

An Integrative Human-Centered Architecture for Interactive Programming Assistants

Andrew Blinn
University of Michigan
Ann Arbor, MI, USA
blinnand@umich.edu

David Moon*
University of Michigan
Ann Arbor, MI, USA
dmoo@umich.edu

Eric Griffis*
University of Michigan
Ann Arbor, MI, USA
egriffis@umich.edu

Cyrus Omar
University of Michigan
Ann Arbor, MI, USA
comar@umich.edu

Abstract—Programming has become a collaboration between human programmers, who drive intent, and interactive assistants that suggest contextually relevant editor actions. There is considerable work on suggestion synthesis strategies—from semantic autocompletion to modern program synthesis, repair, and machine learning research. This diversity of strategies creates a need for an integrative, human-centered perspective on the problem of programming assistant design that (1) confronts the problem of integrating multiple synthesis strategies, fed by shared semantic analyses capable of operating on program sketches, and (2) centers the needs of the human programmer: comprehending, comparing, ranking, and filtering variously-generated suggestions, and sometimes participating in a synthesizer’s search by supplying added expressions of intent.

This paper contributes a conceptual architecture and API to guide assistant designers in confronting these integration and human-centered design challenges. We instantiate this architecture in three prototype end-to-end assistant designs. First, two implementations, developed for the Hazel programming environment, emphasizing understudied design foci including continuity, explainability, human-in-the-loop synthesis, and the integration of multiple analyses with multiple synthesis strategies. Secondly, a formalized assistant founded in programming language theory, serving as a worked-through model for precisely specifying and proving sensibility of suggestions.

I. INTRODUCTION

A *programming assistant* is an editor service that analyzes the editor state (consisting primarily of a program sketch, perhaps with additional data such as history) to present edit action suggestions to a human user and help the user select an action consistent with their broader intent [1–3]. Programming assistants promise to improve programmer productivity by automating tedious tasks. They may also improve software quality by helping programmers avoid mistakes and, in some cases, guarantee that the suggestions satisfy programmer-specified correctness constraints. Moreover, they can reduce knowledge

gaps by surfacing structures and idioms that a programmer might not have otherwise discovered [4–6].

Given these benefits, it is unsurprising that simple assistants like code completion and “hotfix” systems are ubiquitous in modern programming environments, competing in frequency with manual editor actions like code insertion and deletion [7]. There has in turn been substantial research interest [8] in techniques that can synthesize better suggestions, including program synthesis using types [9], examples [10], program sketches [11], edit history [12], demonstrations [13], and logical constraints [14] to generate “hole completions” [15, 16]. Github Copilot [17] is one of several recent efforts focused on synthesizing long-form completions via machine learning techniques that learn idioms from a vast corpus of real programs [18–21].

While much of this research has focused on the underlying synthesis algorithms, there has recently been a renaissance of human-centered approaches reminding us that the human programmer ultimately remains the driver of intent and the arbiter of correctness. Consequently, the interfaces through which the human communicates purpose and considers suggestions must be designed with cognitive costs in mind. For example, many of these synthesis techniques are capable of substantial associative leaps, carrying with them concerns about explainability and the costs of validating correctness. In addition, code search spaces can become large, leading either to lengthy synthesis delays or overwhelming numbers of suggestions. Concerns like these have led to work on interpretable synthesis [22] and interactive search space exploration [23].

We seek to organize this often-overwhelming diversity of efforts by developing an integrative architecture for programming assistant designers that confronts the problem of integrating a wide variety of synthesis techniques (and requisite program analyses) while centering the needs of the human user. Each component of this architecture is the subject of ongoing research, as is the overall design problem. We demonstrate that our architecture can serve to characterize and situate some existing assistant designs. We then describe our ongoing work on two end-to-end prototypes intended to emphasize understudied design criteria, namely continuity of service, integration of multiple shared analyses with multiple synthesis strategies,

*Preliminary Exam Content Note: This paper is an extended version of the short paper published last year at VL/HCC 2022. All of the writing is mine; David Moon and Eric Griffis are listed as co-authors as they share equal credit with myself for the design and implementation of the Hazel Live assistant, one of the three featured prototypes embodying the architecture described in this paper.

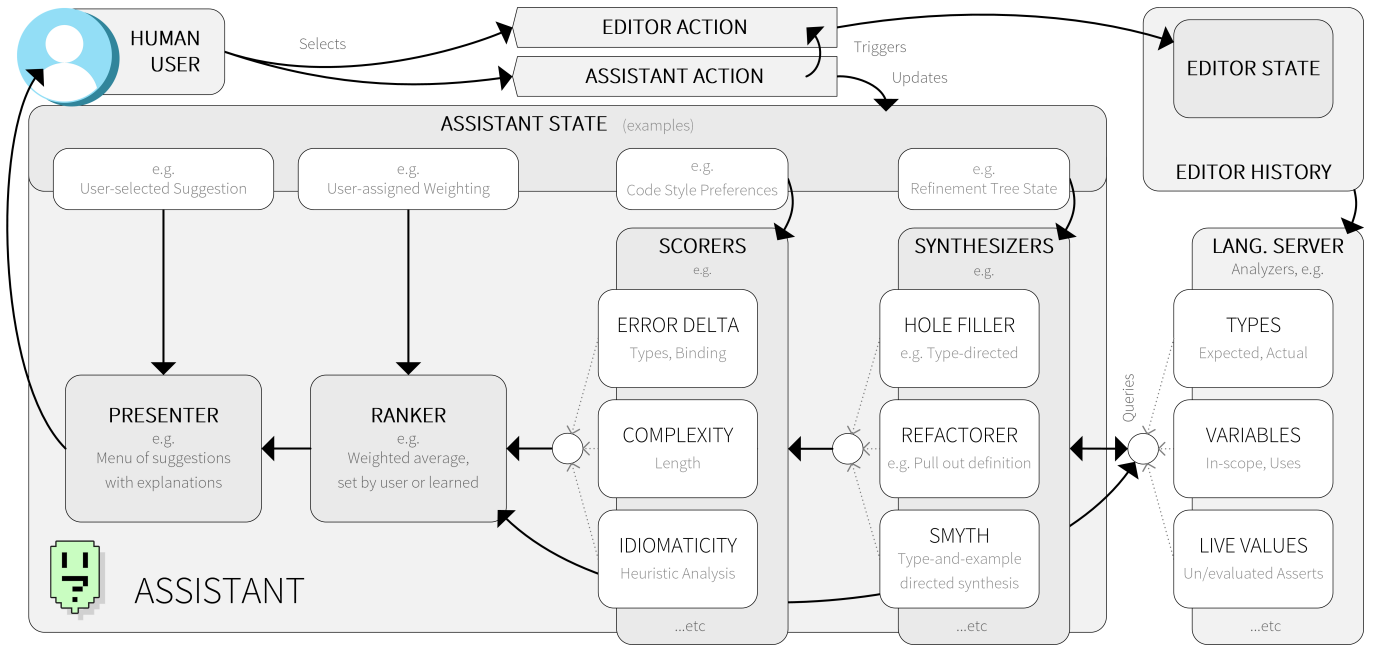


Fig. 1: Integrative human-centered assistant architecture diagram detailing suggestion dataflow and the user interaction loop

explainability, independent semantic ranking techniques, and interfaces for integrating the human into an incremental search.

We go on to present a third prototype, a judgemental formalism of an assistant rather than an implementation, building on the Hazelnut editor calculus [24, 25]. This serves dual purposes. First, it grounds our architecture and attendant vocabulary in the mathematical study of programming language semantics, which we hope may serve as bridge between language researchers and those implementing and studying programming interfaces. Second, we believe that programmers are best served by assistants capable of clear and concrete claims as to the correctness and sensibility of their suggestions. Deciding what it means for a suggestion to be *sensible* is itself a significant design issue [26], one we do not intend to resolve. Rather we select a simple characterization—that suggested actions should, at least, preserve meaningfulness across edit states, and do no harm, in the sense of not increasing the number of static errors—and offer our articulation and proof of this sensibility theorem as a paradigm for further work.

Our intention is *not* to make empirical claims about the specific design choices made in these prototypes. Indeed, there is much work to be done before successors to these designs can claim to improve overall programmer productivity. Rather we present these prototypes as illustrations of the proposed integrative architecture, which we hope will help organize and provide vocabulary for the assistant design community, and to draw attention to understudied but important design criteria that we hope will draw more

interest from the community.

II. ARCHITECTURAL OVERVIEW

Essentially all program editors support a basic interaction loop whereby a human user triggers editor actions, resulting in an updated editor state [24]. The editor state is presented to the user alongside various editor services that provide supporting feedback, e.g. syntax highlighting and type information [27]. This feedback requires performing language-aware analyses. Because the same analyses might be relevant to multiple services and across multiple editors, analyzers are typically collected behind a shared *language server* interface [28].

Analyses made available by a language server also feed the programming assistant (Fig. 1). An important consideration, particularly relevant to programming assistants, is that these servers must be able to cope with program sketches, i.e. editor states that are not yet syntactically valid or complete, or where there are static errors [24, 29]. When unable to do so, this can lead to gaps in service. Consequently, there has been much effort put into heuristics such as error-recovery and incremental parsing [28, 30, 31] or in automatic hole insertion [24, 29]. In the context of a programming assistant, it is exactly these incomplete states where assistance is most necessary, so gaps in the availability of the analyzers offered by the language server must be avoided whenever possible.

Turning now to the assistant itself, we see a dataflow beginning with a collection of *Synthesizers* which each generate sets of edit action suggestions with accompanying explanatory metadata, as expressed in these notional type definitions:

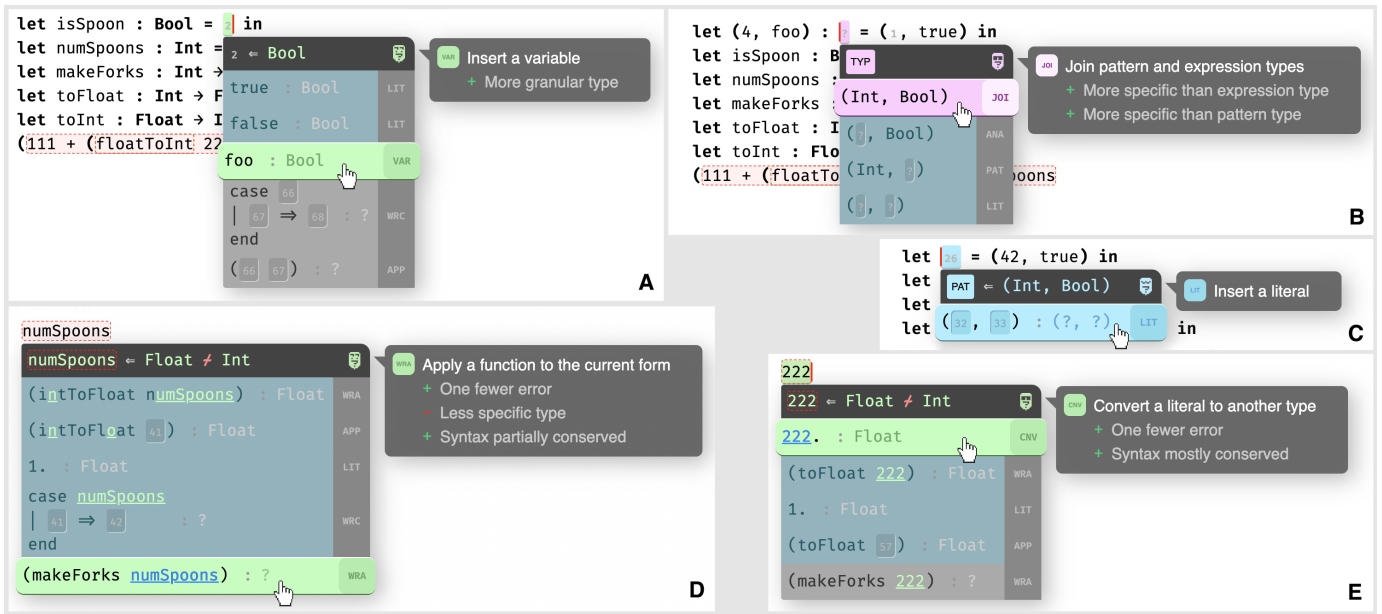


Fig. 2: The Hazel Assistant: (2.A): Completion menu for an expression hole of type `Bool`. The assistant is not limited to expression completion; it can also use type inference to refine type annotations (2.B) and patterns (2.C). (2.D): Wrapper synthesizer being used for code repair. Both the first and last options result in fewer errors, but the last (selected) is ranked down as it results in a less specific type. (2.E): Converter synthesizer targeting numeric type errors

```
type Suggestion = (EditAction, Explanation)
type Synthesizer = LanguageServer -> Set(Suggestion)
```

For example, the standard Java code completion Synthesizer generates field name suggestions by requesting the type of the target expression from the language server, and variations of that synthesizer also incorporate edit history [32], examples [6], or abbreviations [33].

Generated suggestions are collated and assessed by our third layer, the *Scorers*, resulting in reports used by the *Ranker* and *Presenter* to convey suggestions to the user:

```
type Scorer = Suggestion -> Score
type ScoreReport = Map(Scorer, Score)
type RankedSuggestion = (Suggestion, RankExplanation)
type Ranker = Map(Suggestion, ScoreReport)
  -> List(RankedSuggestion)
```

A variety of ranking and sorting methods have been previously considered [4] including alphabetically, by-type, by-relevance, by prevalence in a corpus, and via logical grouping. Explanation of suggestions is under-researched in programming assistants, but has been recognized as increasingly important [34] and *explainable AI* is a burgeoning topic [35].

Finally, the ranked suggestions are presented to the user together with various affordances, i.e. assistant actions, for updating the assistant state, e.g. to sort, filter, or interact with the components just described. While interaction with suggestions is often simply selection from a menu, more involved interaction models include active code completion via palettes [36], interactive example augmentation [23], and work on the Read-Eval-Synthesize-Loop [37], which presents a Read-Eval-Print-Loop-inspired

interaction model for driving human-in-the-loop synthesis.

III. HAZEL ASSISTANT

The Hazel programming environment [24] provides a compelling setting for explorations in programming assistant design. It is a structure editor with a formalized editor action semantics, and it avoids the language server gap problem described above, providing continual static and dynamic analyses even for program sketches [25].

The Hazel Assistant shown in Fig. 2 is a working prototype of a completion and repair assistant for Hazel, serving as a simple end-to-end instantiation of our architecture.

The prototype integrates various Synthesizers that focus on using cursor-local syntactic and static Analyzers to suggest local code transformations. As we are operating on a program sketch—a program with explicit syntactic holes—this task characterization covers both code completion (when the term is an empty hole) and code repair (where the term has a non-empty hole around it, indicating a type error). For type-correct terms, the assistant still provide suggestions for possible transformations, providing lightweight ambient awareness of implementation alternatives.

- **Type Analyzer:** Determines the expected (analytic) type and current (synthetic) type at the cursor
- **Binding Analyzer:** Collects bound variables and uses
- **Syntactic Context Analyzer:** Manages an ascending list of enclosing syntactic forms, rooted at the cursor term and including its parent and ancestors

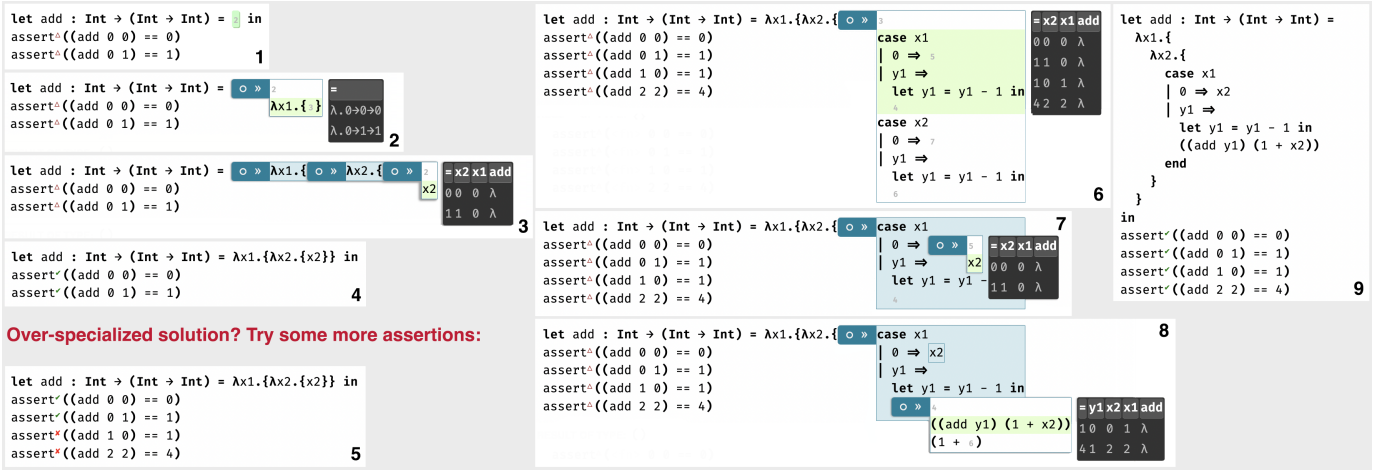


Fig. 3: Hazel Live Assistant: Here we collaborate with the Smyth synthesizer to write a function to add Peano representation integers. Here we are working around the fact that the Smyth synthesizer supports only algebraic data types, which are not supported by Hazel; we translate the successor constructor to “+ 1” and destructure a successor by subtracting 1. (3.1) portrays a stubbed-out function with two user-provided examples. (3.2-3.4) show the process of *stepping through* a synthesis refinement tree: The user is offered a menu of options; at these stages there is only one suggested completion. The black panel displays the unevaluated constraints which must be satisfied. (3.5) shows a finished but overspecialized solution; the user resolves this by stepping out of synthesis and adding two more examples. (3.6-3.8) represent the result of deleting the “x2” reference and resuming synthesis. This time the user has more options; either casing on x1 or x2 (3.6), and adding 1 before or after the recursive call (3.8). (3.9) shows the completed function

These Analyzers are queried by various Synthesizers:

- **Inserter Synthesizers:** These implement basic type-aware code completion, ignoring cursor term content and suggesting a wholesale replacement. For example, at a hole with expected type `Bool`, the Literal Synthesizer suggests `true` and `false` (Fig. 2.A)
- **Wrapper Synthesizers:** These take into account the type of the current term to suggest wrapping it within a larger term. For example the application synthesizer may suggest wrapping a term by applying a function which consumes that term’s type and produces the expected type
- **Converter Synthesizers:** These suggest conversions between types, e.g. `Float/Int` conversions (Fig. 2.E)

Each synthesizer emits a set of suggestions, each equipped with an explanation related to the synthesizer that suggested it. Suggestions are pooled together and fed to the Scorers, who rate each suggestion, possibly with further Language Server consultation. The Hazel Assistant currently employs the following Scorers:

- **Error Delta Scorer:** Determines the integer change in the number of static errors (Fig. 2.D-E)
- **Idiomatcity Scorer:** Performs a heuristic assessment of the idiomatcity of the result, based on a comparison against a fixed list of non-idiomatic syntactic patterns such as using a lambda directly in an application
- **Type Specificity Scorer:** Compares the current and resulting types. This is positive if the resulting type

is more specific, such as moving from ? (unknown) to `Int`

- **Syntax Conservation Scorer:** Compares the string representation of the current term and its suggested replacement via Levenshtein distance

Each suggestion’s scores are collected in a `ScoreReport` and passed to the Ranker, which uses a set score weighting to sort the set of suggestions. The Presenter surfaces the ranked suggestions as a scrollable list. On hover, a suggestion’s explanation is shown as well as an explanation for the ranking in terms of the individual scores. Selecting a suggestion triggers the corresponding `EditAction` and closes the interaction loop. Interactions with the Presenter, e.g. moving up and down, are Assistant Actions and update only the Assistant State.

IV. HAZEL LIVE ASSISTANT

The Hazel Live Assistant extends the Base Assistant by integrating with a more sophisticated external type-and-example-directed synthesizer, Smyth [38]. To support human-in-the-loop synthesis it employs a more complex interaction, with the Smyth synthesizer incorporating non-trivial retained state.

In the Hazel editor, all expressions, including incomplete ones, have well-defined evaluation results. Via a process called live evaluation [25], empty holes are propagated as placeholders into the evaluated result, and ill-typed expressions are partially evaluated by placing them in *non-empty holes* and evaluating around them. Smyth additionally uses *live unevaluation* to propagate the values from

assertions backward through the program as additional synthesis constraints. Thus the Live Assistant demonstrates extending the reach of Analyzers into program execution.

In the small, the Live Assistant behaves similarly to the Base Assistant. When activated on a hole, it presents a list of candidate hole fillings which may be navigated via the up/down arrows. Each entry in this case is not only type compatible, but represents a refinement step which, possibly after further refinements, may result in a term satisfying the provided assertions.

Like the Base Assistant, the Live Assistant explains suggestions. The rows of the black boxes in Figure 3 represent constraints. The column marked '=' indicates the values the current term must take to satisfy the assertions when the variables take on the values indicated in the other columns. Unlike the Base Assistant, the Live Assistant is a suggestion synthesizer that maintains nontrivial state. If the user accepts a non-terminal refinement – that is, a refinement which contains holes – the suggestion is not immediately committed. Rather, the suggestion menu advances to the first contained hole, stepping deeper into the refinement tree. This UI expedites user-directed backtracking, allowing easy exploration of forks in the synthesis process by binding the left/right arrow keys to forward/backward movement in the refinement tree.

This process of flexible exploration is also exploited in the synthesis back-end, as the refinement tree is lazily generated, allowing the possibility of the human in the loop manually resolving chokepoints where the constraints in the code itself do not sufficiently restrict the search space.

V. PAPERCLIP CALCULUS

Here we present a different kind of instantiation of the above architecture: a mathematical formalization of an assistant rather than an implementation, which we term the Paperclip calculus in honor of Clippy, Microsoft’s erstwhile interactive digital assistant [39]. Though this descent into formalism contrasts to the human-factors emphasis of the previous prototypes, we contend that it is essential—particularly as suggestion synthesis draws on more complex methods—that we can offer concrete and specific assurances to users about the meaningfulness of synthesized suggestions.

This prototypical formalism is not intended to offer the last word on what it means for suggestions to be sensible, but instead to serve as a foundational example of defining and demonstrating correctness of an assistant according with our architecture, and to serve as a framing for discussing some design decisions that arise in the process.


A. Review: The Hazelnut Editor Calculus

This time, instead of extending the Hazel editor, we extend its core calculus, Hazelnut as developed by Omar et al. [24], which provides the definitions of the basic typing

and action judgements we’ll be using. We extend this core by adding binary products; the semantics of which are appended as section X:





$$\begin{aligned} \text{HTyp } \tau &::= 1 \mid \tau + \tau \mid \tau \times \tau \mid \tau \rightarrow \tau \mid \emptyset \\ \text{HExp } e &::= () \mid (e, e) \mid \pi_1 e \mid \pi_2 e \mid \text{inl}(e) \mid \text{inr}(e) \\ &\mid \text{case}(e, x.e, y.e) \mid x \mid \lambda x.e \mid e(e) \mid \emptyset \mid (e) \end{aligned}$$

Fig. 4: An extension of Hazelnut’s syntax of H-types and H-expressions. Metavariable x ranges over variables.


Hazelnut is a calculus of editor actions, and provides meta-theorems circumscribing the consequences of these actions; in particular, that each resulting state is not only syntactically well-formed (via empty holes) but also that what would otherwise be ill-typed expressions are enclosed in (non-empty holes). Hazelnut also internalizes  position into the syntactic model, via a syntactic zipper[40], enabling us to suggest cursor positions in the same way as we suggest syntactic transformations. For our purposes we restrict ourselves to the following subset of the action calculus, given here:

$$\begin{aligned} \text{Action } \alpha &::= \leftarrow \hat{e} \mid \text{del} \mid \downarrow \text{move child} \\ \text{ActionList } \bar{\alpha} &::= \cdot \mid \alpha; \bar{\alpha} \end{aligned}$$

Fig. 5: Syntax of actions (adapted from Hazelnut [24])

Our relevant actions include deletion , movement to the first child of an expression , sequencing two actions, and a new construction action  for replacing an empty hole with an arbitrary well-typed expression (in Hazelnut proper, construction is done incrementally, which we depart from here for expediency). The Action Sensibility metatheorem expresses the fact that all actions *on* meaningful edit states *result* in meaningful edit states. We illustrate by means of the below rule, which defines our new construction action .

$$\begin{array}{c} \text{ConstructExp} \\ \alpha = \leftarrow \hat{e}' \quad \Gamma \vdash \hat{e}' \Leftarrow \tau \\ \hline \Gamma \vdash \leftarrow \hat{e}' \Leftarrow \tau \end{array}$$

The conclusion of this rule (below the line) states that, given some typing assumptions Γ , and an empty hole \emptyset under the , our action α replaces that hole with a new censored expression \hat{e}' satisfying τ , the expected type at that position. The premise of the rule ($\Gamma \vdash \hat{e}' \Leftarrow \tau$) indicates that this action judgement holds precisely when the new expression (with the cursor removed^o) analyzes against τ .

B. Suggestion Synthesis Judgement

We are now in position to define a formal judgement to characterize suggestion synthesis. First a note on a vocabulary collision: Hazelnut uses a bidirectional typing system[41], where each syntactic position is either analytic (the parent form imposes an expected type) or synthetic

(the type is determined by the expression itself), assuming a typing context Γ mapping variables to types. This sense of 'type synthesis' is related but distinct from our use of 'suggestion synthesis' to denote the process of generating suggestions.

In keeping with Hazelnut's type system, we have two forms of our suggestion synthesis judgement, expressed symbolically in the following figure. Recall that our expressions \hat{e} bake in a cursor, so Γ and τ below should be interpreted as the static information contextual to that position:

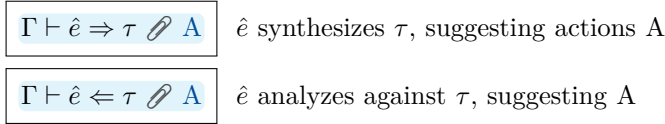


Fig. 6: Suggestion Judgement Forms; here A is a set of Hazelnut action sequences

Now we must decide what it means for a suggested action to be sensible. To set a bar, we want suggested syntax to be well-formed. Type correctness may also seem like a given, but note that it is not completely obvious where this kind of sensibility should be enforced: Do we pursue correctness-by-construction, insisting that our synthesizers produce only type-correct suggestions? Or are we more liberal in our synthesis, and use sensibility either as a hard filter on suggestions, or simply as a ranking criteria?

For simplicity, we choose correctness at time of synthesis, and claim that a suggestion is sensible precisely when (1) its action can be performed successfully and thus preserves static meaningfulness in Hazelnut's sense, and (2) that the number of type errors (non-empty holes, denoted $\#e$) is not increased:

Theorem 1 (Suggestion sensibility):

- For all Γ, \hat{e}, τ, A where $\Gamma \vdash \hat{e}^\circ \Leftarrow \tau$ and $\Gamma \vdash \hat{e} \Leftarrow \tau \text{ } \textcircled{A}$ there exists an \hat{e}' such that for all $\alpha \in A$ we have $\Gamma \vdash \hat{e} \xrightarrow{\alpha} \hat{e}' \Leftarrow \tau$ and $\#e^\circ \leq \#e'^\circ$
- For all Γ, \hat{e}, τ, A where $\Gamma \vdash \hat{e}^\circ \Rightarrow \tau$ and $\Gamma \vdash \hat{e} \Rightarrow \tau \text{ } \textcircled{A}$ there exists an \hat{e}' and a τ' such that for all $\alpha \in A$ we have $\Gamma \vdash \hat{e} \Rightarrow \tau \xrightarrow{\alpha} \hat{e}' \Rightarrow \tau'$ and $\#e^\circ \leq \#e'^\circ$

In the following section we develop the rules to fully articulate the suggestion judgment, and in the appendix (XII) we prove Theorem 1.

C. Analytic Suggestion Judgement: Completions

The first of three base cases for the analytic suggestion judgement, the `SuggestCompleteAna` rule details the synthesis of basic code completions, corresponding to the case where the cursor is on an empty hole:

$$\frac{\text{SuggestCompleteAna}}{\Gamma \vdash \triangleright \langle \rangle \Leftarrow \tau \text{ } \textcircled{\text{Intros}(\tau) \cup \text{Elims}(\Gamma, \tau)}}$$

This rule collects the results of two synthesizer meta-functions, the `Intros` and `Elims`. Such meta-functions take static information localized to the cursor—here some subset of the expected type τ , the typing context Γ , and the indicated subexpression e —to produce a set of suggested actions.

First, the `Intros` synthesizer below suggests introduction forms; for example, if the expected type τ is a product $\tau_1 \times \tau_2$, it suggests constructing a pair of holes, advancing the cursor to the first hole ($\triangleright \langle \rangle \langle \rangle$):

$$\text{Intros}(\tau) =$$

$$\left\{ \begin{array}{ll} \{ \leftarrow \triangleright \langle \rangle \} & \tau = 1 \\ \{ \leftarrow \lambda x. \triangleright \langle \rangle \} & \tau = \tau_1 \rightarrow \tau_2 \\ \{ \leftarrow \langle \triangleright \langle \rangle \rangle \} & \tau = \tau_1 \times \tau_2 \\ \{ \leftarrow \text{inl}(\triangleright \langle \rangle), \leftarrow \text{inr}(\triangleright \langle \rangle) \} & \tau = \tau_1 + \tau_2 \\ \text{Intros}(1) \cup \\ \text{Intros}(\langle \rangle \times \langle \rangle) \cup \\ \text{Intros}(\langle \rangle + \langle \rangle) \cup \\ \text{Intros}(\langle \rangle \rightarrow \langle \rangle) & \tau = \langle \rangle \end{array} \right.$$

Second, the `Elims` synthesizer collates three genres of suggestion, corresponding to Hazelnut's elimination forms:

$$\text{Elims}(\Gamma, \tau) = \text{Case} \cup \text{Var}(\Gamma, \tau) \cup \text{AP}(\Gamma, \tau)$$

First, `Case` suggests simply inserting a case expression:

$$\text{Case} = \{ \leftarrow \text{case}(\triangleright \langle \rangle, x. \langle \rangle, y. \langle \rangle) \}$$

Second, `Var` suggests referencing a variable drawn from the context Γ of a type τ' consistent with expected type τ :

$$\text{Var}(\Gamma, \tau) = \{ \leftarrow \triangleright x \langle \rangle \mid x : \tau' \in \Gamma, \tau \sim \tau' \}$$

Finally, our most complex synthesizer, `AP` essentially performs a deep search of the context Γ , looking for variables with compound types which can be eliminated down to the expected type τ . In particular, it identifies (nested) product and arrow types which, after appropriate projections and function applications, produce the desired type. For example, if we had $f : A \rightarrow (B \times C)$ in Γ , and our expected type was $\tau = B$, `AP` would suggest $\pi_1 f(\langle \rangle)$ as a completion.

$$\text{AP}(\Gamma, \tau) = \{ \leftarrow \triangleright e \langle \rangle \mid x : \tau' \in \Gamma, \Gamma \vdash x \rightsquigarrow e \Leftarrow \tau, \tau \sim \tau' \}$$

This synthesizer requires the specification of an additional judgement form, used to characterize its suggestions:

$$\Gamma \vdash e \rightsquigarrow e' \Leftarrow \tau \quad e \text{ ap-projects to } e' \text{ analyzing to } \tau$$

More verbosely, the ap-projection judgement asserts that e , applied to 0 or more holes, and projected 0 or more times, yields an expression e' analyzing against τ . The base case rule for this judgement simply says that the

judgement is already satisfied by an expression of the provided type.

$$\frac{\text{AP-Base} \quad \Gamma \vdash e \Leftarrow \tau}{\Gamma \vdash e \rightsquigarrow e \Leftarrow \tau}$$

Otherwise, if the input expression is a product or a function, and when projected or applied to a hole, the resulting expression satisfies the judgement, then the original expression also satisfies the judgement.

$$\frac{\text{AP-App} \quad \Gamma \vdash e \Rightarrow _ \rightarrow _ \quad \Gamma \vdash e(\emptyset) \rightsquigarrow e' \Leftarrow \tau}{\Gamma \vdash e \rightsquigarrow e' \Leftarrow \tau}$$

$$\frac{\text{AP-Proj1/2} \quad \Gamma \vdash e \Rightarrow _ \times _ \quad \Gamma \vdash \pi_{1/2}e \rightsquigarrow e' \Leftarrow \tau}{\Gamma \vdash e \rightsquigarrow e' \Leftarrow \tau}$$

(Note that this unlike the other judgements is a non-deterministic, non-algorithmic characterization; to implement this a particular search strategy would need to be chosen)

D. Analytic Suggestion Judgement: Repair & Modification

Our remaining two base cases detail the synthesis of repairs and of other modifications. These correspond to the cases where the cursor is either on a non-empty hole, or some other non-hole expression.

When the cursor is on a non-empty hole, we suggest repair options which transform the contained expression into one agreeing with the expected type. For this we substantially defer to the third and final base case, as shown in the rule's premise:

$$\frac{\text{SuggestRepairAna} \quad \Gamma \vdash \triangleright e \triangleleft \Leftarrow \tau \text{ } \mathcal{A}}{\Gamma \vdash \triangleright (e) \triangleleft \Leftarrow \tau \text{ } \{ \downarrow \text{ move child } ; \alpha \mid \alpha \in \mathcal{A} \}}$$

The final base case describes suggestions which modify an existing expression in-place, either by replacing it wholesale by an expression of the appropriate type, or wrapping it in a function application.

$$\frac{\text{SuggestModifyAna} \quad e \neq \emptyset \quad e \neq _}{\Gamma \vdash \triangleright e \triangleleft \Leftarrow \tau \text{ } \mathcal{A} \cup \text{Wraps}(\Gamma, e, \tau) \cup \text{Replaces}(\Gamma, \tau)}$$

The `Replaces` synthesizer suggests a compound action: a deletion followed by another action α drawn from all the possible suggestions for the resulting empty hole $\triangleright \emptyset \triangleleft$:

$$\text{Replaces}(\Gamma, \tau) = \{ \text{del} ; \alpha \mid \alpha \in \mathcal{A}, \Gamma \vdash \triangleright \emptyset \triangleleft \Leftarrow \tau \text{ } \mathcal{A} \}$$

The `Wraps` synthesizer suggests wrapping the indicated subexpression $\triangleright e \triangleleft$ with a function f drawn from Γ pro-

vided it has input and output types consistent with the context and contained expression:

$$\text{Wraps}(\Gamma, e, \tau) = \{ \leftarrow f(\triangleright e \triangleleft) \mid \Gamma \vdash e \Rightarrow \tau_e, f : \tau_{in} \rightarrow \tau_{out} \in \Gamma, \tau_{in} \sim \tau_e, \tau \sim \tau_{out} \}$$

E. Suggestion Judgement: Zipper Cases

Having described our suggestion synthesizers with respect to the scope and typing information present at the cursor, it remains to detail our inductive (zipper) cases, which describe how that information is propagated down the syntax tree to the cursor. There is at least one such case for each syntactic position of every form. These rules make use of some subsidiary typing judgements defined in [24], and indeed have a close correspondence to the rules given there, as those are ultimately the rules used determine action and hence suggestion sensibility. First have the core rules determining the propagation of expected types through function literals and applications:

$$\frac{\text{SZLam} \quad \tau \blacktriangleright \rightarrow \tau_1 \rightarrow \tau_2 \quad \Gamma, x : \tau_1 \vdash \hat{e} \Leftarrow \tau_2 \text{ } \mathcal{A}}{\Gamma \vdash \lambda x. \hat{e} \Leftarrow \tau \text{ } \mathcal{A}}$$

$$\frac{\text{SZAppR} \quad \Gamma \vdash e_1 \Rightarrow \tau_1 \quad \tau_1 \blacktriangleright \rightarrow \tau_2 \rightarrow \tau \quad \Gamma \vdash \hat{e}_2 \Leftarrow \tau_2 \text{ } \mathcal{A}}{\Gamma \vdash e_1(\hat{e}_2) \Rightarrow \tau \text{ } \mathcal{A}}$$

$$\frac{\text{SZAppL} \quad \begin{array}{l} \mathcal{A}^* = \{ \alpha \in \mathcal{A} \mid \text{if } \Gamma \vdash \hat{e}_1 \Rightarrow \tau_1 \xrightarrow{\alpha} \hat{e}'_1 \Rightarrow \tau' \\ \text{then } \tau' \sim (\tau_2 \rightarrow \emptyset) \} \\ \Gamma \vdash \hat{e}_1 \Rightarrow \tau_1 \text{ } \mathcal{A} \quad \tau_1 \blacktriangleright \rightarrow \tau_2 \rightarrow \tau \quad \Gamma \vdash e_2 \Leftarrow \tau_2 \end{array}}{\Gamma \vdash \hat{e}_1(e_2) \Leftarrow \tau \text{ } \mathcal{A}^*}$$

Note that in the above case for suggestions in the function position of an application, we need to filter the suggestions owing to a technicality of the bidirectional type system; despite this being a synthetic position, there are still type consistency restrictions. In particular, we must ensure that exp_1 does indeed have function type, and that its parameter type is consistent with τ_2 , the type of the argument.

Here the rules for binary products, determining the type information flow for projections:

$$\frac{\text{SZPrj1} \quad \Gamma \vdash \hat{e} \Rightarrow \tau_1 \times \tau_2 \text{ } \mathcal{A}}{\Gamma \vdash \pi_1 \hat{e} \Rightarrow \tau_1 \text{ } \mathcal{A}} \quad \frac{\text{SZPrj2} \quad \Gamma \vdash \hat{e} \Rightarrow \tau_1 \times \tau_2 \text{ } \mathcal{A}}{\Gamma \vdash \pi_2 \hat{e} \Rightarrow \tau_2 \text{ } \mathcal{A}}$$

and for pairs:

$$\text{SZSynPair1} \quad \frac{\Gamma \vdash \hat{e}_1 \Rightarrow \tau_1 \text{ } \wp \text{ } A}{\Gamma \vdash (\hat{e}_1, e_2) \Rightarrow \tau_1 \times \tau_2 \text{ } \wp \text{ } A}$$

$$\text{SZSynPair2} \quad \frac{\Gamma \vdash \hat{e}_2 \Rightarrow \tau_1 \text{ } \wp \text{ } A}{\Gamma \vdash (e_1, \hat{e}_2) \Rightarrow \tau_1 \times \tau_2 \text{ } \wp \text{ } A}$$

$$\text{SZAnaPair1} \quad \frac{\tau \blacktriangleright_{\times} \tau_1 \times \tau_2 \quad \Gamma \vdash \hat{e}_2 \Leftarrow \tau_1 \text{ } \wp \text{ } A}{\Gamma \vdash (e_1, \hat{e}_2) \Leftarrow \tau \text{ } \wp \text{ } A}$$

$$\text{SZAnaPair2} \quad \frac{\tau \blacktriangleright_{\times} \tau_1 \times \tau_2 \quad \Gamma \vdash \hat{e}_1 \Leftarrow \tau_1 \text{ } \wp \text{ } A}{\Gamma \vdash (\hat{e}_1, e_2) \Leftarrow \tau \text{ } \wp \text{ } A}$$

And finally the rules for binary sums, determining information flow through injections:

$$\text{SZInL} \quad \frac{\tau \blacktriangleright_{+} \tau_L + \tau_R \quad \Gamma \vdash \hat{e} \Leftarrow \tau_L \text{ } \wp \text{ } A}{\Gamma \vdash \text{inl}(\hat{e}) \Leftarrow \tau \text{ } \wp \text{ } A}$$

$$\text{SZInR} \quad \frac{\tau \blacktriangleright_{+} \tau_L + \tau_R \quad \Gamma \vdash \hat{e} \Leftarrow \tau_R \text{ } \wp \text{ } A}{\Gamma \vdash \text{inr}(\hat{e}) \Leftarrow \tau \text{ } \wp \text{ } A}$$

and through case expressions

$$\text{SZCaseL} \quad \frac{\Gamma, x : \tau_L \vdash \hat{e}_L \Leftarrow \tau \text{ } \wp \text{ } A \quad \Gamma \vdash e \Rightarrow \tau_+ \quad \tau_+ \blacktriangleright_{+} \tau_L + \tau_R}{\Gamma \vdash \text{case}(e, x.\hat{e}_L, y.e_R) \Leftarrow \tau \text{ } \wp \text{ } A}$$

$$\text{SZCaseR} \quad \frac{\Gamma, y : \tau_R \vdash \hat{e}_R \Leftarrow \tau \text{ } \wp \text{ } A \quad \Gamma \vdash e \Rightarrow \tau_+ \quad \tau_+ \blacktriangleright_{+} \tau_L + \tau_R}{\Gamma \vdash \text{case}(e, x.e_L, y.\hat{e}_R) \Leftarrow \tau \text{ } \wp \text{ } A}$$

$$\text{SZCase0} \quad \text{A}^* = \{\alpha \in A \mid \text{if } \Gamma \vdash \hat{e} \Rightarrow \tau_+ \xrightarrow{\alpha} \hat{e}' \Rightarrow \tau'_+ \text{ then } \tau'_+ \sim (\tau_L + \tau_R)\}$$

$$\frac{\Gamma \vdash \hat{e} \Rightarrow \tau_+ \text{ } \wp \text{ } A \quad \tau_+ \blacktriangleright_{+} \tau_L + \tau_R \quad \Gamma, x : \tau_L \vdash e_L \Leftarrow \tau \quad \Gamma, y : \tau_R \vdash e_R \Leftarrow \tau}{\Gamma \vdash \text{case}(\hat{e}, x.e_L, y.e_R) \Leftarrow \tau \text{ } \wp \text{ } A^*}$$

Note that the above case, similarly to applications, must ensure that suggestions for the case scrutinee expression e are consistent with branch expressions e_L and e_R .

There remains the synthetic base case; the synthetic case in general is non-revealingly technical owing to certain

properties of Hazelnut and bidirectional typing system in general; a detailed treatment including two remaining rules is deferred to an appendix (XI). Otherwise, this concludes the specification of the suggestion judgement. See appendix XII for a proof that it satisfies the suggestion sensibility theorem.

F. Scorers & Rankers Example: Error Delta

Beyond suggestion synthesis, we have provide a simple judgement formalizing of one scoring/ranking criteria, namely the static error count, which in Hazelnut corresponds precisely to the number of non-empty holes:

$$\boxed{\#e = n} \quad \text{expression } e \text{ contains } n \text{ non-empty holes}$$

The main rule states simply that a non-empty hole expression itself has one more error than the contained expression:

$$\text{ErrorsNonEmpty} \quad \frac{\#e = n}{\#(e) = n + 1}$$

Atomic expressions on the other hand have an error count of 0; in particular, we do not consider incompleteness, i.e. an empty hole, to be an error in this formulation:

$$\begin{array}{ccc} \text{ErrorsHole} & \text{ErrorsVar} & \text{ErrorsTriv} \\ \hline \#() = 0 & \#x = 0 & \#() = 0 \end{array}$$

For all other types of composite syntax, we merely combine the errors counts from the children:

$$\begin{array}{cc} \text{ErrorsPair} & \text{ErrorsAp} \\ \frac{\#e_1 = n_1 \quad \#e_2 = n_2}{\#(e_1, e_2) = n_1 + n_2} & \frac{\#e_1 = n_1 \quad \#e_2 = n_2}{\#e_1(e_2) = n_1 + n_2} \end{array}$$

$$\begin{array}{cccc} \text{ErrorsPrjL} & \text{ErrorsPrjR} & \text{ErrorsInL} & \text{ErrorsInR} \\ \frac{\#e = n}{\#\pi_1 e = n} & \frac{\#e = n}{\#\pi_2 e = n} & \frac{\#e = n}{\#\text{inl}(e) = n} & \frac{\#e = n}{\#\text{inr}(e) = n} \end{array}$$

$$\begin{array}{ccc} \text{ErrorsLam} & \text{ErrorsCase} & \\ \frac{\#e = n}{\#\lambda x.e = n} & \frac{\#e_1 = n_1 \quad \#e_2 = n_2 \quad \#e_3 = n_3}{\#\text{case}(e_1, x.e_2, y.e_3) = n_1 + n_2 + n_3} & \end{array}$$

Again, this single ranking criteria is not intended to be sufficient, but to serve as basic illustration. Similar formalizations could be developed for the syntactic conservation criteria, or other more sophisticated metrics.

VI. RELATED WORK

In a general sense, work in this era goes back at least to Licklider's *Man-Machine Symbiosis* [42], a prescient argument for the promise of enhanced problem-solving via the union of human intent and discernment with machines' facility for repetitive activity, error avoidance, and precise recall.

We distinguish in particular some previous work with an explicitly architectural tack:

- The *Programmer’s Apprentice* initiative at MIT [1–3] in the 80s-90s constitutes some of the earliest organized work in this space. This project sought to automate repetitive programming tasks via classical AI and knowledge representation, including the theories of planning and frames.
- Recent work on *Language-parametric Static Semantic Code Completion* [26] presents a framework for code completion based on LSP-style parametric specification of language syntax and semantics. Similar to our assistant calculus, it establishes formal criteria of soundness and completeness to demonstrate the correctness of completions. While this work is more linguistically general, it focuses entirely on completion and not repair or modification, and has a more limited treatment of incomplete program sketches.

There is a large body of literature on specific methods of suggestion synthesis, as well as architectural approaches to interactive assistants for other kinds of computing activities. Many of these are referenced in the introduction; we select some highlights here:

- From the programming synthesis literature we have work focusing on both the presentation stage and on synthesis techniques to interactively enhance user discernment of suggestions, including on interpretability of suggestions [22], ambiguity resolution via generated examples [23], affordances for granular user feedback to refine suggestions [43], and REPL-style dialogic in-place synthesis [37]
- Long a mainstay of the Agda proof assistant, work on type-directed hole-filling of program sketches in production programming languages includes [16], [15], and [44]. *Type-Directed Completion of Partial Expressions* [45] covers type-directed completion on explicitly incomplete program sketches featuring a sophisticated discussion of ranking, including a notion of *type distance* similar to our *type specificity*.
- In data science research there is an analogous architecture with a human-centric focus, instead specializing on data cleaning tasks [46]. In particular it addresses combining the results of heterogeneous analyses and supporting human-in-the-loop processes.
- Work which is editor-action focused, particular on the topic of using sequences of actions to inform suggestion synthesis, includes Blue Pencil [12], Overwatch [47], and CoditT5 [48].
- Synthesis work focusing on user interaction models [49] may help users disambiguate between large numbers of options resulting from underspecification.
- *Best-effort program synthesis* [50], on the other hand, presents a system which attempts to compensate for overspecified or otherwise erroneous example sets by generating and ranking partially-valid results.

- On the issue of correctness concerns with LLM-based suggestion synthesizers like OpenAI’s Codex, *Interactive Code Generation via Test-Driven User-Intent Formalization* [51] presents a workflow based on test-driven user intent formalization, prompting users to confirm or reject input/output examples derived from synthesized suggestions. See [52] for other work in this space.

VII. FUTURE DIRECTIONS

Our own design efforts remain ongoing. One key direction is to directly incorporate AI techniques, in particular deep reinforcement learning, which is also oriented around an action space and a reward/scoring structure (human acceptance, tests passing, type errors resolution). We are currently running experiments training an RL agent to perform basic example-based completion tasks in a modified Hazel environment, where the agent’s action space is both enriched and circumscribed by type information.

We are also working to incorporate more complex multistage refactorings via interactive monadic edit actions, extending work on edit-time tactics in dependently-typed languages [53] and in proof assistants, in particular the Mtac [54] approach to custom proof automation in Coq. On the architectural side, we seek to explore and incorporate interfaces for the temporally and spatially-extended process of updating code after changes to type definitions, and, conversely, suggesting changes in type definitions based on intent expressed via changes to expressions. On the mathematical side, this consists of extending our formalism in a direction analogous to Hoare triples in order to support proving that relevant invariants are maintained as multiple dependent steps of human and machine actions are interleaved.

VIII. CONCLUSION

This paper attempts to organize the burgeoning area of programming assistant design with a high level architecture and terminology, and then demonstrates the feasibility of this architecture for design explorations with three prototype assistants, extending both the Hazel development environment and its underlying editor calculus. These design explorations highlight design criteria that are perhaps understudied: semantic ranking, integration, explanations, and human-in-the-loop interactions with synthesizers. We presented a formal model for specifying suggestion synthesizers, and provide a characterization of suggestion sensibility and a worked-out proof that our example model satisfies that characterization. We hope that this work helps to bring together various communities working on individual components of the overall system design and ultimately to tap the creative and collaborative potential of human-in-the-loop programming assistants.

IX. ACKNOWLEDGEMENTS

We would like to thank Xinyu Wang for his encouragement and support; the Hazel Live Assistant began as a

project in his program synthesis class. We would also like to thank Justin Lubin for his patient advice in integrating the Smyth synthesizer.

REFERENCES

- [1] C. Rich, H. E. Shrobe, and R. C. Waters, “Overview of the Programmer’s Apprentice,” in *Sixth International Joint Conference on Artificial Intelligence, IJCAI 79*, 1979, pp. 827–828.
- [2] C. Rich and H. E. Shrobe, “Initial Report on a Lisp Programmer’s Apprentice,” *IEEE Trans. Software Eng.*, vol. 4, no. 6, pp. 456–467, 1978. [Online]. Available: <https://doi.org/10.1109/TSE.1978.233869>
- [3] H. E. Shrobe, B. Katz, and R. Davis, “Towards a Programmer’s Apprentice (Again),” in *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015, pp. 4062–4066.
- [4] D. Hou and D. M. Pletcher, “An Evaluation of the Strategies of Sorting, Filtering, and Grouping API Methods for Code Completion,” in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2011, pp. 233–242.
- [5] R. Robbes and M. Lanza, “How Program History Can Improve Code Completion,” in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2008, pp. 317–326.
- [6] M. Bruch, M. Monperrus, and M. Mezini, “Learning from Examples to Improve Code Completion Systems,” in *7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium*, 2009, pp. 213–222.
- [7] G. C. Murphy, M. Kersten, and L. Findlater, “How Are Java Software Developers Using the Eclipse IDE?” *IEEE Software*, vol. 23, no. 4, pp. 76–83, 2006.
- [8] S. Gulwani, O. Polozov, and R. Singh, “Program Synthesis,” *Found. Trends Program. Lang.*, vol. 4, no. 1-2, pp. 1–119, 2017. [Online]. Available: <https://doi.org/10.1561/25000000010>
- [9] P.-M. Osera and S. Zdancewic, “Type-and-Example-Directed Program Synthesis,” *Programming Language Design and Implementation (PLDI)*, vol. 50, no. 6, pp. 619–630, 2015.
- [10] J. Frankle, P.-M. Osera, D. Walker, and S. Zdancewic, “Example-Directed Synthesis: A Type-Theoretic Interpretation,” in *Symposium on Principles of Programming Languages (POPL)*, 2016.
- [11] A. Solar-Lezama, “Program Sketching,” *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 5, pp. 475–495, 2013.
- [12] A. Miltner, S. Gulwani, V. Le, A. Leung, A. Radhakrishna, G. Soares, A. Tiwari, and A. Udupa, “On the Fly Synthesis of Edit Suggestions,” *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, oct 2019. [Online]. Available: <https://doi.org/10.1145/3360569>
- [13] T. A. Lau, P. M. Domingos, and D. S. Weld, “Version Space Algebra and its Application to Programming by Demonstration,” in *ICML*, 2000, pp. 527–534.
- [14] N. Polikarpova, I. Kuraj, and A. Solar-Lezama, “Program Synthesis From Polymorphic Refinement Types,” *Programming Language Design and Implementation (PLDI)*, vol. 51, no. 6, pp. 522–538, 2016.
- [15] M. P. Gissurarson, “Suggesting Valid Hole Fits for Typed-Holes (Experience Report),” *ACM SIGPLAN International Symposium on Haskell 11*, vol. 53, no. 7, pp. 179–185, 2018.
- [16] —, “The Hole Story: Type-Driven Synthesis and Repair,” Licentiate Thesis, 2022.
- [17] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, “Evaluating Large Language Models Trained on Code,” 2021. [Online]. Available: <https://arxiv.org/abs/2107.03374>
- [18] E. C. R. Shin, M. Allamanis, M. Brockschmidt, and A. Polozov, “Program Synthesis and Semantic Parsing with Learned Code Idioms,” in *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems (NeurIPS)*, H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. B. Fox, and R. Garnett, Eds., 2019, pp. 10 824–10 834.
- [19] V. Raychev, M. Vechev, and E. Yahav, “Code Completion with Statistical Language Models,” in *Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery, 2014, p. 419–428. [Online]. Available: <https://doi.org/10.1145/2594291.2594321>
- [20] J. Li, Y. Wang, I. King, and M. R. Lyu, “Code Completion with Neural Attention and Pointer Networks,” *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI’18)*, 2017. [Online]. Available: <http://arxiv.org/abs/1711.09573>
- [21] C. Liu, X. Wang, R. Shin, J. E. Gonzalez, and D. Song, “Neural Code Completion,” 2016.
- [22] T. Zhang, Z. Chen, Y. Zhu, P. Vaithilingam, X. Wang, and E. L. Glassman, “Interpretable Program Synthesis,” in *2021 CHI Conference on Human Factors in Computing Systems*, ser. CHI ’21. Association for Computing Machinery, 2021. [Online]. Available: <https://doi.org/10.1145/3411764.3445646>
- [23] T. Zhang, L. Lowmanstone, X. Wang, and E. L. Glassman, *Interactive Program Synthesis by Augmented Examples*. UIST, 2020, p. 627–648. [Online]. Available: <https://doi.org/10.1145/3379337.3415900>
- [24] C. Omar, I. Voysey, M. Hilton, J. Aldrich, and M. A. Hammer, “Hazelnut: A Bidirectionally Typed Structure Editor Calculus,” in *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2017.
- [25] C. Omar, I. Voysey, R. Chugh, and M. A. Hammer, “Live Functional Programming with Typed Holes,” *Proceedings of the ACM on Programming Languages (PACMPL)*, Issue POPL, 2019.
- [26] D. A. Pelsmaeker, H. van Antwerpen, C. B. Poulsen, and E. Visser, “Language-parametric static semantic code completion,” *Proceedings of the ACM on Programming Languages (PACMPL)*, Issue OOPSLA, 2022.
- [27] H. Potter and C. Omar, “Hazel Tutor: Guiding Novices Through Type-Driven Development Strategies,” *Human Aspects of Types and Reasoning Assistants (HATRA)*, 2020.
- [28] F. Bour, T. Refis, and G. Scherer, “Merlin: A Language Server for OCaml (Experience Report),” *Proc. ACM Program. Lang.*, vol. 2, no. ICFP, jul 2018. [Online]. Available: <https://doi.org/10.1145/3236798>
- [29] C. Omar, I. Voysey, M. Hilton, J. Sunshine, C. Le Goues, J. Aldrich, and M. A. Hammer, “Toward Semantic Foundations for Program Editors,” in *Summit on Advances in Programming Languages (SNAPL)*, 2017.
- [30] S. L. Graham, C. B. Haley, and W. N. Joy, “Practical LR Error Recovery,” in *SIGPLAN Symposium on Compiler Construction (CC)*, 1979.
- [31] M. de Jonge, E. Nilsson-Nyman, L. C. L. Kats, and E. Visser, “Natural and Flexible Error Recovery for Generated Parsers,” in *Software Language Engineering (SLE)*, 2009.
- [32] R. Robbes and M. Lanza, “How Program History Can Improve Code Completion,” in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008, pp. 317–326.
- [33] S. Han, D. R. Wallace, and R. C. Miller, “Code Completion from Abbreviated Input,” in *2009 IEEE/ACM International Conference on Automated Software Engineering*, 2009, pp. 332–343.
- [34] H. Finkel and I. Laguna, “Report of the Workshop on Program Synthesis for Scientific Computing,” 2021. [Online]. Available: <https://arxiv.org/abs/2102.01687>
- [35] D. Doran, S. Schulz, and T. R. Besold, “What Does Explainable AI Really Mean? A New Conceptualization of Perspectives,” *arXiv preprint arXiv:1710.00794*, 2017.

- [36] C. Omar, Y. S. Yoon, T. D. LaToza, and B. A. Myers, “Active Code Completion,” in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 859–869.
- [37] H. Peleg, R. Gabay, S. Itzhaky, and E. Yahav, “Programming with a Read-Eval-Synth Loop,” *OOPSLA*, vol. 4, nov 2020. [Online]. Available: <https://doi.org/10.1145/3428227>
- [38] J. Lubin, N. Collins, C. Omar, and R. Chugh, “Program Sketching with Live Bidirectional Evaluation,” *Proceedings of the ACM on Programming Languages*, vol. 4, no. ICFP, pp. 1–29, 2020.
- [39] L. Swartz, “Why people hate the paperclip: Labels, appearance, behavior, and social responses to user interface agents,” Ph.D. dissertation, Citeseer, 2003.
- [40] G. Huet, “The zipper,” *Journal of functional programming*, 1997.
- [41] B. C. Pierce and D. N. Turner, “Local type inference,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2000.
- [42] J. Licklider, “Man-machine symbiosis,” 1960.
- [43] H. Peleg, S. Shoham, and E. Yahav, “Programming not only by example,” in *ICSE*, 2018.
- [44] P. Redmond, G. Shen, and L. Kuper, “Toward hole-driven development with liquid haskell,” *arXiv preprint arXiv:2110.04461*, 2021.
- [45] D. Perelman, S. Gulwani, T. Ball, and D. Grossman, “Type-directed completion of partial expressions,” in *Conference on Programming Language Design and Implementation (PLDI)*, 2012.
- [46] E. K. Rezig, M. Ouzzani, A. K. Elmagarmid, W. G. Aref, and M. Stonebraker, “Towards an end-to-end human-centric data cleaning framework,” in *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, 2019.
- [47] Y. Zhang, Y. Bajpai, P. Gupta, A. Ketkar, M. Allamanis, T. Barik, S. Gulwani, A. Radhakrishna, M. Raza, G. Soares *et al.*, “Overwatch: Learning patterns in code edit sequences,” *arXiv preprint arXiv:2207.12456*, 2022.
- [48] J. Zhang, S. Panthaplackel, P. Nie, J. J. Li, and M. Gligoric, “Coditt5: Pretraining for source code and natural language editing,” in *37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.
- [49] M. Mayer, G. Soares, M. Grechkin, V. Le, M. Marron, O. Polozov, R. Singh, B. Zorn, and S. Gulwani, “User Interaction Models for Disambiguation in Programming by Example,” in *Symposium on User Interface Software and Technology (UIST)*, 2015.
- [50] H. Peleg and N. Polikarpova, “Perfect is the enemy of good: Best-effort program synthesis,” *Leibniz international proceedings in informatics*, 2020.
- [51] S. K. Lahiri, A. Naik, G. Sakkas, P. Choudhury, C. von Veh, M. Musuvathi, J. P. Inala, C. Wang, and J. Gao, “Interactive code generation via test-driven user-intent formalization,” *arXiv preprint*, 2022.
- [52] D. Wong, A. Kothig, and P. Lam, “Exploring the verifiability of code generated by github copilot,” *arXiv preprint arXiv:2209.01766*, 2022.
- [53] J. Korkut, “Edit-Time Tactics in Idris,” Master’s thesis, Wesleyan University, 2018.
- [54] J.-O. Kaiser, B. Ziliani, R. Krebbers, Y. Régis-Gianas, and D. Dreyer, “Mtac2: Typed tactics for backward reasoning in coq,” *ICFP*, 2018.

X. APPENDIX: HAZELNUT PRODUCTS EXTENSION

The following extends the statics and action rules for Hazelnut[24], continuing the original numbering scheme.

$\boxed{\tau_1 \sim \tau_2}$ τ_1 is consistent with τ_2

$$\frac{27}{\tau_1 \sim \tau'_1 \quad \tau_2 \sim \tau'_2} \tau_1 \times \tau_2 \sim \tau'_1 \times \tau'_2$$

$\boxed{\tau \blacktriangleright \times \tau_1 \times \tau_2}$ τ has matched product type $\tau_1 \times \tau_2$

$$\frac{28a}{\emptyset \blacktriangleright \times \emptyset \times \emptyset} \quad \frac{28b}{\tau_1 \times \tau_2 \blacktriangleright \times \tau_1 \times \tau_2}$$

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

$$\frac{29}{\Gamma \vdash () \Rightarrow 1} \quad \frac{30a}{\Gamma \vdash e \Rightarrow \tau_1 \times _} \quad \frac{30b}{\Gamma \vdash e \Rightarrow _ \times \tau_2} \quad \frac{30c}{\Gamma \vdash e_1 \Rightarrow \tau_1 \quad \Gamma \vdash e_2 \Rightarrow \tau_2} \Gamma \vdash (e_1, e_2) \Rightarrow \tau_1 \times \tau_2$$

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

$$\frac{31}{\tau \blacktriangleright \times \tau_1 \times \tau_2 \quad \Gamma \vdash e_1 \Leftarrow \tau_1 \quad \Gamma \vdash e_2 \Leftarrow \tau_2} \Gamma \vdash (e_1, e_2) \Leftarrow \tau$$

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

$$\frac{32a}{\Gamma \vdash \hat{e} \Rightarrow \tau_1 \times _ \xrightarrow{\alpha} \hat{e}' \Rightarrow \tau'_1 \times _} \Gamma \vdash \pi_1 \hat{e} \Rightarrow \tau_1 \xrightarrow{\alpha} \pi_1 \hat{e}' \Rightarrow \tau'_1$$

$$\frac{32b}{\Gamma \vdash \hat{e} \Rightarrow _ \times \tau_2 \xrightarrow{\alpha} \hat{e}' \Rightarrow _ \times \tau'_2} \Gamma \vdash \pi_2 \hat{e} \Rightarrow \tau_2 \xrightarrow{\alpha} \pi_2 \hat{e}' \Rightarrow \tau'_2$$

$$\frac{32c}{\Gamma \vdash \hat{e}_1 \Rightarrow \tau_1 \xrightarrow{\alpha} \hat{e}'_1 \Rightarrow \tau'_1} \Gamma \vdash (\hat{e}_1, e_2) \Rightarrow (\tau_1, \tau_2) \xrightarrow{\alpha} (\hat{e}'_1, e_2) \Rightarrow (\tau'_1, \tau_2)$$

$$\frac{32d}{\Gamma \vdash \hat{e}_2 \Rightarrow \tau_2 \xrightarrow{\alpha} \hat{e}'_2 \Rightarrow \tau'_2} \Gamma \vdash (e_1, \hat{e}_2) \Rightarrow (\tau_1, \tau_2) \xrightarrow{\alpha} (e_1, \hat{e}'_2) \Rightarrow (\tau_1, \tau'_2)$$

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

$$\frac{33a}{\tau \blacktriangleright \times \tau_1 \times _ \quad \Gamma \vdash \hat{e}_1 \xrightarrow{\alpha} \hat{e}'_1 \Leftarrow \tau_1} \Gamma \vdash (\hat{e}_1, e_2) \xrightarrow{\alpha} (\hat{e}'_1, e_2) \Leftarrow \tau$$

$$\frac{33b}{\tau \blacktriangleright \times _ \times \tau_2 \quad \Gamma \vdash \hat{e}_2 \xrightarrow{\alpha} \hat{e}'_2 \Leftarrow \tau_2} \Gamma \vdash (e_1, \hat{e}_2) \xrightarrow{\alpha} (e_1, \hat{e}'_2) \Leftarrow \tau$$

XI. APPENDIX: SYNTHETIC SUGGESTION JUDGEMENT

Our focus in the paper proper has been on the analytic case: suggestions specific to type constraints imposed by the surrounding syntax. In synthetic position, with no type constraints, we would ideally simply defer to the analytic version, using an expected type of \emptyset , which, being consistent with all other types, would simply collect all suggestions from the analytic synthesizers:

$$\frac{\text{SuggestSyn???}}{\frac{\Gamma \vdash \hat{e} \Leftarrow \emptyset \text{ } \emptyset \text{ } A}{\Gamma \vdash \hat{e} \Rightarrow \tau \text{ } \emptyset \text{ } A}}$$

Unfortunately under this rule, the synthetic sensibility theorem would not hold. This is because there are some introduction forms, namely (unannotated) anonymous functions and injections, which according to the Hazelnut semantics are not well-typed in synthetic position. Thus we simply our presentation here by simply returning no suggestion as the synthetic base case:

$$\frac{\text{SuggestSyn}}{\Gamma \vdash \triangleright e \triangleleft \Rightarrow \tau \text{ } \emptyset \text{ } \{\}} \quad \text{SuggestSyn}$$

Alternatively, we could repeat a similar presentation as the analytic base cases, taking care to omit the problematic injection and anonymous function forms. One might ask why we do not dispense with the synthetic suggestion form entirely in our treatment. This is because, as will be seen in the below zipper cases, we must pass through synthetic positions to obtain suggestions for analytic ones. In fact, some of the zipper rules have no analytic form, necessitating a subsumption rule, directly analogous to the action subsumption rule for Hazelnut[24].

$$\frac{\text{Subsumption} \quad \Gamma \vdash \hat{e}^\circ \Rightarrow \tau' \quad \Gamma \vdash \hat{e} \Rightarrow \tau' \text{ } \emptyset \text{ } A \quad \tau \sim \tau'}{\Gamma \vdash \hat{e} \Leftarrow \tau \text{ } \emptyset \text{ } A^*}$$

where $A^* = \{\alpha \in A \mid \text{if } \Gamma \vdash \hat{e} \Rightarrow \tau' \xrightarrow{\alpha} \hat{e}'' \Rightarrow \tau'' \text{ then } \tau \sim \tau''\}$

In any case, while the preceding rules are necessary to complete the technical treatment, they offer no real content relating to architectural assistant decisions and are thus relegated to this appendix.

XII. APPENDIX: PROOF OF SENSIBILITY

We will, by mutual induction, demonstrate the principle conclusions of both analytic and synthetic sensibility, that is:

- For all $\Gamma, \hat{e}, \tau, \alpha \in A$ where $\Gamma \vdash \hat{e}^\circ \Leftarrow \tau$ and $\Gamma \vdash \hat{e} \Leftarrow \tau \text{ } \emptyset \text{ } A$ there exists an \hat{e}' such that $\Gamma \vdash \hat{e} \xrightarrow{\alpha} \hat{e}' \Leftarrow \tau$
- And for all $\Gamma, \hat{e}, \tau, \alpha \in A$ where $\Gamma \vdash \hat{e}^\circ \Rightarrow \tau$ and $\Gamma \vdash \hat{e} \Rightarrow \tau \text{ } \emptyset \text{ } A$ there exists an \hat{e}' and a τ' such that $\Gamma \vdash \hat{e} \Rightarrow \tau \xrightarrow{\alpha} \hat{e}' \Rightarrow \tau'$

We will then provide a proof sketch, mirroring the structure of the main proof, of the secondary conclusions; that is, that in both cases $\#e^\circ \leq \#e'^\circ$.

We will proceed by induction on derivations of the rules defining both analytic and synthetic judgements. We will first case on our 3 analytic base case (cursor term) rules, then our 12 inductive (zipper case) rules, and finally our single synthetic base case and subsumption rule. In each case we will ultimately produce a derivation of an analytic action judgement whose resultant is our desired \hat{e}' (and in the synthetic cases, synthesizes our desired τ'). Where not otherwise noted, numerical references to rules and theorems are to Hazelnut[24].

Let Γ be arbitrary.

A. Analytic Base Cases

1) SuggestCompleteAna:

$$\frac{}{\Gamma \vdash \triangleright \emptyset \triangleleft \Leftarrow \tau \text{ } \emptyset \text{ } \text{Intros}(\tau) \cup \text{Elims}(\Gamma, \tau)}$$

Given an action α synthesized by this rule, we must consider two cases: first, that $\alpha \in \text{Intros}(\tau)$ and second, that $\alpha \in \text{Elims}(\Gamma, \tau)$.

1) Say $\alpha \in \text{Intros}(\tau)$ and proceed by induction on τ . By definition of HType there are 5 cases:

a) Suppose $\tau = 1$. Since $\text{Intros}(1) = \{\swarrow \triangleright() \triangleleft\}$ suffice it to consider $\alpha = \swarrow \triangleright() \triangleleft$. Let D_1 be:

$$\frac{\frac{}{\Gamma \vdash () \Rightarrow 1} \text{29} \quad \frac{}{1 \sim 1} \text{3C}}{\Gamma \vdash () \Leftarrow 1} \text{2B}$$

Thus we have:

$$\frac{D_1 \quad \alpha = \swarrow \triangleright() \triangleleft}{\Gamma \vdash \triangleright \emptyset \triangleleft \xrightarrow{\alpha} \triangleright() \triangleleft \Leftarrow 1} \text{CONSEXP}$$

That is, the analytic action judgement holds for the action of completing a hole with the trivial value, completing this case.

b) Suppose $\tau = \tau_1 \rightarrow \tau_2$. As above, suffice it to consider $\alpha = \swarrow \lambda x. \triangleright \emptyset \triangleleft$. Now let D_1 be

$$\frac{\frac{}{\Gamma, x : \tau_1 \vdash \emptyset \Rightarrow \emptyset} \text{1F} \quad \frac{}{\emptyset \sim \tau_2} \text{3A}}{\Gamma, x : \tau_1 \vdash \emptyset \Leftarrow \tau_2} \text{2B}$$

and let D_2 be

$$\frac{\frac{}{\tau_1 \rightarrow \tau_2 \blacktriangleright \tau_1 \rightarrow \tau_2} \text{4B} \quad D_1}{\Gamma \vdash \lambda x. \emptyset \Leftarrow \tau_2} \text{2A}$$

and so we have that

$$\frac{D_2 \quad \alpha = \swarrow \lambda x. \triangleright \emptyset \triangleleft}{\Gamma \vdash \triangleright \emptyset \triangleleft \xrightarrow{\alpha} \lambda x. \triangleright \emptyset \triangleleft \Leftarrow \tau_1 \rightarrow \tau_2} \text{CONSEXP}$$

c) Suppose $\tau = \tau_1 + \tau_2$. Proceeding as above, consider the case $\alpha = \blacktriangleleft \text{inl}(\blacktriangleright \langle \langle \rangle \rangle \blacktriangleleft)$; $\alpha = \blacktriangleleft \text{inr}(\blacktriangleright \langle \langle \rangle \rangle \blacktriangleleft)$ is precisely analogous. Now let D_1 be

$$\frac{\frac{\Gamma \vdash \langle \rangle \Rightarrow \langle \rangle}{\Gamma \vdash \langle \rangle \Leftarrow \tau_1} \text{1F} \quad \frac{\langle \rangle \sim \tau_1}{\langle \rangle \sim \tau_1} \text{3A}}{\Gamma \vdash \langle \rangle \Leftarrow \tau_1} \text{2B}$$

and let D_2 be

$$\frac{\frac{\tau_1 + \tau_2 \blacktriangleright_+ \tau_1 + \tau_2}{\Gamma \vdash \text{inl}(\langle \rangle) \Leftarrow \tau_1 + \tau_2} \text{20B} \quad D_1}{\Gamma \vdash \text{inl}(\langle \rangle) \Leftarrow \tau_1 + \tau_2} \text{2A}$$

and so we have that

$$\frac{D_2 \quad \alpha = \blacktriangleleft \text{inl}(\blacktriangleright \langle \langle \rangle \rangle \blacktriangleleft)}{\Gamma \vdash \blacktriangleright \langle \langle \rangle \rangle \xrightarrow{\alpha} \text{inl}(\blacktriangleright \langle \langle \rangle \rangle) \Leftarrow \tau_1 + \tau_2} \text{CONSEXP}$$

d) Suppose $\tau = \tau_1 \times \tau_2$. Now let D_1 be

$$\frac{\frac{\Gamma \vdash \langle \rangle \Rightarrow \langle \rangle}{\Gamma \vdash (\langle \rangle, \langle \rangle) \Rightarrow \langle \rangle \times \langle \rangle} \text{1F} \quad \frac{\Gamma \vdash \langle \rangle \Rightarrow \langle \rangle}{\Gamma \vdash (\langle \rangle, \langle \rangle) \Rightarrow \langle \rangle \times \langle \rangle} \text{1F}}{\Gamma \vdash (\langle \rangle, \langle \rangle) \Rightarrow \langle \rangle \times \langle \rangle} \text{30C}$$

and let D_2 be

$$D_1 \quad \frac{\frac{\tau_1 \sim \langle \rangle}{(\tau_1, \tau_2) \sim (\langle \rangle, \langle \rangle)} \text{3C} \quad \frac{\tau_2 \sim \langle \rangle}{(\tau_1, \tau_2) \sim (\langle \rangle, \langle \rangle)} \text{3C}}{(\tau_1, \tau_2) \sim (\langle \rangle, \langle \rangle)} \text{27}}{\Gamma \vdash (\langle \rangle, \langle \rangle) \Leftarrow \tau_1 \times \tau_2} \text{2B}$$

and so we have that

$$\frac{D_2 \quad \alpha = \blacktriangleleft (\blacktriangleright \langle \langle \rangle \rangle, \langle \rangle)}{\Gamma \vdash \blacktriangleright \langle \langle \rangle \rangle \xrightarrow{\alpha} (\blacktriangleright \langle \langle \rangle \rangle, \langle \rangle) \Leftarrow \tau_1 \times \tau_2} \text{CONSEXP}$$

e) Finally, suppose $\tau = \langle \rangle$. By definition, the actions for this case is precisely the collection of all the actions from the previous 4 cases. Thus we are already done.

2) Say instead that $\alpha \in \text{Elims}(\Gamma, \tau)$. Here, let τ be arbitrary. By the definition of Elims we have three cases: Either we have $\alpha \in \text{Case}$, or $\alpha \in \text{Var}(\Gamma, \tau)$, or $\alpha \in \text{AP}(\Gamma, \tau)$.

a) By definition of Case , suffice it to consider $\alpha = \blacktriangleleft \text{case}(\blacktriangleright \langle \langle \rangle \rangle \blacktriangleleft, x. \langle \rangle, y. \langle \rangle)$. Now let $D_{(var)}$ be

$$\frac{\frac{\Gamma, var : \langle \rangle \vdash \langle \rangle \Rightarrow \langle \rangle}{\Gamma, var : \langle \rangle \vdash \langle \rangle \Leftarrow \tau} \text{1F} \quad \frac{\langle \rangle \sim \tau}{\langle \rangle \sim \tau} \text{3A}}{\Gamma, var : \langle \rangle \vdash \langle \rangle \Leftarrow \tau} \text{2B}$$

and let D_1 be

$$\frac{\frac{\Gamma \vdash \langle \rangle \Rightarrow \langle \rangle}{\Gamma \vdash \text{case}(\langle \rangle, x. \langle \rangle, y. \langle \rangle) \Leftarrow \tau} \text{1F} \quad \frac{\langle \rangle \blacktriangleright_+ \langle \rangle + \langle \rangle}{\Gamma \vdash \text{case}(\langle \rangle, x. \langle \rangle, y. \langle \rangle) \Leftarrow \tau} \text{20A} \quad D_{(x)} \quad D_{(y)}}{\Gamma \vdash \text{case}(\langle \rangle, x. \langle \rangle, y. \langle \rangle) \Leftarrow \tau} \text{21B}$$

and so we have that

$$\frac{D_1 \quad \alpha = \blacktriangleleft \text{case}(\blacktriangleright \langle \langle \rangle \rangle \blacktriangleleft, x. \langle \rangle, y. \langle \rangle)}{\Gamma \vdash \blacktriangleright \langle \langle \rangle \rangle \xrightarrow{\alpha} \text{case}(\blacktriangleright \langle \langle \rangle \rangle, x. \langle \rangle, y. \langle \rangle) \Leftarrow \tau} \text{CONSEXP}$$

b) By definition of Var , suffice it to consider $\alpha = \blacktriangleleft \blacktriangleright x \blacktriangleleft$ for some $x : \tau' \in \Gamma$ where $\tau' \sim \tau$. Now let D_1 be

$$\frac{\frac{x : \tau' \in \Gamma}{\Gamma \vdash x \Rightarrow \tau'} \text{1A} \quad \frac{\tau' \sim \tau}{\tau' \sim \tau} \text{3C}}{\Gamma \vdash x \Leftarrow \tau} \text{2B}$$

so that

$$\frac{D_1 \quad \alpha = \blacktriangleleft \blacktriangleright x \blacktriangleleft}{\Gamma \vdash \blacktriangleright \langle \rangle \blacktriangleleft \xrightarrow{\alpha} \blacktriangleright x \blacktriangleleft \Leftarrow \tau} \text{CONSEXP}$$

c) By definition of AP , suffice it to consider $\alpha = \blacktriangleleft \blacktriangleright e \blacktriangleleft$ for some e satisfying $\Gamma \vdash x \rightsquigarrow e \Leftarrow \tau$ where $x : \tau' \in \Gamma$ and $\tau' \sim \tau$. Now we state and prove a lemma:

Lemma: For all Γ, e, e', τ where $\Gamma \vdash e' \Rightarrow _$ and $\Gamma \vdash e' \rightsquigarrow e \Leftarrow \tau$, we have that $\Gamma \vdash e \Leftarrow \tau$.

Proof: Let Γ, e, e', τ be arbitrary; we proceed by induction on derivations of the AP judgement. By definition of this judgement there is one base and three inductive cases to consider:

- **AP – Base:** In this case, the premise of the rule gives us our conclusion precisely:

$$\frac{\text{AP-Base} \quad \Gamma \vdash e \Leftarrow \tau}{\Gamma \vdash e \rightsquigarrow e \Leftarrow \tau}$$

- **AP – App:** In this case, we may apply the inductive hypothesis directly to the second premise, which immediately gives us our conclusion:

$$\frac{\text{AP-App} \quad \Gamma \vdash e \Rightarrow _ \rightarrow _ \quad \Gamma \vdash e(\langle \rangle) \rightsquigarrow e' \Leftarrow \tau}{\Gamma \vdash e \rightsquigarrow e' \Leftarrow \tau}$$

- **AP – Proj1 and AP – Proj2:** These proceed directly analogously to the previous case.

Note that while this demonstrates that expression satisfying the judgement are indeed properly typed, this judgement (unlike the rest in this paper) is neither deterministic or computational; in practice a search strategy would need to be specified, and be shown to be terminating.

So using the lemma and letting Γ be our Γ and $e' = x$ which immediately satisfies $\Gamma \vdash x \rightsquigarrow _$ we have that $\Gamma \vdash e \Leftarrow \tau$ and thus

$$\frac{\frac{\Gamma \vdash e \Leftarrow \tau}{\Gamma \vdash \blacktriangleright \langle \rangle \blacktriangleleft \xrightarrow{\alpha} \blacktriangleright e \blacktriangleleft \Leftarrow \tau} \text{Ass.} \quad \alpha = \blacktriangleleft \blacktriangleright e \blacktriangleleft}{\Gamma \vdash \blacktriangleright \langle \rangle \blacktriangleleft \xrightarrow{\alpha} \blacktriangleright e \blacktriangleleft \Leftarrow \tau} \text{CONSEXP}$$

2) *SuggestModifyAna:*

$$\frac{\text{SuggestModifyAna} \quad e \neq \langle \rangle \quad e \neq (_)}{\Gamma \vdash \blacktriangleright e \blacktriangleleft \Leftarrow \tau \text{ } \wp \text{ Wraps}(\Gamma, e, \tau) \cup \text{Replaces}(\Gamma, \tau)}$$

Let τ be arbitrary. Given an action α satisfying this judgement, we have two cases: Either $\alpha \in \text{Wraps}(\Gamma, e, \tau)$ or $\alpha \in \text{Replaces}(\Gamma, \tau)$.

- 1) Suppose $\alpha \in \text{Wraps}(\Gamma, e, \tau)$. Then by definition of **Wraps**, $\alpha = \langle f(\triangleright e \triangleleft) \rangle$ for some $f : \tau_{in} \rightarrow \tau \in \Gamma$ where $\Gamma \vdash e \Rightarrow \tau_e$ and $\tau_e \sim \tau_{in}$. Now let D_1 be

$$\frac{\frac{f : \tau_{in} \rightarrow \tau \in \Gamma}{\Gamma \vdash f \Rightarrow \tau_{in} \rightarrow \tau} \text{1A} \quad \frac{\frac{\frac{\text{Ass.}}{\Gamma \vdash e \Rightarrow \tau_e} \quad \frac{\text{Ass.}}{\tau_e \sim \tau_{in}}}{\Gamma \vdash e \Leftarrow \tau_{in}} \text{2B}}{\Gamma \vdash f(e) \Leftarrow \tau} \text{1B}}$$

so we have that

$$\frac{D_1 \quad \alpha = \langle f(\triangleright e \triangleleft) \rangle}{\Gamma \vdash \langle \triangleright \langle \langle \langle \triangleright e \triangleleft \rangle \rangle \rangle \Leftarrow \tau} \text{CONSEXP}$$

- 2) Suppose instead that $\alpha \in \text{Replaces}(\Gamma, \tau)$. Then by definition of **Replaces**, $\alpha = \langle \text{del}; \alpha' \rangle$ for some $\alpha' \in A'$ from $\Gamma \vdash \langle \triangleright \langle \langle \langle \triangleright \langle \langle \triangleright \rangle \rangle \rangle \rangle \Leftarrow \tau \circlearrowleft A' \rangle$. Let \hat{e}' be the result of performing α' on $\langle \triangleright \langle \langle \langle \triangleright \rangle \rangle \rangle$ so that by the **SuggestCompleteAna** case we know $\Gamma \vdash \langle \triangleright \langle \langle \langle \triangleright \rangle \rangle \rangle \xrightarrow{\alpha'} \hat{e}' \Leftarrow \tau$. Then we have:

$$\frac{\frac{\frac{\frac{\frac{\Gamma \vdash \langle \triangleright e \triangleleft \rangle \xrightarrow{\langle \text{del} \rangle} \langle \triangleright \langle \langle \langle \triangleright \rangle \rangle \rangle \rangle \Leftarrow \tau}{\text{Ass.}}}{\Gamma \vdash \langle \triangleright \langle \langle \langle \triangleright \rangle \rangle \rangle \xrightarrow{\alpha'} \hat{e}' \Leftarrow \tau} \text{11B}}{\Gamma \vdash \langle \triangleright e \triangleleft \rangle \xrightarrow{\langle \text{del}; \alpha' \rangle} \hat{e}' \Leftarrow \tau} \text{15B}}$$

- 3) *SuggestRepairAna*:

$$\frac{\text{SuggestRepairAna} \quad \Gamma \vdash \langle \triangleright e \triangleleft \rangle \Leftarrow \tau \circlearrowleft A}{\Gamma \vdash \langle \triangleright \langle \langle \langle \langle \triangleright e \triangleleft \rangle \rangle \rangle \rangle \Leftarrow \tau \circlearrowleft \{\text{move child}; \alpha \mid \alpha \in A\}}$$

Let τ be arbitrary. Then $\alpha = \text{move child}; \alpha$ for some $\alpha \in A$ satisfying $\Gamma \vdash \langle \triangleright e \triangleleft \rangle \Leftarrow \tau \circlearrowleft A$. Let \hat{e}' be the result of performing α on $\langle \triangleright e \triangleleft \rangle$ so that by the **SuggestModifyAna** case we know $\Gamma \vdash \langle \triangleright \langle \langle \langle \triangleright \rangle \rangle \rangle \xrightarrow{\alpha'} \hat{e}' \Leftarrow \tau$. Then we have:

$$\frac{\frac{\frac{\frac{\Gamma \vdash \langle \triangleright \langle \langle \langle \langle \triangleright \langle \langle \langle \triangleright \rangle \rangle \rangle \rangle \rangle \rangle \xrightarrow{\text{move child}} \langle \triangleright \langle \langle \langle \triangleright \rangle \rangle \rangle \rangle \Leftarrow \tau}{\text{Ass.}}}{\Gamma \vdash \langle \triangleright e \triangleleft \rangle \xrightarrow{\alpha} \hat{e}' \Leftarrow \tau} \text{11B}}{\Gamma \vdash \langle \triangleright \langle \langle \langle \langle \triangleright \rangle \rangle \rangle \rangle \xrightarrow{\text{move child}; \alpha} \hat{e}' \Leftarrow \tau} \text{15B}}$$

B. Zipper Cases

- 1) *Rule: SZLam*:

$$\frac{\text{SZLam} \quad \tau \blacktriangleright \rightarrow \tau_1 \rightarrow \tau_2 \quad \Gamma, x : \tau_1 \vdash \hat{e} \Leftarrow \tau_2 \circlearrowleft A}{\Gamma \vdash \lambda x. \hat{e} \Leftarrow \tau \circlearrowleft A}$$

As rule's premise we know $\Gamma, x : \tau_1 \vdash \hat{e} \Leftarrow \tau_2 \circlearrowleft A$. Let $\alpha \in A$ satisfying that premise. Then by our induction

hypothesis we have that there exists an \hat{e}' satisfying $\Gamma \vdash \hat{e} \xrightarrow{\alpha} \hat{e}' \Leftarrow \tau$. Thus

$$\frac{\frac{\text{Ass.}}{\tau \blacktriangleright \rightarrow \tau_1 \rightarrow \tau_2} \quad \frac{\text{Ass.}}{\Gamma \vdash \hat{e} \xrightarrow{\alpha} \hat{e}' \Leftarrow \tau}}{\Gamma \vdash \lambda x. \hat{e} \xrightarrow{\alpha} \lambda x. \hat{e}' \Leftarrow \tau} \text{18A}$$

- 2) *Rule: SZAppR*:

$$\frac{\text{SZAppR} \quad \Gamma \vdash e_1 \Rightarrow \tau_1 \quad \tau_1 \blacktriangleright \rightarrow \tau_2 \rightarrow \tau \quad \Gamma \vdash \hat{e}_2 \Leftarrow \tau_2 \circlearrowleft A}{\Gamma \vdash e_1(\hat{e}_2) \Rightarrow \tau \circlearrowleft A}$$

As the rule's premise, we know $\Gamma \vdash \hat{e}_2 \Leftarrow \tau_2 \circlearrowleft A$. Let $\alpha \in A$ satisfying that premise. Then by our induction hypothesis we have that there exists an \hat{e}' satisfying $\Gamma \vdash \hat{e}_2 \xrightarrow{\alpha} \hat{e}' \Leftarrow \tau$. Thus

$$\frac{\frac{\frac{\frac{\text{Ass.}}{\Gamma \vdash e_1 \Rightarrow \tau_1} \quad \frac{\text{Ass.}}{\Gamma \vdash \hat{e}_2 \xrightarrow{\alpha} \hat{e}' \Leftarrow \tau_2}}{\Gamma \vdash e_1(\hat{e}_2) \Rightarrow \tau \xrightarrow{\alpha} e_1(\hat{e}') \Rightarrow \tau} \text{18C}}{\Gamma \vdash e_1(\hat{e}_2) \Rightarrow \tau \xrightarrow{\alpha} e_1(\hat{e}') \Rightarrow \tau}}$$

- 3) *Rule: SZAppL*:

$$\frac{\Gamma \vdash \hat{e}_1 \Rightarrow \tau_1 \circlearrowleft A \quad \tau_1 \blacktriangleright \rightarrow \tau_2 \rightarrow \tau \quad \Gamma \vdash e_2 \Leftarrow \tau_2}{\Gamma \vdash \hat{e}_1(e_2) \Rightarrow \tau \circlearrowleft A^*}$$

where $A^* = \{\alpha \in A \mid \text{if } \Gamma \vdash \hat{e}_1 \Rightarrow \tau_1 \xrightarrow{\alpha} \hat{e}'_1 \Rightarrow \tau'_1 \text{ then } \tau'_1 \sim (\tau_2 \rightarrow \langle \rangle)\}$

As the rule's premise, we know $\Gamma \vdash \hat{e}_1 \Rightarrow \tau_1 \circlearrowleft A$. Let $\alpha \in A$ satisfying that premise. Then by our induction hypothesis we have that there exists an \hat{e}'_1 and a τ'_1 satisfying $\Gamma \vdash \hat{e}_1 \Rightarrow \tau_1 \xrightarrow{\alpha} \hat{e}'_1 \Rightarrow \tau'_1$.

Furthermore, since we know from the condition on $\alpha \in A^*$ that τ'_1 must have either function type with argument consistent with τ_2 (being the matched-arrow argument type of τ_1), or hole type, we know that $\tau'_1 \blacktriangleright \rightarrow \tau'_2 \rightarrow \tau'$ holds and establishes that $\tau_2 \sim \tau'_2$ and thus, along with the premise $\Gamma \vdash e_2 \Leftarrow \tau_2$, establishes that $\Gamma \vdash e_2 \Leftarrow \tau'_2$.

Finally, from the typing-synthesis premise of the theorem and inversion we have $\Gamma \vdash e_1 \Rightarrow \tau_1$ and hence $\Gamma \vdash \hat{e}_1^\circ \Rightarrow \tau_1$. Thus

$$\frac{\frac{\frac{\frac{\text{Ass.}}{\Gamma \vdash \hat{e}_1^\circ \Rightarrow \tau_1} \quad \frac{\text{Ass.}}{\tau'_1 \blacktriangleright \rightarrow \tau'_2 \rightarrow \tau'}}{\Gamma \vdash \hat{e}_1 \Rightarrow \tau_1 \xrightarrow{\alpha} \hat{e}'_1 \Rightarrow \tau'_1} \text{Ass.}}{\Gamma \vdash e_2 \Leftarrow \tau'_2} \text{Ass.}}{\Gamma \vdash e_1(\hat{e}_2) \Rightarrow \tau \xrightarrow{\alpha} e_1(\hat{e}'_2) \Rightarrow \tau'} \text{18D}}$$

- 4) *Rule: SZInL (SZInR similar)*:

$$\frac{\tau \blacktriangleright + \tau_1 + \tau_2 \quad \Gamma \vdash \hat{e} \Leftarrow \tau_1 \circlearrowleft A}{\Gamma \vdash \text{inl}(\hat{e}) \Leftarrow \tau \circlearrowleft A}$$

As the rule's premise we know $\Gamma \vdash \hat{e} \Leftarrow \tau_1 \circlearrowleft A$. Let $\alpha \in A$ satisfying that premise. Then by our induction

hypothesis we have that there exists an \hat{e}' satisfying $\Gamma \vdash \hat{e} \xrightarrow{\alpha} \hat{e}' \Leftarrow \tau_1$. Thus

$$\frac{\frac{}{\tau \blacktriangleright_+ \tau_1 + \tau_2} \text{Ass.} \quad \frac{}{\Gamma \vdash \hat{e} \xrightarrow{\alpha} \hat{e}' \Leftarrow \tau} \text{Ass.}}{\Gamma \vdash \text{inl}(\hat{e}) \xrightarrow{\alpha} \text{inl}(\hat{e}') \Leftarrow \tau} \text{23D}$$

5) Rule: *SZCaseR* (*SZCaseL* similar):

$$\frac{\Gamma \vdash e \Rightarrow \tau_+ \quad \tau_+ \blacktriangleright_+ \tau_L + \tau_R \quad \Gamma, y : \tau_R \vdash \hat{e}_R \Leftarrow \tau \wp A}{\Gamma \vdash \text{case}(e, x.e_L, y.\hat{e}_R) \Leftarrow \tau \wp A}$$

As the rule's premise, we know $\Gamma, y : \tau_R \vdash \hat{e}_R \Leftarrow \tau \wp A$. Let $\alpha \in A$ satisfying that premise. Then by our induction hypothesis we have that there exists an \hat{e}'_R satisfying $\Gamma \vdash \hat{e}_R \xrightarrow{\alpha} \hat{e}'_R \Leftarrow \tau_R$. Thus

$$\frac{\frac{}{\Gamma \vdash e \Rightarrow \tau_+} \text{Ass.} \quad \frac{}{\Gamma \vdash \hat{e}_R \xrightarrow{\alpha} \hat{e}'_R \Leftarrow \tau_R} \text{Ass.}}{\tau_+ \blacktriangleright_+ \tau_L \rightarrow \tau_R \quad \Gamma \vdash \text{case}(e, x.e_L, y.\hat{e}_R) \xrightarrow{\alpha} \text{case}(e, x.e_L, y.\hat{e}'_R) \Leftarrow \tau} \text{23G}$$

6) Rule: *SZCase0*:

$$\frac{\Gamma \vdash \hat{e} \Rightarrow \tau_+ \wp A \quad \tau_+ \blacktriangleright_+ \tau_L + \tau_R \quad \Gamma, x : \tau_L \vdash e_L \Leftarrow \tau \quad \Gamma, y : \tau_R \vdash e_R \Leftarrow \tau}{\Gamma \vdash \text{case}(\hat{e}, x.e_L, y.e_R) \Leftarrow \tau \wp A^*}$$

where $A^* = \{\alpha \in A \mid \text{if } \Gamma \vdash \hat{e} \Rightarrow \tau_+ \xrightarrow{\alpha} \hat{e}' \Rightarrow \tau' \text{ then } \tau' \sim (\tau_L + \tau_R)\}$

As the rule's premise, we know $\Gamma \vdash \hat{e} \Rightarrow \tau_+ \wp A$. Let $\alpha \in A$ satisfying that premise. Then by our induction hypothesis we have that there exists an \hat{e}' and a τ'_+ satisfying $\Gamma \vdash \hat{e}_1 \Rightarrow \tau_+ \xrightarrow{\alpha} \hat{e}' \Rightarrow \tau'_+$.

Furthermore, since we know from the condition on $\alpha \in A^*$ that τ'_+ must have either sum type consistent with $\tau_L + \tau_R$ (being the matched-sum of τ_+), or hole type, we know that $\tau'_+ \blacktriangleright_+ \tau'_L \rightarrow \tau'_R$ holds and establishes $\tau_L \sim \tau'_L$ and $\tau_R \sim \tau'_R$ and thus, along with the last two premises of *SZCase0*, that $\Gamma, x : \tau'_L \vdash e_L \Leftarrow \tau$ and $\Gamma, y : \tau'_R \vdash e_R \Leftarrow \tau$.

Finally from the typing-analysis premise of the theorem and inversion we have $\Gamma \vdash e \Rightarrow \tau_+$ and hence $\Gamma \vdash \hat{e}^\circ \Rightarrow \tau_+$. Thus

$$\frac{\frac{}{\Gamma \vdash \hat{e}^\circ \Rightarrow \tau_+} \text{Ass.} \quad \frac{}{\Gamma \vdash \hat{e} \Rightarrow \tau_+ \xrightarrow{\alpha} \hat{e}' \Rightarrow \tau'_+} \text{Ass.} \quad \frac{}{\tau'_+ \blacktriangleright_+ \tau'_L \rightarrow \tau'_R} \text{Ass.}}{\frac{\frac{}{\Gamma, x : \tau'_L \vdash e_L \Leftarrow \tau} \text{Ass.} \quad \frac{}{\Gamma, y : \tau'_R \vdash e_R \Leftarrow \tau} \text{Ass.}}{\Gamma \vdash \text{case}(\hat{e}, x.e_L, y.e_R) \xrightarrow{\alpha} \text{case}(\hat{e}', x.e_L, y.e_R) \Leftarrow \tau} \text{23E}}$$

7) Rule: *SZPrj1* (*SZPrj2* similar) :

$$\frac{\Gamma \vdash \hat{e} \Rightarrow \tau_1 \times \tau_2 \wp A}{\Gamma \vdash \pi_1 \hat{e} \Rightarrow \tau_1 \wp A}$$

As the rule's premise, we know $\Gamma \vdash \hat{e} \Rightarrow \tau_1 \times \tau_2 \wp A$. Let $\alpha \in A$ satisfying that premise. Then by our (synthetic) induction hypothesis we have that there exists an \hat{e}' and a τ'_1 satisfying $\Gamma \vdash \hat{e} \Rightarrow \tau_1 \times \tau_2 \xrightarrow{\alpha} \hat{e}' \Rightarrow \tau'_1 \times \tau_2$. Thus we satisfy

$$\frac{\Gamma \vdash \hat{e} \Rightarrow \tau_1 \times _ \xrightarrow{\alpha} \hat{e}' \Rightarrow \tau'_1 \times _}{\Gamma \vdash \pi_1 \hat{e} \Rightarrow \tau_1 \xrightarrow{\alpha} \pi_1 \hat{e}' \Rightarrow \tau'_1} \text{32A}$$

8) Rule: *SZSynPair1* (*SZSynPair2* similar) :

$$\frac{\Gamma \vdash \hat{e}_1 \Rightarrow \tau_1 \wp A}{\Gamma \vdash (\hat{e}_1, e_2) \Rightarrow \tau_1 \times \tau_2 \wp A}$$

As the rule's premise, we know $\Gamma \vdash \hat{e}_1 \Rightarrow \tau_1 \wp A$. Let $\alpha \in A$ satisfying that premise. Then by our (synthetic) induction hypothesis we have that there exists an \hat{e}'_1 and a τ'_1 satisfying $\Gamma \vdash \hat{e}_1 \Rightarrow \tau_1 \xrightarrow{\alpha} \hat{e}'_1 \Rightarrow \tau'_1$. Thus we satisfy

$$\frac{\Gamma \vdash \hat{e}_1 \Rightarrow \tau_1 \xrightarrow{\alpha} \hat{e}'_1 \Rightarrow \tau'_1}{\Gamma \vdash (\hat{e}_1, e_2) \Rightarrow (\tau_1, \tau_2) \xrightarrow{\alpha} (\hat{e}'_1, e_2) \Rightarrow (\tau'_1, \tau_2)} \text{32C}$$

9) Rule: *SZAnaPair1* (*SZAnaPair2* similar) :

$$\frac{\tau \blacktriangleright_\times \tau_1 \times \tau_2 \quad \Gamma \vdash \hat{e}_2 \Leftarrow \tau_1 \wp A}{\Gamma \vdash (e_1, \hat{e}_2) \Leftarrow \tau \wp A}$$

As the rule's premise, we know $\Gamma \vdash \hat{e}_2 \Leftarrow \tau_1 \wp A$ satisfying that premise. Then by our (analytic) induction hypothesis we have that there exists an \hat{e}'_1 satisfying $\Gamma \vdash \hat{e}_1 \xrightarrow{\alpha} \hat{e}'_1 \Leftarrow \tau_1$. Thus along with the rule's other premise we satisfy

$$\frac{\tau \blacktriangleright_\times \tau_1 \times _ \quad \Gamma \vdash \hat{e}_1 \xrightarrow{\alpha} \hat{e}'_1 \Leftarrow \tau_1}{\Gamma \vdash (\hat{e}_1, e_2) \xrightarrow{\alpha} (\hat{e}'_1, e_2) \Leftarrow \tau} \text{33A}$$

C. *Synthesis & Subsumption*

1) Rule: *SuggestSyn*:

$$\frac{}{\Gamma \vdash \triangleright e \Leftarrow \tau \wp \{ \}}$$

This rule suggests no actions, so the conclusion holds vacuously.

2) Rule: *Subsumption*:

$$\frac{\text{Subsumption} \quad \Gamma \vdash \hat{e}^\circ \Rightarrow \tau' \quad \Gamma \vdash \hat{e} \Rightarrow \tau' \wp A \quad \tau \sim \tau'}{\Gamma \vdash \hat{e} \Leftarrow \tau \wp A^*}$$

where $A^* = \{\alpha \in A \mid \text{if } \Gamma \vdash \hat{e} \Rightarrow \tau' \xrightarrow{\alpha} \hat{e}' \Rightarrow \tau'' \text{ then } \tau \sim \tau''\}$

As rule's premise we know $\Gamma \vdash \hat{e} \Rightarrow \tau' \wp A$. Let $\alpha \in A^*$. Then by our synthetic induction hypothesis we have that there exists an \hat{e}' and a τ'' satisfying $\Gamma \vdash \hat{e} \Rightarrow \tau' \xrightarrow{\alpha} \hat{e}' \Rightarrow \tau''$

τ'' . Finally, from our condition on $\alpha \in A^*$, we have that $\tau \sim \tau''$. Thus

$$\frac{\frac{\overline{\Gamma \vdash \hat{e}^\circ \Leftarrow \tau'} \text{Ass.}}{\Gamma \vdash \hat{e} \Rightarrow \tau' \xrightarrow{\alpha} \hat{e}' \Rightarrow \tau''} \text{Ass.} \quad \frac{}{\tau \sim \tau''} \text{Ass.}}{\Gamma \vdash \hat{e} \xrightarrow{\alpha} \hat{e}' \Leftarrow \tau} 5$$

D. Error Count Sketch

We sketch a proof for the second clause of the suggestion sensibility theorem; namely that the non-empty-hole count (reflecting the number of what would otherwise be type errors) does not increase.

Note that from inspection of the analytic base cases, only three types of actions are ultimately performed: delete, move child, and construct expression. Movement leaves the non-empty-hole count unchanged, and deletion either leaves it unchanged or reduces it. Construction will increase non-empty-hole count by precisely the number of non-empty holes in the constructed expression. By inspection, none of the expressions constructed in the above cases contain non-empty holes, so the non-empty-hole count can never increase.

(Note that the proof of the first clause of sensibility established that all of our constructions remain statically meaningful: that is, they ensure we did not introduce any type errors unaccounted for by explicit non-empty-holes)