdash \dot e \Leftarrow \dot\tau_3\end{array}}{\dot\Gamma \vdash \hat e(\dot e) \Rightarrow \dot\tau_1 \xrightarrow{\alpha} \hat e'(\dot e) \Rightarrow \dot\tau_4} \quad (21d)$$

$$\frac{\dot\Gamma \vdash \hat e^{\diamond} \Rightarrow \dot\tau_2 \qquad \dot\Gamma \vdash \hat e \Rightarrow \dot\tau_2 \xrightarrow{\alpha} \hat e' \Rightarrow (\!|\,|\!) \qquad \dot\Gamma \vdash \dot e \Leftarrow (\!|\,|\!)}{\dot\Gamma \vdash \hat e(\dot e) \Rightarrow \dot\tau_1 \xrightarrow{\alpha} \hat e'(\dot e) \Rightarrow (\!|\,|\!)} \quad (21e)$$

Similarly, there are two rules that handle the case where the focus is in the argument position:

$$\frac{\dot\Gamma \vdash \dot e \Rightarrow \dot\tau_2 \to \dot\tau \qquad \dot\Gamma \vdash \hat e \xrightarrow{\alpha} \hat e' \Leftarrow \dot\tau_2}{\dot\Gamma \vdash \dot e(\hat e) \Rightarrow \dot\tau \xrightarrow{\alpha} \dot e(\hat e') \Rightarrow \dot\tau} \quad (21f)$$

$$\frac{\dot\Gamma \vdash \dot e \Rightarrow (\!|\,|\!) \qquad \dot\Gamma \vdash \hat e \xrightarrow{\alpha} \hat e' \Leftarrow (\!|\,|\!)}{\dot\Gamma \vdash \dot e(\hat e) \Rightarrow (\!|\,|\!) \xrightarrow{\alpha} \dot e(\hat e') \Rightarrow (\!|\,|\!)} \quad (21g)$$

The rules for the addition operator follow from the statics directly:

$$\frac{\dot\Gamma \vdash \hat e \xrightarrow{\alpha} \hat e' \Leftarrow \texttt{num}}{\dot\Gamma \vdash \hat e + \dot e \Rightarrow \texttt{num} \xrightarrow{\alpha} \hat e' + \dot e \Rightarrow \texttt{num}} \quad (21h)$$

$$\frac{\dot\Gamma \vdash \hat e \xrightarrow{\alpha} \hat e' \Leftarrow \texttt{num}}{\dot\Gamma \vdash \dot e + \hat e \Rightarrow \texttt{num} \xrightarrow{\alpha} \dot e + \hat e' \Rightarrow \texttt{num}} \quad (21i)$$

Finally, if the focus is inside a non-empty hole, we special case the situation where the action results in a doubly-nested empty hole, $(\!|(\!|\,|\!)|\!)$, to eliminate the nesting (given our current action semantics, only the delete action can cause this form to arise and the form $(\!|(\!|\hat e|\!)|\!)$ cannot arise):

$$\frac{\dot\Gamma \vdash \hat e^{\diamond} \Rightarrow \dot\tau \qquad \dot\Gamma \vdash \hat e \Rightarrow \dot\tau \xrightarrow{\alpha} \hat e' \Rightarrow \dot\tau' \qquad \hat e' \neq \triangleright(\!|\,|\!)\triangleleft}{\dot\Gamma \vdash (\!|\hat e|\!) \Rightarrow (\!|\,|\!) \xrightarrow{\alpha} (\!|\hat e'|\!) \Rightarrow (\!|\,|\!)} \quad (21j)$$

$$\frac{\dot\Gamma \vdash \hat e^{\diamond} \Rightarrow \dot\tau \qquad \dot\Gamma \vdash \hat e \Rightarrow \dot\tau \xrightarrow{\alpha} \triangleright(\!|\,|\!)\triangleleft \Rightarrow (\!|\,|\!)}{\dot\Gamma \vdash (\!|\hat e|\!) \Rightarrow (\!|\,|\!) \xrightarrow{\alpha} \triangleright(\!|\,|\!)\triangleleft \Rightarrow (\!|\,|\!)} \quad (21k)$$

4. Mechanization

In the previous section, we gave an overview of the most important rules in the semantics of Hazelnut, and stated the important theorems. In a few cases, we sketched out why these theorems will hold.

In order to formally verify that our design meets the stated objectives, we are preparing a formalization of the grammar, the judgements given above and the metatheory in the proof assistant Agda [**?**]. We refer readers unfamiliar with Agda to the Agda Wiki, hosted at http://wiki.portal.chalmers.se/agda/.

Our formalization of Hazelnut is under development at http://github.com/hazelgrove/agda-tfp16. At the time of submission, we have completed much of the initial groundwork, but the proofs of Theorem 1 and Theorem 2 (and a few lemmas that we elided here) are under construction.

The documentation includes a more detailed discussion of the technical representation decisions made. The core idea of our formalization is to encode each judgement as a depen-

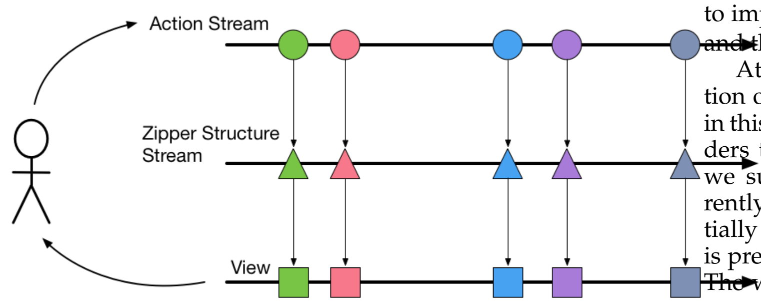

Figure 5. Implementation Concepts. Each Action–Zipper Structure–View combination is considered to appear "instantaneously" on the timeline.

dent type. The rules of the judgements become the constructors of the type, and derivations of theorems values of the type. This is a rich setting that allows proofs to take advantage of pattern matching on the shape of derivations, closely matching on-paper proofs of similar properties.

The formalization differs from the calculus defined here in a few small ways. Most interestingly, instead of giving separate movement rules for each form of Z-expression, we abstract over different corresponding forms that happen to have the same arity, e.g. additions and applications. Formalizing this intuition reduces the number of cases we need to consider somewhat, but more importantly allows us to write a slightly more general calculus – it will be easier to extend Hazelnut with more interesting language features with this generic infrastructure in place.

Because the metatheory in this paper is largely concerned with the statics of Hazelnut rather than its dynamics, we adopt Barendregt's convention for bound variables and avoid substitution entirely [?]. Future iterations will need a more mature technique for reasoning about binding—likely de Bruijn indices or abstract binding trees [? ?].

5. Implementation

5.1 Implementation Concepts

The key question that must be answered for any implementation strategy is: how do we model a stream of actions from a user? Let us assume that these actions are chosen (using some input device, preferably, a keyboard) from some "palette" that never presents the user with actions that are not semantically well-defined, according to the action semantics defined earlier. As such, each new action will "atomically" generate a new Z-expression. This insight leads us to conclude that a natural way to implement this editor would be using event-based Functional Reactive Programming [?] (FRP). Figure 5 illustrates the concept of an FRP-based implementation of a structure editor organized like Hazelnut. The input from the user is a stream of actions. Each action results in a change to the underlying abstract model (i.e., a new Z-expression is created after each action.) Each model change results in an updated *view* which is then presented to the user. The user can then consider this new view when they choose a new action as input.

5.2 HZ

We explore the concepts presented in the paper in HZ, our implementation of Hazelnut. In order to reach a wide audience, we decided to implement HZ in the web browser. In order to take advantage of all the benefits of FRP, we chose to implement HZ using OCaml[3], the `js_of_ocaml` compiler[4] and the OCaml React library[5].

At the time of the writing of this paper, our implementation of HZ includes encodings of Z-expression as presented in this paper. We consider this ZExp to be our model. HZ renders the model as a string embedded in HTML. Currently we support only the delete action. Other actions are currently under development. We anticipate having a substantially more functional implementation by the time this work is presented (our focus thusfar has been on the metatheory.) The work-in-progress code as well as directions for how to compile and run it can be found here: `https://github.com/hazelgrove/impl-tfp16`.

A substantially simpler system that we developed while exploring the ideas that led to Hazelnut can be found at the following URL: `http://www.cs.cmu.edu/~comar/nestedpairs/`.

6. Related Work

Structured editing has been recognized as a way to avoid the possibility of syntax errors for decades. An early example is the The Cornell Program Synthesizer [?], first published in 1981. The synthesizer generator [?] allows the user to create an attribute-grammar specification that then can be used to generate a structured editor. CENTAUR [?] produces a language specific environment from a user defined formal specification of a language. Barista [?] is a modern take on the same basic concept.

Novice programmers have been a common target for structure editors. For example, GNOME[?] was developed to teach programming to undergraduates. Scratch [?] is a structure editor targeted at children ages 8 to 16. Touchdevelop [?] incorporates a structure editor for programming on touch-based devices, and is used to teach high school students. Alice [?] is a 3-D programming language with an integrated structure editor for teaching novice CS undergraduate students. These are largely drag-and-drop user interfaces with a limited action model and an unclear semantics.

Not all structure editors are for educational purposes. For example, mbeddr [?] is an extensible C-based Programming Language and IDE (nominally, for programming embedded systems.) mbeddr is build on top of the commercial JetBrains MPS framework for constructing structure editors. Another popular approach is to bring elements of structured editing into a traditional editor. Codelets [?] uses structured editing to add interactive documentation and examples in an editor. Our previous work on Graphite [?] allows developers to associate structured editing interfaces called *palettes* with types. Graphite is integrated into a text-based program editor (Eclipse.)

Agda and Idris are two dependently typed languages that attempt to simulate a structured editor from within a rich text editor (e.g. Emacs.) These systems also have notions of holes and use types to guide the user toward filling these holes. These systems are also, to our knowledge, not formally well-defined but rather exist only as part of system implementations.

Perhaps the systems most similar in spirit to Hazelnut are Lamdu [?] and Unison [?]. Like Hazelnut, these are both statically typed functional language editors. In both cases, the language is similar to Haskell. In Lamdu, the editor uses

[3] https://ocaml.org/

[4] http://ocsigen.org/js_of_ocaml/

[5] http://erratique.ch/software/react

structure editing to enable Live Programming, where the code is always being executed as it is being written.

Our work differs from all of these in that we begin with a formal editor calculus and build from there, rather than starting with an implementation and leaving many of the formal details formally unspecified. For example, while Lamdu has many interesting features, there is no theoretical basis presented for their work – it is a rather large body of Haskell code with an unclear (and indeed, often somewhat perplexing, in our experience) action model. Unison is also a rather large body of Haskell code, though its action model appears superficially more similar to ours. We maintain what we believe to be a stronger action sensibility invariant than Unison (i.e. in Unison, one must construct expressions from the outside-in.) These systems are rich sources of interesting ideas, however – there is room enough for many different approaches in this (re-)emerging space.

7. Discussion & Conclusion

This paper presented Hazelnut, a type theoretic structure editor calculus. Our aim is to take a principled approach to its design by formally specifying its semantics, providing strong metatheoretic guarantees, mechanizing its semantics and metatheory in Agda and implementing it using the concepts of functional reactive programming. As of this submission, we have achieved reasonable confidence in the formal system presented above, and have transitioned our focus toward the mechanization and implementation efforts. By the time of presentation, we anticipate having complete or nearly complete versions of these.

7.1 Future Work

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. Our plan is to simultaneously maintain a mechanization and implementation (following, for example, Standard ML) as we proceed, ultimately producing the first large-scale, formally verified bidirectionally typed language codesigned with a type-aware 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.

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. It also did not provide any specific model of how available actions will be determined for presentation to the user. In practice, we would want also to rank available actions in some reasonable manner (perhaps based on usage data gathered from other users or code repositories.)

Another research direction is in exploring how types can be used to control the presentation of expressions in the editor. For example, following our approach in a textual setting on *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 [**?**].

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 focii for different users, and so on. These are all interesting avenues for future work.

On the theoretical side, the notion of having one of many possible holes in a term in focus has a very strong intuitive connection to the proof theoretic notion of focusing [**?**]. Beyond just the name, both seem to involve, in some sense, a search through the space of possible ways to finish a derivation. We intend to explore this connection to see if it's coincidental or more meaningful and welcome insights in this regard.

We already discussed a connection to gradual typing [**?**]. We hope to explore this connection more thoroughly. In particular, it may be possible to better support exploratory and live programming by allowing even programs with holes in them to execute as long as those holes are only in the type portions, by deferring to the semantics given in work on gradual typing.

It may also be possible to give a dynamics to incomplete expressions. Prior work on staged evaluation suggests that there may be a connection to modal logic, viewing holes as quantifying over all possible terms that may fill them [**?**]. In developing a dynamic semantics, we will also need to handle terms like $⦇⦇⦈⦈$ and $⦇⦇\dot{e}⦈⦈$. In our semantics given here, we eliminated them as they came up in a somewhat *ad hoc* manner. We have not yet explored an equational theory for terms with holes, but intend to once our formalization effort is more mature.

> 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