Polymorphism with Typed Holes

Adam Chen[1(✉)], Thomas Porter[2], and Cyrus Omar[2]

[1] Stevens Institute of Technology, Hoboken, USA
achen19@stevens.edu
[2] University of Michigan, Ann Arbor, USA
{thomasjp,comar}@umich.edu

Abstract. Live programming environments aim to provide rapid and continuous feedback to developers, but this can be challenging when a program is incomplete. Hazel is a live programming environment that aims to solve this problem by using expression and type holes to stand for missing terms or mark erroneous terms. Hazel is based on the Hazelnut Live calculus presented in prior work.

This paper starts by presenting Polymorphic Hazelnut Live, an extension of Hazelnut Live to support explicit System F-style polymorphism. We show, with mechanized proofs in Agda, that this extended system satisfies the key metatheoretic properties necessary for live programming with typed holes. We compare the type system of Polymorphic Hazelnut Live to other systems that combine gradual typing (i.e. the theory of type holes) with polymorphism, discussing subtleties related to parametricity and the gradual guarantee.

Finally, we present a method to integrate a form of implicit type application into the Hazel architecture. We propose a system in which the programmer may omit explicit type applications, and the editor (rather than downstream tools like a typechecker or compiler) implicitly inserts and fills them, allowing the user to see and override these implicit type applications as needed.

1 Introduction

Live programming environments seek to provide programmers with continuous feedback by analyzing and evaluating programs as they are being edited [24]. Some common examples of live programming environments are Jupyter Notebooks [11] (which integrate with several languages such as Julia, Python, and R), spreadsheets [26], and editor-integrated debuggers [13]. However, one challenge to the live model is that in most languages, incomplete programs do not have formal structure or meaning. Many IDEs therefore either exhibit gaps in liveness or rely on heuristic error recovery methods to reason about incomplete programs. This creates many situations where a programmer may receive incomplete or incorrect information about their code as they are in the process of editing it. In these cases, they must finish their edit then wait for the tool's analysis to update.

J. Hemann and S. Chang (Eds.): TFP 2024, LNCS 14843, pp. 134–159, 2025.
https://doi.org/10.1007/978-3-031-74558-4_7

The Hazel programming language and environment seeks to solve this problem of reasoning about incomplete programs by defining a formal semantics for expressions that can contain *holes* in types, expressions, and patterns (collectively, terms). Empty holes stand for missing terms, while non-empty holes serve as marked membranes around erroneous terms. The Hazel editor inserts these holes automatically [15], and Hazel is unique in assigning rich static and dynamic meaning to every editor state [19].

Hazel is based on the Hazelnut Live calculus, which defines a static and dynamic semantics for programs with expression and type holes [18]. Hazelnut Live combines and extends ideas from contextual modal type theory [16] and the gradually typed lambda calculus [21]. Section 2.1 provides more background on the details of Hazel and its calculi.

The problem that motivates this work is that Hazelnut Live did not consider abstraction over types, which is key for practical typed functional programming. Our contributions in this paper are to:

- extend the theory of Hazelnut Live to include explicit polymorphism,
- extend the existing Hazel implementation to support polymorphic programming with typed holes,
- define a novel weakening of parametricity that is shown to hold of the system,
- prove the static gradual guarantee for the system, and
- propose an editor service that combines some of the strengths of explicit and implicit polymorphism.

Section 3 presents *Polymorphic Hazelnut Live*, a polymorphic extension of Hazelnut Live. Section 4 establishes that Polymorphic Hazelnut Live retains the key metatheoretic properties of Hazelnut Live, suitably modified to account for type variables, including type safety (in the presence of holes), and discusses the important metatheoretic properties that were not previously considered for Hazel but that have been studied in the literature, namely parametricity [9] and the gradual guarantee [22]. Type holes are known to weaken parametricity. We review this active research area and discuss a weakening of parametricity that holds of our system. Section 5 describes our mechanization of these metatheoretic properties in Agda and the implementation of polymorphism in the Hazel programming environment. In practice, it is cumbersome to explicitly apply type abstractions and most major functional languages support implicitly inferred type applications. Section 6 approaches the problem of implicit type application in a unique way – by outlining an in-progress edit-time implicit type application system for Hazel, whereby users can see and intervene in the implicit application when desired rather than relying on invisible implicit type application logic.

2 Background

2.1 The Hazel Programming Environment

Hazel is a live functional programming environment that provides gapless editor support; that is, all editor states, including those corresponding to incomplete

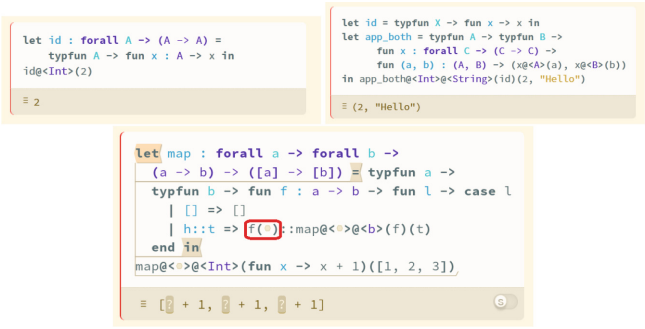

Fig. 1. Screenshots of the current Hazel UI showing off polymorphic functions, including the polymorphic identity, a rank-2 polymorphic function, and a polymorphic map featuring both type and expression holes (expression hole is the argument of the outlined application).

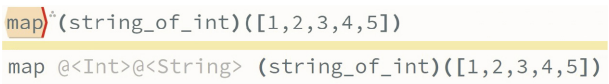

Fig. 2. Mockup of a hypothetical Hazel UI showing a folded (above) and unfolded (below) automatically inserted type application. Such an editor service fits into the existing Hazel architecture and would allow polymorphic functions to be used as if they were implicit.

programs, are endowed with static and dynamic meaning. Figure 1 displays some examples of polymorphic code in the Hazel editor and shows typed expression holes within polymorphic code. Figure 2 displays a hypothetical Hazel interface for a editor service that would insert type applications automatically, and would recover some benefits of implicit polymorphism. A reader who is more interested in the user-centered aspect of Hazel and less in its formal theory may wish to jump directly to the discussion of this feature, in Sect. 6.

2.2 The Theory of Hazel

The theory of Hazel is organized into several calculi, somewhat analogous to compiler phases. The Hazel grammar is shown in Fig. 3, including types and three different kinds of expressions. b stands for any base types of the language, and c for any constants. We have simplified the presentation slightly by removing

$$
\begin{aligned}
\text{Type } \tau &::= b \mid \tau \to \tau \mid ? \\
\text{HExp } e &::= c \mid x \mid \lambda x : \tau.\, e \mid \lambda x.\, e \mid e\ e \mid (\!|\,|\!) \\
\text{IHExp } d &::= c \mid x \mid \lambda x : \tau.\, d \mid \lambda x.\, d \mid d\ d \mid (\!|\,|\!) \mid (\!|d|\!) \mid d\langle \tau \Rightarrow \tau \rangle \mid d\langle \tau \Rightarrow ? \nRightarrow \tau \rangle
\end{aligned}
$$

Fig. 3. Simplified Hazel grammar.

information carried by the holes in the different calculi and other constructs not relevant to this work (such as abstract data types, etc.).

To illustrate the main components of the theory of Hazel, we provide an example rule for dealing with function applications at each major stage of Hazel in Fig. 4.

Parsing with Holes. Hazel users enter code using an editor that inserts holes automatically to ensure that every editor state contains a well-formed expression with holes, e. The Hazelnut calculus [19] defined an edit action calculus that inserts these holes, which has since evolved to a support more natural keyboard-driven input [15].

In Hazelnut, the resulting expression is guaranteed to be both syntactically and statically valid. However, in the newer editor calculus with flexible keyboard-driven input, the resulting expression is only guaranteed to be syntactically valid, and may still have static errors. To address this, Zhao et al. [28] describe the *marked lambda calculus*, a system that restores static correctness by inserting marks, or nonempty holes, into syntactically valid programs. We further discuss this procedure in Sect. 6, but do not include it in our presentation.

Elaboration. After marking, the program is syntactically and statically valid. *Hazelnut Live* is a system for endowing Hazel with dynamic semantics. It operates by bidirectionally elaborating external (hole) expressions (HExp) into internal (holes) expressions (IHExp). We will identify the external expressions of Hazelnut Live with the marked expressions of the marked lambda calculus. In Hazelnut Live, all holes also carry a unique name, and internal holes carry a finite substitution of terms in for variables.

The elaboration judgment forms are $\Gamma \vdash e \Rightarrow \tau \rightsquigarrow d \dashv \Delta$ and $\Gamma \vdash e \Leftarrow \tau_1 \rightsquigarrow d : \tau_2 \dashv \Delta$. Note that in the analytic case a type τ_2 is output as well, which must be consistent with the input τ_1. During elaboration, casts between types are inserted to enable dynamic type checks that are necessary due to missing type information, i.e. type holes, in the original program. Elaboration also produces a hole context Δ, which maps the program's empty and nonempty holes to their contexts and expected types, and associates each hole occurrence with a sequence of substitutions that has been applied to it. This facilitates the *fill-and-resume* operation, by which a reduced program with holes can be refined (by filling an empty hole with an expression) and reduction can be resumed.

These internal expressions d are typed via a type assignment judgment of form $\Delta; \Gamma \vdash d : \tau$. This type assignment judgment is not bidirectional, unlike for external expressions. The hole context is an input to this judgment, as it assigns types to holes. Figure 4 contains the elaboration and internal typing rules for applications, ESAp and TALam.

Evaluation. Internal expressions are evaluated via a stepping relation of form $d \mapsto d'$, and the resulting normal form, if it exists, is reported back to the end user. The stepping relation is defined via contextual semantics, meaning redexes and evaluation contexts are defined (not shown here), with a transition

$$\textsc{SAp}\quad \frac{\Gamma \vdash e_1 \Rightarrow \tau \qquad \tau \blacktriangleright_{\to} \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 \Leftarrow \tau_1}{\Gamma \vdash e_1\ e_2 \Rightarrow \tau_2}$$

$$\textsc{ESAp}\quad \frac{\tau \blacktriangleright_{\to} \tau_1 \to \tau_2 \qquad \Gamma \vdash e_1 \Leftarrow \tau_2 \to \tau \rightsquigarrow d_1 : \tau_1' \dashv \Delta_1 \qquad \Gamma \vdash e_2 \Rightarrow \tau_2 \rightsquigarrow d_2 \dashv \tau_2' \Delta_2}{\Gamma \vdash e_1\ e_2 \Rightarrow \tau \rightsquigarrow (d_1\langle \tau_1' \Rightarrow \tau_2 \to \tau \rangle)\ (d_2\langle \tau_2' \Rightarrow \tau_2 \rangle) \dashv \Delta_1 \cup \Delta_2}$$

$$\textsc{TALam}\quad \frac{\Delta; \Gamma \vdash d_1 : \tau_2 \to \tau \qquad \Delta; \Gamma \vdash d_2 : \tau_2}{\Delta; \Gamma \vdash d_1\ d_2 : \tau}$$

$$\textsc{ITLam}\quad \frac{[d_2\ \mathsf{final}]}{(\lambda x : \tau.\, d_1)\ (d_2) \longrightarrow [d_2/x]d_1}$$

Fig. 4. Example rules for function application expressions. SAp synthesizes a type for the external expression ($\blacktriangleright_{\to}$ expands the gradual type to a function type if need be), TAAp assigns a type to the internal expression, and ITLam describes how it reduces. The brackets on ITLam indicate that the hypothesis may be omitted for a non-deterministic evaluation system.

relation $d \longrightarrow d'$ defined on redexes. Figure 4 contains the transition relation corresponding to beta reduction, ITLam.

Note that having a gradually typed, user-facing language that elaborates into an internal cast calculus is typical of gradually typed systems [22]. The transitions include standard features like beta reduction, holes blocking reduction, and tracking casts to catch dynamic type errors. Casts are decomposed as much as possible, until they are between top-level type formers applied only to holes (ground types), with inconsistencies at this level resulting in a failed cast, representing a dynamic type error.

2.3 Polymorphic Gradual Typing

The gradually typed lambda calculus is an extension to the simply typed lambda calculus [4] that adds the gradual type (commonly notated as a question mark: ?) to the simply typed lambda calculus, with type equivalence giving way to a (non-transitive) consistency relation between types: the unknown type is consistent with every type [21]. Gradual typing offers a compromise between an untyped and a simply typed calculus; indeed, any untyped term may be embedded into the gradually typed calculus by typing everything at the gradual type. This offers programmers the flexibility to work outside of the type system when they deem it beneficial to do so, and the benefits of allowing this are exemplified in the success of TypeScript [3]. Siek et al. [22] later formalized some properties that hold of the gradually typed lambda calculus that make working in a gradually typed system intuitive for programmers. These properties are collectively known as the gradual guarantee, and are the gold standard properties that are desirable for extensions of the gradually typed lambda calculus. Informally, the gradual

guarantee states that removing parts of type annotations from a program cannot introduce new static or dynamic errors.

System F [8,20] is another extension to the simply typed lambda calculus that adds type functions and polymorphic types. System F has become the go-to model for polymorphic functions, with restrictions of System F becoming the type systems for widely-used functional programming languages such as Haskell, OCaml, F#, and more. One of the reasons System F is so powerful is the strong metatheoretic property of parametricity [9]. Informally, parametricity asserts that a polymorphic function should behave in analogous ways no matter what type the function is instantiated at. This means that one cannot perform computations based on types, and indeed, parametric systems allow for identical evaluation after type erasure [14], which can also be used as a run-time optimization.

There have been several attempts to combine the gradually typed lambda calculus with System F. λB, presented by Ahmed et al. [2], was the first system to add the gradual type to a cast calculus based on System F while preserving parametricity. They achieved this by using type bindings as opposed to type substitutions. In effect, this means that if a polymorphic functions error on any instantiation, then it will error on all instantiations, even if it makes sense to successfully evaluate some of the instantiations. (We will argue that this behavior is undesirable and investigate what metatheoretic properties hold without type bindings.) This also introduces some run-time overhead. System F_G, presented by Igarashi et al. [10], presents a user-facing gradually typed calculus that compiles to a cast calculus, akin to the gradually typed lambda calculus. System F_G is shown to both be parametric as well as satisfy the gradual guarantee – albeit for a modified notion of precision for polymorphic types. Parametric and gradual system PolyG$^\nu$, presented by New et al. [17], requires explicit sealing and unsealing of type variables. The system is based on the intuition that parametricity arises from disallowing computation on types. Gradual System F (GSF) is a system presented by Labrada et al. [12], which, like System F_G, also exhibits parametricity as well as the gradual guarantee. Similarly to the previous systems, this is accomplished with the use of type bindings. However unlike System F_G, a more intuitive notion of precision is defined. Out of the previously presented systems, GSF is the most similar to the one we will present. It is also worth noting the work of Xie et al. [27], who define a system that uses subtyping to provide implicit polymorphism. Notably, their system violates the gradual guarantee due to the necessity of an oracle for some ambiguous instantiations.

3 The System

We extend the types, external expressions, and internal expressions of Hazelnut Live with type variables, universal types, polymorphic abstraction, and type application (Fig. 5).

$$\begin{aligned} \mathsf{Type}\ \tau &::= \ldots \mid \alpha \mid \forall \alpha.\ \tau \\ \mathsf{HExp}\ e &::= \ldots \mid \Lambda \alpha.\ e \mid e\ [\tau] \\ \mathsf{IHExp}\ d &::= \ldots \mid \Lambda \alpha.\ d \mid d\ [\tau] \end{aligned}$$

Fig. 5. Syntax extension of polymorphic Hazel.

With the introduction of the new type form, we define a corresponding matching judgment for expanding the gradual type into a forall form where necessary (Fig. 6). Note that the gradual type is identified with the type hole and is denoted ?.

3.1 Gradually Typed Calculus

We extend the bidirectional typing rules as shown in Fig. 8. We augment typing judgments with type variable contexts Σ, which are sets of type variables in scope. Notably, type functions may be typed both analytically and synthetically. Type functions admit both analytic and synthetic rules. This is a deviation from Dunfield and Krishnaswami's "bidirectional recipe" [6], which prescribes only the analysis rule for type abstractions, since they are introduction forms. We include both rules because doing so improves the expressiveness of the system (for example, the type of the polymorphic identity function $\Lambda\alpha.\ \lambda x : \alpha.\ x$ may be synthesized).

Rule STAP has as a premise $\Sigma \vdash \tau_1$, the well-formedness of τ_1 with respect to Σ. This new judgment is defined in Fig. 7 and must also be inserted as a premise into existing rules for terms involving types, namely ascriptions and annotated lambdas. The well-formedness judgment ensures that all type variables appearing in a term are either bound or appear in the type variable context.

It is at this stage that marking would occur. As previously mentioned, we do not include this process in our presentation; we assume the user provides an expression that is already well typed in the gradually typed calculus. We discuss how our system may be extended to include marking and the use of marking to simulate a form of implicit polymorphism in Sect. 6.

3.2 Elaboration and Cast Calculus

We extend the typed elaboration rules as shown in Fig. 9. Type function applications are elaborated analogously to function applications. The function is cast to the output of the matched type judgment to accommodate the gradual type, and the function is analyzed against this type. Note that to ensure elaboration unicity (further discussed in Appendix A), EATLAM cannot be applied when the function body is a hole, although disallowing type functions in subsumption achieves the same effect. An implicit change from Hazelnut Live is that hole contexts must now assign to each hole a type variable context Σ in addition to a type τ and context Γ.

3.3 Dynamics and Final Forms

Figure 10 contains the type assignment rules for the internal cast calculus. They are the standard System F typing rules, with the addition of the hole context. No matched forall premise is necessary in TATAP because a cast to a forall type has been inserted before the type application during elaboration.

The universal type creates a new ground type case (Fig. 11). The matched ground judgment is used in the ITGROUND and ITEXPAND instruction transitions, which are not presented here as they are not directly modified. The ground type judgment is used to simplify the range of final forms, presented in Fig. 12. We add new value, boxed value, and indeterminate form cases for type functions and casts between universal types. Each normal form is exactly one of these three kinds of final form.

The operational semantics is presented as a contextual semantics. Evaluation contexts must be extended to allow for the new syntactic forms. Since type functions are values, all that is required is to extend evaluation contexts into type function application (omitted from figures as this is standard).

Finally, we add new transition rules. Type functions applied to a type are evaluated by type substitution, as in System F. Note that previous work [2,10, 12,17] avoided this approach, instead choosing to keep a partial mapping from type variables to types. A discussion of the decision to eschew this development and its implications for parametricity and graduality is contained in Sect. 4. In contrast with merely extending GTLC with System F rules, we must add an additional rule that allows type function application to move past casts. The rule is analogous to the rule for term functions.

The instruction transitions are shown in Fig. 13. The bracketed d final premise may be omitted to have an unspecified evaluation strategy. Choosing to including it for eager evaluation creates a deterministic evaluation useful for implementations of the system. Finally, the step relation $d \mapsto d'$ is defined by performing instruction transitions in an evaluation context; this is typical of contextual semantics, and we introduce no modifications to the original presentation, so it is not reproduced here.

$\boxed{\tau \blacktriangleright_\forall \forall\alpha.\ \tau'}$ τ has matched forall type $\forall\alpha.\ \tau'$

$$\text{MFHOLE}\ \frac{}{? \blacktriangleright_\forall \forall\alpha.\ ?} \qquad \text{MFFORALL}\ \frac{}{\forall\alpha.\ \tau \blacktriangleright_\forall \forall\alpha.\ \tau}$$

Fig. 6. Matched forall types.

$\boxed{\Sigma \vdash \tau}$ τ is well-formed in type variable context Σ

$$\text{WFBASE}\ \frac{}{\Sigma \vdash b} \qquad \text{WFHOLE}\ \frac{}{\Sigma \vdash ?} \qquad \text{WFVAR}\ \frac{\alpha \in \Sigma}{\Sigma \vdash \alpha}$$

$$\text{WFARR}\ \frac{\Sigma \vdash \tau_1 \qquad \Sigma \vdash \tau_2}{\Sigma \vdash \tau_1 \to \tau_2} \qquad \text{WFFORALL}\ \frac{\Sigma, \alpha \vdash \tau}{\Sigma \vdash \forall\alpha.\ \tau}$$

Fig. 7. Well-formedness rules.

$\boxed{\Sigma;\Gamma \vdash e \Rightarrow \tau}$ Expression e synthesizes type τ in context $\Sigma;\Gamma$

$$\textsc{StLam}\ \frac{\Sigma,\alpha;\Gamma \vdash e \Rightarrow \tau}{\Sigma;\Gamma \vdash \Lambda\alpha.\, e \Rightarrow \forall\alpha.\, \tau} \qquad \textsc{StAp}\ \frac{\Sigma \vdash \tau_1 \quad \Sigma;\Gamma \vdash e \Rightarrow \tau_2 \quad \tau_2 \blacktriangleright_\forall \forall\alpha.\, \tau_3}{\Sigma;\Gamma \vdash e\ [\tau_1] \Rightarrow [\tau_1/\alpha]\tau_3}$$

$\boxed{\Sigma;\Gamma \vdash e \Leftarrow \tau}$ Expression e analyzes against type τ in context $\Sigma;\Gamma$

$$\textsc{AtLam}\ \frac{\tau_1 \blacktriangleright_\forall \forall\alpha.\, \tau_2 \quad \Sigma,\alpha;\Gamma \vdash e \Leftarrow \tau_2}{\Sigma;\Gamma \vdash \Lambda\alpha.\, e \Leftarrow \tau_1}$$

Fig. 8. Bidirectional typing rules.

$\boxed{\Sigma;\Gamma \vdash e \Rightarrow \tau \rightsquigarrow d \dashv \Delta}$ e synthesizes type τ and elaborates to d with hole context Δ

$$\textsc{EstLam}\ \frac{\Sigma,\alpha;\Gamma \vdash e \Rightarrow \tau \rightsquigarrow d \dashv \Delta}{\Sigma;\Gamma \vdash \Lambda\alpha.\, e \Rightarrow \forall\alpha.\, \tau \rightsquigarrow \Lambda\alpha.\, d \dashv \Delta}$$

$$\textsc{EstAp}\ \frac{\Sigma \vdash \tau_1 \quad \Sigma;\Gamma \vdash e \Rightarrow \tau_2 \quad \tau_2 \blacktriangleright_\forall \forall\alpha.\, \tau_3 \quad \Sigma;\Gamma \vdash e \Leftarrow \forall\alpha.\, \tau_3 \rightsquigarrow d : \tau_4 \dashv \Delta}{\Sigma;\Gamma \vdash e\ [\tau_1] \Rightarrow [\tau_1/\alpha]\tau_3 \rightsquigarrow d\langle \tau_4 \Rightarrow \forall\alpha.\, \tau_3\rangle\ [\tau_1] \dashv \Delta}$$

$\boxed{\Sigma;\Gamma \vdash e \Leftarrow \tau_1 \rightsquigarrow d : \tau_2 \dashv \Delta}$ e analyzes against τ_1 and elaborates to d of consistent type τ_2 with hole context Δ

$$\textsc{EAtLam}\ \frac{e \neq (\!|)^u \quad e \neq (\!|e'|\!)^u \quad \tau_1 \blacktriangleright_\forall \forall\alpha.\, \tau_2 \quad \Sigma,\alpha;\Gamma \vdash e \Leftarrow \tau_2 \rightsquigarrow d : \tau_3 \dashv \Delta}{\Sigma;\Gamma \vdash \Lambda\alpha.\, e \Leftarrow \tau_1 \rightsquigarrow \Lambda\alpha.\, d : \forall\alpha.\, \tau_3 \dashv \Delta}$$

Fig. 9. Elaboration rules from external expressions to internal expressions.

$\boxed{\Delta;\Sigma;\Gamma \vdash d : \tau}$ d is assigned type τ

$$\textsc{TatLam}\ \frac{\Delta;\Sigma,\alpha;\Gamma \vdash d : \tau}{\Delta;\Sigma;\Gamma \vdash \Lambda\alpha.\, d : \forall\alpha.\, \tau} \qquad \textsc{TatAp}\ \frac{\Sigma \vdash \tau_1 \quad \Delta;\Sigma;\Gamma \vdash d : \forall\alpha.\, \tau_2}{\Delta;\Sigma;\Gamma \vdash d\ [\tau_1] : [\tau_1/\alpha]\tau_2}$$

Fig. 10. Type assignment for internal expressions.

$\boxed{\tau\ \mathsf{ground}}$ τ is a ground type

$$\textsc{GForall}\ \frac{}{\forall\alpha.\, ?\ \mathsf{ground}}$$

$\boxed{\tau \blacktriangleright_{\mathsf{ground}} \tau'}$ τ has matched ground type τ'

$$\textsc{MGForall}\ \frac{\forall\alpha.\, \tau \neq \forall\alpha.\, ?}{\forall\alpha.\, \tau \blacktriangleright_{\mathsf{ground}} \forall\alpha.\, ?}$$

Fig. 11. Ground and matched ground rules.

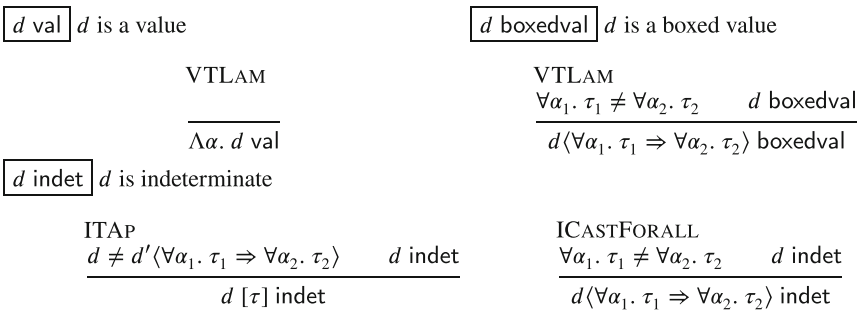

Fig. 12. Final form rules.

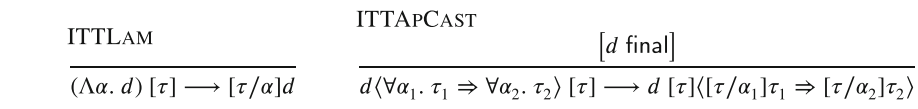

Fig. 13. Instruction transitions.

4 Metatheoretic Properties

Our system conserves all of the typing properties that held of the original system (c.f. Theorems 3.1 through 3.14 in [18]), up to the inclusion of an additional type variable context. Notably, we show type safety of the system:

Theorem 1 (Type Safety). *The system presented in Sect. 3 is type safe:*

1. *Progress: If* $\emptyset \vdash \Delta$ *and* $\Delta; \emptyset; \emptyset \vdash d : \tau$ *then either* d indet, d boxedval, *or there exists an* IHExp d' *such that* $d \mapsto d'$.
2. *Preservation: If* $\emptyset \vdash \Delta$, $\Delta; \emptyset; \emptyset \vdash d : \tau$ *and* $d \mapsto d'$ *then* $\Delta; \emptyset; \emptyset \vdash d' : \tau$.

We also show the properties of elaboration and properties of complete terms that hold of Hazelnut Live. The full statements and explanations of these theorems can be found in Appendix A.

We now discuss the two key metatheoretic properties of this work: parametricity and the gradual guarantee.

4.1 Parametricity

As mentioned before, previous systems obeyed additional restrictions in order to preserve parametricity. These stem from the approach of Ahmed et al. [1], which showed that substitution typing is not parametric, but the approach of using type bindings is. The example to show this presented in Igarashi et al. is thus:

$$f = \Lambda\alpha.\ \lambda x : \mathsf{num}.\ x\langle \mathsf{num} \Rightarrow ? \Rightarrow \alpha \rangle$$

Noting that

$$\begin{aligned} f\ [\mathsf{num}]\ 1 \quad &\mapsto^* 1 \\ f\ [\mathsf{bool}]\ \mathsf{true} &\mapsto^* \mathsf{true}\langle \mathsf{num} \Rightarrow ? \nRightarrow \mathsf{bool} \rangle \end{aligned}$$

where 1 $\mathsf{boxedval}$, and $\mathsf{true}\langle \mathsf{num} \Rightarrow ? \nRightarrow \mathsf{bool} \rangle$ indet. Indeterminate forms correspond to blame/errors in other calculi. Using type bindings would instead result in:

$$f\ [\mathsf{num}]\ 1 \mapsto^* 1\langle \mathsf{num} \Rightarrow ? \nRightarrow \alpha \rangle$$

We argue that creating errors from otherwise sensibly executable programs is against the spirit of live programming with holes, and we would like to avoid doing so. In live programming, we would like to explore the implications of filling the type holes, where it is helpful to not prematurely error. Igarashi et al. further note that type bindings carry overhead, and they introduce static and gradual type variables to allow for substitution typing when no cast to the gradual type exists. Our system does not contain the labels on type variables; they complicate decisions for the programmer, and also complicate the definition of consistency, which must now allow for quasi-polymorphic functions to appear in places expecting polymorphic functions. We furthermore argue that in our setting, since expressions can be holes that might be filled in later, it is impossible to know *a priori* whether a static type variable label is appropriate.

Since the approach of using type bindings (also used in systems such as GSF) only enforces parametricity by introducing unnecessary error states, we instead focus on weakening parametricity. We follow the construction of parametricity presented by Crary [5]. We define equality up to type annotations similarly:

Definition 1. $d_1 =_0 d_2$ *whenever* d_1 *and* d_2 *differ only in syntactic types. Namely, this means that types in lambda annotations, type function applications, and casts may vary, and only such types may vary.*

We define a similar relation for external expressions. We say that a program (term) is *complete* if it does not contain any expression or type holes. If a program is complete and well-typed, all of its casts are identity casts. This can be intuitively thought of as an embedding in System F, which is parametric. We formalize this intuition in the following theorem:

Theorem 2. *Suppose* e_1 complete *and* e_2 complete *and* $\Sigma; \Gamma \vdash e_1 \Rightarrow \tau_1 \rightsquigarrow d_1 \dashv \emptyset$ *and* $\Sigma; \Gamma \vdash e_2 \Rightarrow \tau_2 \rightsquigarrow d_2 \dashv \emptyset$ *and* $e_1 =_0 e_2$.
If $d_1 \mapsto^* v_1$ *and* v_1 val *then there exists* v_2 *such that* $d_2 \mapsto^* v_2$ *and* v_2 val *and* $v_1 =_0 v_2$.

Corollary 1. *If* $\mapsto$ *is confluent and* $d_1 \mapsto^* v_1$ *and* $d_2 \mapsto^* v_2$ *and* v_1 val *and* v_2 final *then* v_2 val *and* $v_1 =_0 v_2$.

We actually have too strong of a hypothesis here; we merely need $d_1 =_0 d_2$ to prove the result, but we focus on external expressions to emphasize that is what the programmer sees and has control over. Note also that the evaluation of all complete terms terminates in a val (this is not proven of the original system, but should also hold from embedding in System F). And so, the hypothesis of the theorem is satisfied by any well-typed, complete expression.

The theorem is proven by formalizing the intuition that all casts are identity casts. We define a relation $=_0'$ that also requires that casts must be identity casts, and is thus a sub-relation of $=_0$. We show that if two terms are $=_0$ and are complete, well-typed programs, then they must be $=_0'$. We show that if $d_1 =_0' d_2$ and $d_1 \mapsto d_1'$ then there exists a d_2' such that $d_1' =_0' d_2'$ and $d_2 \mapsto d_2'$. Thus we can push the relation through the entire evaluation of d_1. Since it relates values to values, then when we reach the end of d_1's trace, we have a v_2 that is a value. (2) follows from (1) by using the unique normal form property (which follows from confluence); the final (normalized) value of d_2 must be unique and so v_2 must be identical to the existentially quantified normal value from (1). We do not show confluence of the system in this work, so we leave it as a hypothesis in the theorem.

In short, this theorem asserts that if we do not deal with graduality then we are able to ensure full parametricity. However, to motivate the use of substitution in our *gradual* system, we also make a guarantee about possibly incomplete terms when using eager evaluation semantics (by which we mean requiring the argument to a beta reduction to be final)[1].

Definition 2. $d_1 =_{casts} d_2$ *when* d_1 *and* d_2 *differ only in syntactic types and presence of casts. More formally, we say* $=_{casts}$ *is the smallest congruence that contains* $=_0$*, and for all types* τ_1, τ_2*,* $d_1 =_{casts} d_2$ *implies* $d_1\langle \tau_1 \Rightarrow \tau_2 \rangle =_{casts} d_2$ *and* $d_1 =_{casts} d_2\langle \tau_1 \Rightarrow \tau_2 \rangle$

Theorem 3. *If* $\Delta; \emptyset; \emptyset \vdash d_1 : \tau_1$ *and* $\Delta; \emptyset; \emptyset \vdash d_2 : \tau_2$ *and* $d_1 =_{casts} d_2$ *and* $d_1 \mapsto^* v_1$ *and* v_1 boxedval *then there exists* v_2 *such that* $d_2 \mapsto^* v_2$ *and either* (v_2 boxedval *and* $v_1 =_{casts} v_2$) *or* v_2 indet.

[1] The reason this is required is that if the argument to a function loops infinitely, if it is lazily evaluated it may be completely thrown away (consider the function $\lambda x :?.\ \lambda y :?.\ y$), whereas if the beta reduction cannot happen due to a failing cast in the function, the argument *will* be continually reduced.

Corollary 2. *If* $\mapsto$ *is confluent and* $\Delta;\emptyset;\emptyset \vdash d_1 : \tau_1$ *and* $\Delta;\emptyset;\emptyset \vdash d_2 : \tau_2$ *and* $d_1 =_{casts} d_2$ *and* $d_1 \mapsto^* v_1$ *and* $d_2 \mapsto^* v_2$ *and* v_1 boxedval *and* v_2 boxedval *then* $v_1 =_{casts} v_2$.

We remark that $=_0$ is a subrelation of $=_{casts}$, so we may have $d_1 =_0 d_2$ in the hypothesis instead. The theorem is proven via induction on the evaluation of d_1. Namely, we show two lemmas:

Lemma 1 (Stepwise Parametricity Lemma). *If* $\Delta;\emptyset;\emptyset \vdash d_1 : \tau_1$ *and* $\Delta;\emptyset;\emptyset \vdash d_2 : \tau_2$ *and* $d_1 =_{casts} d_2$ *and* $d_1 \mapsto d_1'$ *then there exists a* d_2' *such that* $d_2 \mapsto^* d_2'$ *and either* $d_1' = d_2'$ *or* d_2' indet.

Lemma 2 (One-sided Parametricity Lemma). *If* $d_1 =_{casts} d_2$ *and* d_1 boxedval *then there exists a* d_2' *such that* $d_2 \mapsto^* d_2'$ *and* $d_1 =_{casts} d_2'$ *and* d_2' final.

Lemma 1 allows us to use the trace of d_1 as a target for induction until d_1 boxedval. We find a sequence of steps for d_2 that preserve $=_{casts}$; the difficulty here is that we may have to do some sequence of cast reductions before applying the appropriate beta reduction. We proceed by induction on the syntactic sub-expression of d_2 that corresponds to the sub-expression of d_1 that is reduced (picked such that the outer form of the sub-expression is not a cast). We show that we either perform the same beta reduction (in the absence of casts), or we can find a sequence of cast reductions that reduce an ordering[2] and thus we may proceed inductively.

Once this process has finished, Lemma 2 may be applied to show that the evaluation of d_2 terminates, and thus we have the desired final result. The proof of this proceeds similarly to the previous Lemma, noting that d_2 cannot beta reduce due to the $=_{casts}$ constraint and for any trailing casts, there is a sequence of cast reductions that reduces the syntactic size of d_2.

Together, these two properties establish that modulo successful termination, even gradual terms behave in a parametric manner. Existing systems that possess parametricity uniformly introduce cast errors to do so, so we have effectively proven our claim that the difference between those systems and ours is that we raise fewer cast errors on otherwise successfully executing programs.

4.2 Gradual Guarantee

The gradual guarantee specifies how the static and dynamic semantics of a gradual language should behave relative to a precision relation between programs. This relation captures the operation of filling in holes – a program is made

[2] The exact ordering is to, for each natural number n up to the number of function applications, count the number of casts in the function position of n function applications. Then lexicographically order them from largest to smallest n with syntactic size at the end. Such a bizarre ordering must be used as the ITApCast rule increases the number of casts and the size of the term, but moves a cast from the function position to argument and external positions.

more precise by replacing its empty holes with other terms. We shall consider a precision relation on types, external expressions, internal expressions, and hole contexts, all of which we denote $\sqsubseteq$.

Intuitively, the gradual guarantee states that making a program less precise results in more permissive type checking (the *static* gradual guarantee) and a less precise evaluated result (the *dynamic* gradual guarantee). Another way to phrase the static gradual guarantee is that deleting parts of a program, e.g. type annotations, cannot introduce new static errors. We take the formal statement of the gradual guarantee from Siek et al. [22] and adapt it to our notation:

Definition 3. *We say $d \Uparrow$ when there does not exist a v such that $d \mapsto^* v$ and v final (in other words, d diverges).*

Definition 4. *The Gradual Guarantee is a collection of four properties:*
Suppose $e \sqsubseteq e'$ and $\emptyset;\emptyset \vdash e \Rightarrow \tau \rightsquigarrow d \dashv \Delta$.

1. *There exists a τ' such that $\emptyset;\emptyset \vdash e' \Rightarrow \tau'$ with $\tau \sqsubseteq \tau'$.*
2. *There exist a d' and a Δ' such that $\emptyset;\emptyset \vdash e' \Rightarrow \tau' \rightsquigarrow d' \dashv \Delta'$ with $\Delta \sqsubseteq \Delta'$ and $\Delta, \Delta';\emptyset;\emptyset,\emptyset \vdash d \sqsubseteq d'$.*
3. *If $d \mapsto^* v$ with v $\mathsf{boxedval}$ then $d' \mapsto^* v'$ with v' $\mathsf{boxedval}$ and $v \sqsubseteq v'$.*
   If $d \Uparrow$ then $d' \Uparrow$.
4. *If $d' \mapsto^* v'$ with v' $\mathsf{boxedval}$ then $d \mapsto^* v$ such that either v $\mathsf{boxedval}$ and $v \sqsubseteq v'$ or v indet.*
   If $d' \Uparrow$ then $d \Uparrow$ or $d \mapsto^* v$ with v indet.

Of course, what the guarantee establishes depends on the definitions of the various precision relations. Recall that our setting is that of a live programming environment. Thus we would like increased precision to coincide with the intuitive idea of the programmer filling holes in the program. Thus we do not want a definition similar to System F_G's definition, but rather one closer to GSF's definition.

We omit the rules for precision for types and for external expressions, since they are completely straightforward and do not depart from previous presentations of precision in gradual languages, such as for the Gradually Typed Lambda Calculus (GTLC) of Siek et al. The precision relation holds between two syntactic constructs with the same head if it holds between their corresponding children, and anything is more precise than a hole. Using these straightforward definitions, we can prove the following theorem:

Theorem 4. *Our system satisfies Property 1 of Definition 4.*

This property, the static gradual guarantee, states that a less precise term synthesizes a less precise type. It is amenable to direct proof by induction, strengthened to include contexts and both analytic and synthetic typing. We have completed such a proof in our Agda mechanization as discussed in Sect. 5.

The remaining components of the gradual guarantee are not proven for our system, and we leave them to future work. However, there are some features of the system that seem necessary in order to hope for these properties to hold.

Firstly, in order to satisfy Property 2 of Definition 4, a less precise term must analytically elaborate with a less precise type. This is broken with the standard subsumption rule found in Hazelnut Live, in which terms elaborate their synthesized type, even if it is less precise than the analyzed type, so long as the two are consistent. Therefore we consider a modified subsumption rule in Fig. 14. It employs the meet between types, which is the greatest lower bound of types with respect to the aforementioned precision relation, and is a partial function defined only on consistent types. With the new rule, subsumption elaborates with a type that is more precise than both the analyzed and synthesized type, supporting graduality. In order to retain the properties of typed elaboration and elaboration unicity, the rule must insert a cast and must not apply to type functions. It is with this version of the subsumption rule that we have proven the theorems in this paper.

$\boxed{\tau_1 \sqcap \tau_2 = \tau_3}$ The meet of types τ_1 and τ_2 is τ_3

$$\textsc{MeetHoleL}\ \frac{}{? \sqcap \tau = \tau} \qquad \textsc{MeetHoleR}\ \frac{}{\tau \sqcap ? = \tau} \qquad \textsc{MeetBase}\ \frac{}{b \sqcap b = b} \qquad \textsc{MeetVar}\ \frac{}{\alpha \sqcap \alpha = \alpha}$$

$$\textsc{MeetArr}\ \frac{\tau_1 \sqcap \tau_3 = \tau_5 \qquad \tau_2 \sqcap \tau_4 = \tau_6}{(\tau_1 \to \tau_2) \sqcap (\tau_3 \to \tau_4) = \tau_5 \to \tau_6} \qquad \textsc{MeetForall}\ \frac{\tau_1 \sqcap \tau_2 = \tau_3}{(\forall \alpha.\ \tau_1) \sqcap (\forall \alpha.\ \tau_2) = \forall \alpha.\ \tau_3}$$

$\boxed{\Sigma; \Gamma \vdash e \Leftarrow \tau_1 \rightsquigarrow d : \tau_2 \dashv \Delta}$ e analyzes against τ_1 and elaborates to d of consistent type τ_2 with hole context Δ

$$\textsc{EASubsume'}\ \frac{e \neq (\!|\,|\!)^u \qquad e \neq (\!|e'|\!)^u \qquad e \neq \Lambda \alpha.\ e' \qquad \Sigma; \Gamma \vdash e \Rightarrow \tau_2 \rightsquigarrow d \dashv \Delta \qquad \tau_1 \sim \tau_2}{\Sigma; \Gamma \vdash e \Leftarrow \tau_1 \rightsquigarrow d\langle \tau_2 \Rightarrow \tau_1 \sqcap \tau_2 \rangle : \tau_1 \sqcap \tau_2 \dashv \Delta}$$

Fig. 14. Modified subsumption rule.

Properties 2–4 of Definition 4 depend on the precision relation for internal expressions, which is considerably more subtle than that for external expressions and types. Siek et al. define an appropriate definition of internal precision for the GTLC, and prove the gradual guarantee for this system. We adapt this relation to our system. The rules that do not involve casts or the empty hole are analogous to the rules for external expressions, so are elided. The remaining rules are provided in Fig. 15. The rules involving casts correspond directly to the rules for term precision for the GTLC's cast calculus. Since internal holes carry type information, a term is only more precise than a hole if its assigned type is more precise than the hole's contextually assigned type. Since internal precision involves type assignment, it depends on a pair of contexts, one for each side of the relation, which may share a type context since the different contexts need only vary by term precision.

We define $\Gamma \sqsubseteq \Gamma'$ to hold when Γ and Γ' have equal domains and for all x such that $x : \tau \in \Gamma$ and $x : \tau' \in \Gamma'$, $\tau \sqsubseteq \tau'$. We define $\Delta \sqsubseteq \Delta'$ to hold when for all u such that $u :: \tau[\Sigma; \Gamma] \in \Delta$ and $u :: \tau'[\Sigma; \Gamma'] \in \Delta'$, $\tau \sqsubseteq \tau'$ and $\Gamma \sqsubseteq \Gamma'$.

$\boxed{\Delta, \Delta'; \Sigma; \Gamma, \Gamma' \vdash d \sqsubseteq d'}$ d in context $\Delta; \Sigma; \Gamma$ is more precise than d' in context $\Delta'; \Sigma; \Gamma'$

$$\cdots \qquad \textsc{PIEHole}\ \frac{\Delta; \Sigma; \Gamma \vdash d : \tau \qquad u :: \tau'[\Sigma'; \Gamma''] \in \Delta' \qquad \tau \sqsubseteq \tau'}{\Delta, \Delta'; \Sigma; \Gamma, \Gamma' \vdash d \sqsubseteq (\!|\,|\!)^u_\sigma}$$

$$\textsc{PICast}\ \frac{\Delta, \Delta'; \Sigma; \Gamma, \Gamma' \vdash d \sqsubseteq d' \qquad \tau_1 \sqsubseteq \tau_1' \qquad \tau_2 \sqsubseteq \tau_2'}{\Delta, \Delta'; \Sigma; \Gamma, \Gamma' \vdash d\langle \tau_1 \Rightarrow \tau_2 \rangle \sqsubseteq d'\langle \tau_1' \Rightarrow \tau_2' \rangle} \qquad \textsc{PIFailedCast}\ \frac{\Delta'; \Sigma; \Gamma' \vdash d' : \tau \qquad \tau_2 \sqsubseteq \tau}{\Delta, \Delta'; \Sigma; \Gamma, \Gamma' \vdash d\langle \tau_1 \Rightarrow ? \not\Rightarrow \tau_2 \rangle \sqsubseteq d'}$$

$$\textsc{PIRemoveCast}\ \frac{\Delta, \Delta'; \Sigma; \Gamma, \Gamma' \vdash d \sqsubseteq d' \qquad \Delta'; \Sigma; \Gamma' \vdash d' : \tau' \qquad \tau_1 \sqsubseteq \tau' \qquad \tau_2 \sqsubseteq \tau'}{\Delta, \Delta'; \Sigma; \Gamma, \Gamma' \vdash d\langle \tau_1 \Rightarrow \tau_2 \rangle \sqsubseteq d'}$$

$$\textsc{PIAddCast}\ \frac{\Delta, \Delta'; \Sigma; \Gamma, \Gamma' \vdash d \sqsubseteq d' \qquad \Delta; \Sigma; \Gamma \vdash d : \tau \qquad \tau \sqsubseteq \tau_1 \qquad \tau \sqsubseteq \tau_2}{\Delta, \Delta'; \Sigma; \Gamma, \Gamma' \vdash d \sqsubseteq d'\langle \tau_1 \Rightarrow \tau_2 \rangle}$$

Fig. 15. Precision between internal expressions

Having adapted internal precision from the GTLC to our system, the gradual guarantee is fully defined. We have proven the static gradual guarantee, but it remains to be seen whether the rest of the properties hold, as they do for the GTLC. The updated subsumption rule is necessary for typed elaboration to behave monotonically on types.

5 Implementation

5.1 Agda Mechanization

This paper is accompanied by an Agda mechanization of the system and proofs of most of the theorems[3]. The mechanization is based on that of Hazelnut Live, with a few differences. This mechanization uses de Bruijn indices to represent both term and type bindings, and uses an ordered, combined context rather than independent term and type contexts. It also does not include hole names or hole contexts, and therefore does not capture the formalism for the *fill-and-resume* operation of Hazelnut Live, since the present work does not extend that part of the theory. The mechanization uses the updated subsumption rule for

[3] The code is available at https://github.com/hazelgrove/hazelnut-polymorphism-agda/.

elaboration, EASubsume'. Due to the absence of hole contexts and the new subsumption rule, the analytic elaboration rules for holes can be removed entirely, as can the premises disallowing holes in the subsumption elaboration rule. Matched arrow and matched forall judgments are implemented using the meet operation.

The following theorems have been proven: all parts of Theorem 5, Theorem 1, and Theorem 6, along with the parametricity theorems (Theorems 2 and 3) and the static gradual guarantee (Theorem 4). The repository indicates where to find the proof of each theorem.

5.2 Implementation

We have fully implemented the polymorphic system into the Hazel programming environment[4]. Notably, the implementation coexists with other extensions to Hazelnut Live, such as algebraic data types, recursion, type aliases, etc.; the combination of all of these features has not been formalized. Hazel is implemented in ReasonML and uses js_of_ocaml [25] to compile to a website.

Refer again to Fig. 1 for the appearance of the Hazel user interface. The user-facing gradually typed calculus is input via a gradual structure editor that uses obligations (see Tylr [15], the basis for Hazel's input system[5]); for example, inserting a `typfun` creates an obligation for a `->` and inserts the appropriate expression hole. Type function application is denoted with `@< >`.

6 Towards Implicit Polymorphism

For practical programming purposes, it is burdensome to write explicit type applications for each use of a polymorphic term. Therefore, many general-purpose functional languages adopt implicit polymorphism, in which the type applications are left out of the concrete syntax, and the instantiated types are statically inferred. There exist bidirectional calculi for implicit polymorphism, such as in Dunfield and Krishnaswami [7], which we could have chosen to gradualize in the same manner as we did for System F above.

Instead we propose to take advantage of the structured editing capabilities of Hazel. The widespread practice of implicit language features represents a compromise between the interests of the language user and the language developer. Compared to explicit features, the user benefits by typing and seeing less code, and by achieving more flexible code, at the cost of language transparency, consistency, and control. The implementer benefits by maintaining the same user interface and language architecture, only needing to insert an instantiation phase in the language processing pipeline, at the cost of increased language complexity.

We believe that by improving the programming environment architecture, this compromise can in turn be improved. Hazel is a gapless editor, meaning that at every point in time, syntactic information, static information, and all

[4] The Hazel project is described, with a link to the source code, at https://hazel.org.

[5] In short, there is an indication for necessary syntactic forms that must be added before a valid editor state can be reached.

downstream services are maintained by the editor. In this context we propose a system of implicit polymorphism in which the editor maintains appropriate type applications in the visible surface syntax of the program. This improves the transparency and regularity of the language while retaining the ease of editing and flexibility of implicit systems.

For example, this system would ideally insert the $[A]$ and $[B]$ type applications into the program below, supposing that $f : A \to B$ and $l : A$ list.

map $[A]$ $[B]$ (f) (l)

These type applications could be folded by default to avoid cluttering the screen with useless information. Despite this diminution of the type application forms, the proposed strategy is distinct from usual implicit schemes because the persisted program will retain the inferred type applications, and because the user will be able to see and edit the type arguments if needed. Figures 16, 17, 18 and 19 display mock ups of various editor states in a hypothetical version of Hazel with implicit polymorphism.

6.1 Mark Insertion

Zhao et al. [28] describe the "mark insertion" component of the Hazel architecture. Hazel programs begin as unmarked expressions e, corresponding exactly to the external expressions in Hazelnut Live except that they do not contain nonempty holes. Next comes a bidirectionally typed mark insertion phase, with judgment forms $\Sigma; \Gamma \vdash e \leadsto \check{e} \Rightarrow \tau$ and $\Sigma; \Gamma \vdash e \leadsto \check{e} \Leftarrow \tau$. Each unmarked expression e is mapped to a corresponding marked expression $\check{e}$, which is identical to e except for the presence of annotated nonempty holes called marks. Both unmarked and marked expressions have typing rules, and the mark insertion phase inserts the minimal marks needed to produce a well-typed result, essentially sectioning off ill-typed subexpressions with informatively annotated nonempty holes. After mark insertion comes the elaboration and evaluation stages of Hazelnut Live.

Figure 20 contains the mark insertion rules for the polymorphic fragment. The other mark insertion rules are similarly related to the basic typing rules. For each typing rule, there is a mark insertion rule that inserts no marks in the case that the premises of the typing rule are met. Each check in the premise of a typing rule gives rise to an additional mark insertion rule which inserts an appropriate mark when the check fails.

Mark insertion should satisfy certain metatheoretic properties. These properties include totality and unicity, which mean that the insertion operation is a total function on unmarked expressions. Mark insertion should only generate well-typed terms, and erasing marks from the marked term should recover the original term. The mark insertion process should not affect terms that already type check, and should insert at least one mark into terms that do not type check. These properties hold of the original marked lambda calculus, but do not all hold of our polymorphic extension. The reason is that a mark inserted around a type abstraction in analytic position may not be well typed. Considering the mark

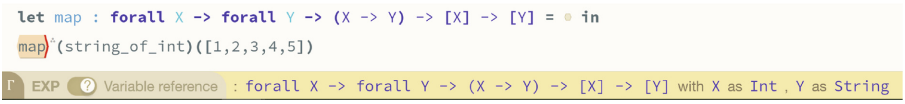

Fig. 16. Hypothetical behavior: when a type application insertion succeeds, the type arguments are listed along with the type of the polymorphic term. An ellipsis mark indicates folded code and provides a way to examine it.

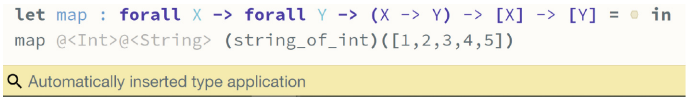

Fig. 17. Hypothetical behavior: when the ellipsis mark is selected, it expands to reveal the explicit type applications and arguments that have been inserted by the editor. If this code is edited by the user, it becomes fully explicit and is colored accordingly.

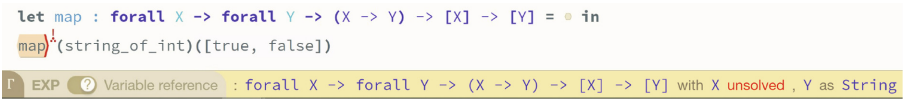

Fig. 18. Hypothetical behavior: when a type application insertion fails, the conflicted type arguments are indicated. The ellipsis mark signals an error.

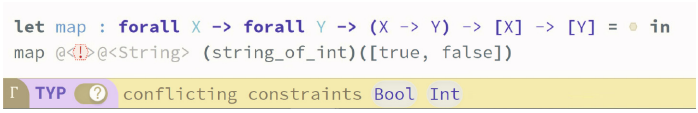

Fig. 19. Hypothetical behavior: the editor displays the conflicting required refinements of the unfillable type argument hole. If the user hovers over or selects one of the refinements, it will be applied to the hole, resulting in errors elsewhere in the code.

$\boxed{\Sigma;\Gamma \vdash e \leadsto \check{e} \Rightarrow \tau}$ e is marked into $\check{e}$ and synthesizes type τ

$$\textsc{MkSTypeLam}\ \frac{\Sigma, \alpha;\Gamma \vdash e \leadsto \check{e} \Rightarrow \tau}{\Sigma;\Gamma \vdash \Lambda\alpha.\, e \leadsto \Lambda\alpha.\, \check{e} \Rightarrow \forall\alpha.\, \tau} \qquad \textsc{MkSTypeAp1}\ \frac{\Sigma;\Gamma \vdash e \leadsto \check{e} \Rightarrow \tau \qquad \tau \blacktriangleright_\forall \forall\alpha.\, \tau_1}{\Sigma;\Gamma \vdash e\,[\tau_2] \leadsto \check{e}\,[\tau_2] \Rightarrow \tau_1[\tau_2/\alpha]}$$

$$\textsc{MkSTypeAp2}\ \frac{\Sigma;\Gamma \vdash e \leadsto \check{e} \Rightarrow \tau_1 \qquad \tau_1 \blacktriangleright_\forall}{\Sigma;\Gamma \vdash e\,[\tau_2] \leadsto \langle\!|\check{e}|\!\rangle^{\Rightarrow}_{\blacktriangleright_\forall}\,[\tau_2] \Rightarrow ?}$$

$\boxed{\Sigma;\Gamma \vdash e \leadsto \check{e} \Leftarrow \tau}$ e is marked into $\check{e}$ and analyzes against type τ

$$\textsc{MkATypeLam1}\ \frac{\tau_1 \blacktriangleright_\forall \forall\alpha.\, \tau_2 \qquad \Sigma, \alpha;\Gamma \vdash e \leadsto \check{e} \Leftarrow \tau_2}{\Sigma;\Gamma \vdash \Lambda\alpha.\, e \leadsto \Lambda\alpha.\, \check{e} \Leftarrow \tau_1} \qquad \textsc{MkATypeLam2}\ \frac{\tau \blacktriangleright_\forall \qquad \Sigma, \alpha;\Gamma \vdash e \leadsto \check{e} \Leftarrow ?}{\Sigma;\Gamma \vdash \Lambda\alpha.\, e \leadsto \langle\!|\Lambda\alpha.\, \check{e}|\!\rangle^{\Leftarrow}_{\blacktriangleright_\forall} \Leftarrow \tau}$$

Fig. 20. Mark insertion rules

to be a nonempty hole, it can only analyze against a type if its contents synthesizes a type, and in the case of a type abstraction, this only holds if its body synthesizes a type. Since our language includes unannotated lambda abstractions, which do not have a type synthesis rule, this case is possible. Concretely, judgements such as:

$$\Sigma; \Gamma \vdash (\!|\Lambda\alpha.\ \lambda x.\ e|\!)^{\Leftarrow}_{\blacktriangleright_\forall} \Leftarrow \tau$$

do not hold, even if the term is the output of the marking process. To address this, additional typing and elaboration rules for marked type abstractions would need to be added.

6.2 Type Application Insertion

The mark insertion phase currently uses the bidirectional typing flow to insert marks into a Hazel program where there would otherwise be a static error. Type applications may be inserted by enriching this operation, so that some static errors are addressed not by inserting a mark, but by inserting a type applications with a type hole as the argument. Specifically, when a polymorphic term is found, but is inconsistent with the expected type or type former, a type application may be inserted rather than a mark. The new rules for these insertions at type applications are given in Fig. 21. These rules replace the previous rules for applications presented in Zhao et al. Rules for type application insertion at projections and at subsumptions are omitted for brevity.

$\boxed{\Sigma; \Gamma \vdash e \leadsto \check{e} \Rightarrow \tau}$ Synthetic marking judgment

$$\textsc{InsertSAp1}\quad \frac{\Sigma; \Gamma \vdash e_1 \leadsto \check{e}_1 \Rightarrow \tau_1 \qquad \tau_1 \blacktriangleright_\to \tau_2 \to \tau_3 \qquad \Sigma; \Gamma \vdash e_2 \leadsto \check{e}_2 \Leftarrow \tau_2}{\Sigma; \Gamma \vdash e_1\ e_2 \leadsto \check{e}_1\ \check{e}_2 \Rightarrow \tau_3}$$

$$\textsc{InsertSAp2}\quad \frac{\Sigma; \Gamma \vdash e_1 \leadsto \check{e}_1 \Rightarrow \tau_1 \qquad \tau_1 \blacktriangleright_{\not\to} \qquad \forall^{\square}(\tau_1) \blacktriangleright_\to \tau_2 \to \tau_3 \qquad \Sigma; \Gamma \vdash e_1\ [?]\ e_2 \leadsto \check{e}_3 \Rightarrow \tau_4}{\Sigma; \Gamma \vdash e_1\ e_2 \leadsto \check{e}_3 \Rightarrow \tau_4}$$

$$\textsc{InsertSAp3}\quad \frac{\Sigma; \Gamma \vdash e_1 \leadsto \check{e}_1 \Rightarrow \tau \qquad \forall^{\square}(\tau) \blacktriangleright_{\not\to} \qquad \Sigma; \Gamma \vdash e_2 \leadsto \check{e}_2 \Leftarrow ?}{\Sigma; \Gamma \vdash e_1\ e_2 \leadsto (\!|\check{e}_1|\!)^{\Rightarrow}_{\blacktriangleright_{\not\to}}\ \check{e}_2 \Rightarrow ?}$$

Fig. 21. Implicit insertion.

These type application insertion rules are derived from the mark insertion rules in Zhao et al. The rules that have been updated are those with a premise of the form $\tau \blacktriangleright_\to \tau_1 \to \tau_2$, $\tau \blacktriangleright_{\not\to}$, $\tau \blacktriangleright_\times \tau_1 \times \tau_2$, $\tau \blacktriangleright_{\not\times}$, $\tau \sim \tau_1$, or $\tau \not\sim \tau_1$, where

τ and none of the other types involved are synthesized from a subexpression of the expression being marked. These conditions correspond to an opportunity to insert a type application that may avoid a failed type matching or consistency check, and thereby avoid a mark insertion.

The mark insertion rule for conditionals involves a consistency check between the types synthesized from the branches of the conditional. It is possible to write valid type application insertion rules for conditionals, but it is not clear what should be done in the case of inconsistent polymorphic branches.

In order to gauge when it is appropriate to insert a type application, we introduce a prenex erasure operation $\forall^{\square}(\tau)$, which erases all leading $\forall$. constructors of τ and replaces their bound variables with ?. If a type matching or consistency check fails on the originally synthesized type, the check is retried on the prenex erased type. This new check corresponds to whether, according to the structure of the type, it may be possible to pass the type matching or consistency check after type applications are inserted around the subexpression.

The new rules are designed so that the new mark and type application insertion operation retains most desirable metatheoretic properties of the original mark insertion operation. The combined insertion phase should still be a total function, generate well-typed terms, and not affect terms which already type check. However, it is no longer the case that the operation's only effect is the insertion of marks. The new rules ought to ensure that erasing all marks and some subset of the type applications recovers the original term, and that the insertion operation applied to a term that does not type check produces a term that includes either a mark or a type application. These properties remain conjectural for the new system, but they should be straightforward to prove.

6.3 Type Arguments

The type application insertion process described above simply inserts type holes as arguments, which is not satisfactory for handling type errors that arise from implicit polymorphic code. Therefore we propose a static phase for instantiating type arguments, which occurs after mark and type application insertion and before elaboration into the internal calculus. Ideally this phase would simply reuse type hole inference machinery from mark insertion, which uses constraints on type holes generated during the mark insertion phase to generate the possible fillings for the type holes that appear in the program. To simulate implicit instantiation, when the constraints for an inserted type hole contain no conflicts, the editor would automatically fill the hole accordingly. When there are conflicts, the same user interface that appears in Zhao et al. would be used to convey this information to the user, who could then select an option for filling the hole.

Unfortunately, the type hole inference technique dose not directly generalize to the polymorphic setting. As the type level of System F is isomorphic to the untyped lambda calculus, the constraints on type holes comprise general higher-order unification problems, the solution of which is not decidable. For example, the code below generates the higher order unification problem $?^1\ (?^3) = ?^2$, where application between types is defined so as to obey the obvious beta rule.

$\mathsf{let}\ f :?^1 = ⦇⦈\ \mathsf{in}\ \mathsf{let}\ x :?^2 = f\ [?^3]\ \mathsf{in}\ ⦇⦈$

However, this is a rather unnatural example. In many cases, a programmer is applying a polymorphic library function, like map, the type of which is completely known, in the sense of containing no metavariables (holes). This suggests the following algorithm: perform standard unification, eagerly simplifying constraints by applying (language level, not meta-level) substitutions into types that are free of metavariables. All other substitutions are stuck, but may become unstuck as metavariables are solved. Constraints involving perpetually stuck substitutions would never be used, the fact of which is a manifestation of the incompleteness of the algorithm, but many useful cases, like the application of polymorphic library functions, could be solved.

7 Related and Future Work

Fill and Resume. Hazelnut Live presented a notion of fill-and-resume. That is, that a program could be evaluated, after which the programmer fills in (i.e. replaces) a hole with a valid expression. Then, because program evaluation is pure, the operation of program reduction commutes with replacing the hole, so the evaluator can replace the corresponding hole(s) in the evaluated expression, and continue evaluating. This required a notion of tracking substitutions that occurred in the closure around holes, and replaying those substitutions on the newly provided expression. This has a very close connection to contextual modal type theory, with the hole context tracking contexts. These substitutions were a part of the cast calculus, and their validity was checked via substitution typing, which is used as a premise to type assignment on holes.

We do not argue for the correctness of fill-and-resume in this work, but we conjecture it to still be valid. This is because our system is still pure, so evaluation should still commute with hole filling. There are subtle issues with naively extending substitution typing. It is clear the substitutions must now also track type substitutions. Term substitutions that happen after a type substitution may have their typing affected, and it is not immediately obvious how to account for this with a static typing judgment. An analogous problem does not exist in the original formulation, since substituting in sub-terms does not change the type of a term, which is all that is tracked in the substitution typing.

Thus, we leave proving validity of fill-and-resume with corresponding substitution typing judgments as future work.

Implicit Polymorphism. We have described a system for allowing polymorphic terms to be used without an explicit type application, as with implicit polymorphism. Yet this editor service does not address implicit polymorphism on a theoretical level, and may fail to catch type errors that a truly implicit system can. As seen in Xie et al. [27], implicit polymorphism may force instantiations that cause errors that may be resolved with additional typing information, violating the gradual guarantee. As far we know, the problem of a gradually parametric implicit polymorphic system has yet to be solved. We are interested in whether

such a system exists, and whether the solution to this problem relates to the problems described previously and could be adapted to type hole inference.

References. References see popular use even in functional programming languages, such as the ML family of languages. However, references have not yet been implemented into the Hazel programming environment, nor has there been development on the theory of how references interact with expression holes. Siek et al. [23] have shown that references can work with the gradually typed lambda calculus. We are unaware of any work that adds references to a polymorphic gradually typed calculus.

It appears that combining graduality, polymorphism, hole expressions, and references creates unique problems; for example, type $\forall\alpha.\ \alpha$ ref can be populated with $\Lambda\alpha.$ ref(⦇⦈), but cannot be populated in a system without expression holes. Such examples that create a new reference with each type function application may preclude future attempts at type erasure run-time semantics, which are otherwise a promising optimization as shown in Igarashi et al.

Acknowledgements. The authors would like to thank the referees and attendees of TFP 2024 for their insightful feedback about earlier drafts of this work. Adam Chen would like to acknowledge Eric Koskinen for helping with the opportunity and logistics for working on a project outside of the topic of their funded work. We would also like to acknowledge Kevin Li and Yuchen Jiang for the preliminary implementation of type functions and polymorphic types in the Hazel codebase. This work was partially funded through the NSF grant #CCF-2238744.

A Type Safety Theorems

Our system conserves all of the typing properties that held of the original system (c.f. Theorems 3.1 through 3.14 in [18]). To begin with, the bidirectional typing allows for unique elaboration to a term of a consistent type:

Theorem 5. *The following properties hold:*

- *Elaborability: any term typable by the bidirectional system has an elaboration.*
  1. *If* $\Sigma;\Gamma \vdash e \Rightarrow \tau$ *then there exist* d *and* Δ *such that* $\Sigma;\Gamma \vdash e \Rightarrow \tau \rightsquigarrow d \dashv \Delta$.
  2. *If* $\Sigma;\Gamma \vdash e \Leftarrow \tau$ *then there exist* d, τ', *and* Δ *such that* $\Sigma;\Gamma \vdash e \Leftarrow \tau \rightsquigarrow d : \tau' \dashv \Delta$.
- *Elaboration Generality: the converse of the above is true.*
  1. *If* $\Sigma;\Gamma \vdash e \Rightarrow \tau \rightsquigarrow d \dashv \Delta$ *then* $\Sigma;\Gamma \vdash e \Rightarrow \tau$.
  2. *If* $\Sigma;\Gamma \vdash e \Leftarrow \tau \rightsquigarrow d : \tau' \dashv \Delta$ *then* $\Sigma;\Gamma \vdash e \Leftarrow \tau$.
- *Elaboration Unicity: elaboration of terms is unique.*
  1. *If* $\Sigma;\Gamma \vdash e \Rightarrow \tau_1 \rightsquigarrow d_1 \dashv \Delta_1$ *and* $\Sigma;\Gamma \vdash e \Rightarrow \tau_2 \rightsquigarrow d_2 \dashv \Delta_2$ *then* $\tau_1 = \tau_2$, $d_1 = d_2$, *and* $\Delta_1 = \Delta_2$.
  2. *If* $\Sigma;\Gamma \vdash e \Leftarrow \tau \rightsquigarrow d_1 : \tau_1 \dashv \Delta_1$ *and* $\Sigma;\Gamma \vdash e \Leftarrow \tau \rightsquigarrow d_2 : \tau_2 \dashv \Delta_2$ *then* $\tau_1 = \tau_2$, $d_1 = d_2$, *and* $\Delta_1 = \Delta_2$.

– *Typed Elaboration: the elaboration is consistent with the type assignment system.*
  1. *If* $\Sigma; \Gamma \vdash e \Rightarrow \tau \rightsquigarrow d \dashv \Delta$ *then* $\Delta; \Sigma; \Gamma \vdash d : \tau$.
  2. *If* $\Sigma; \Gamma \vdash e \Leftarrow \tau \rightsquigarrow d : \tau' \dashv \Delta$ *then* $\Delta; \Sigma; \Gamma \vdash d : \tau'$ *with* $\tau \sim \tau'$.
– *Type Assignment Unicity: type assignment assigns a unique type. If* $\Delta; \Sigma; \Gamma \vdash d : \tau$ *and* $\Delta; \Sigma; \Gamma \vdash d : \tau'$ *then* $\tau = \tau'$

In short, these properties show that elaboration defines a unique embedding from the user-facing gradually typed calculus into the typed cast calculus. Thus it is sufficient to state type safety solely in terms of the cast calculus. We repeat that we prove that the system with the instruction transitions defined in Fig. 13 is type safe:

Theorem 1 (Type Safety). *The system presented in Sect. 3 is type safe:*

1. *Progress: If* $\emptyset \vdash \Delta$ *and* $\Delta; \emptyset; \emptyset \vdash d : \tau$ *then either* d indet, d boxedval, *or there exists an* IHExp d' *such that* $d \mapsto d'$.
2. *Preservation: If* $\emptyset \vdash \Delta$, $\Delta; \emptyset; \emptyset \vdash d : \tau$ *and* $d \mapsto d'$ *then* $\Delta; \emptyset; \emptyset \vdash d' : \tau$.

Recall that a program (term) is *complete* if it does not contain any expression or type holes. Complete programs are elaborated into internal expressions with only identity casts and without type or expression holes. This fragment of internal expressions is equivalent to System F, and therefore recovers its properties such as strong normalization. These properties are formalized in the following theorem that our system conserves from Hazelnut Live [18]:

Theorem 6. *The following properties about complete programs hold:*

1. *Complete Elaboration: If* Γ complete, e complete, *and* $\Gamma; \Sigma \vdash e \Rightarrow \tau \rightsquigarrow d \dashv \Delta$ *then* τ complete, d complete, *and* $\Delta = \emptyset$.
2. *Complete Preservation: If* d complete, $\Delta; \Sigma; \Gamma \vdash d : \tau$, *and* $d \mapsto d'$ *then* d' complete *and* $\Delta; \Sigma; \Gamma \vdash d' : \tau$
3. *Complete Progress: If* d complete *and* $\Delta; \Sigma; \Gamma \vdash d : \tau$ *then either* d val *or there exists an* IHExp d' *such that* $d \mapsto d'$.

Complete elaboration states that a complete program in the user-facing gradually typed calculus elaborates into a complete program in the cast calculus. Complete preservation states that the step relation preserves completeness as well as typing, and complete progress states that every complete term is a val or can step.

References

1. Ahmed, A., Findler, R.B., Siek, J.G., Wadler, P.: Blame for all. In: Ball, T., Sagiv, M. (eds.) Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, 26–28 January 2011, pp. 201–214. ACM (2011). https://doi.org/10.1145/1926385.1926409

2. Ahmed, A., Jamner, D., Siek, J.G., Wadler, P.: Theorems for free for free: parametricity, with and without types. Proc. ACM Program. Lang. **1**(ICFP), 39:1–39:28 (2017). https://doi.org/10.1145/3110283
3. Bierman, G.M., Abadi, M., Torgersen, M.: Understanding typescript. In: Jones, R.E. (ed.) ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, 28 July–1 August 2014, Proceedings. LNCS, vol. 8586, pp. 257–281. Springer, Cham (2014). https://doi.org/10.1007/978-3-662-44202-9_11
4. Church, A.: A formulation of the simple theory of types. J. Symb. Log. **5**(2), 56–68 (1940). https://doi.org/10.2307/2266170
5. Crary, K.: A simple proof technique for certain parametricity results. In: Rémy, D., Lee, P. (eds.) Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP 1999), Paris, France, 27–29 September 1999, pp. 82–89. ACM (1999). https://doi.org/10.1145/317636.317787
6. Dunfield, J., Krishnaswami, N.: Bidirectional typing. ACM Comput. Surv. **54**(5), 98:1–98:38 (2022). https://doi.org/10.1145/3450952
7. Dunfield, J., Krishnaswami, N.R.: Complete and easy bidirectional typechecking for higher-rank polymorphism. CoRR abs/1306.6032 (2013). http://arxiv.org/abs/1306.6032
8. Girard, J.Y.: Interpretation fonctionelle et elimination des coupures dans l'aritmetique d'ordre superieur (1972). https://api.semanticscholar.org/CorpusID:117631778
9. Harper, R.: Practical Foundations for Programming Languages, 2nd edn. Cambridge University Press, Cambridge (2016). https://www.cs.cmu.edu/%7Erwh/pfpl/index.html
10. Igarashi, Y., Sekiyama, T., Igarashi, A.: On polymorphic gradual typing. Proc. ACM Program. Lang. **1**(ICFP), 40:1–40:29 (2017). https://doi.org/10.1145/3110284
11. Kluyver, T., et al.: Jupyter notebooks - a publishing format for reproducible computational workflows. In: Loizides, F., Schmidt, B. (eds.) Positioning and Power in Academic Publishing: Players, Agents and Agendas, 20th International Conference on Electronic Publishing, Göttingen, Germany, 7–9 June 2016, pp. 87–90. IOS Press (2016). https://doi.org/10.3233/978-1-61499-649-1-87
12. Labrada, E., Toro, M., Tanter, É.: Gradual system F. J. ACM **69**(5), 38:1–38:78 (2022). https://doi.org/10.1145/3555986
13. McCauley, R., et al.: Debugging: a review of the literature from an educational perspective. Comput. Sci. Educ. **18**(2), 67–92 (2008). https://doi.org/10.1080/08993400802114581
14. Mishra-Linger, N., Sheard, T.: Erasure and polymorphism in pure type systems. In: Amadio, R.M. (ed.) Foundations of Software Science and Computational Structures, 11th International Conference, FOSSACS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, 29 March–6 April 2008, Proceedings. LNCS, vol. 4962, pp. 350–364. Springer, Cham (2008). https://doi.org/10.1007/978-3-540-78499-9_25
15. Moon, D., Blinn, A., Omar, C.: Gradual structure editing with obligations. In: IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2023, Washington, DC, USA, 3–6 October 2023, pp. 71–81. IEEE (2023). https://doi.org/10.1109/VL-HCC57772.2023.00016
16. Nanevski, A., Pfenning, F., Pientka, B.: Contextual modal type theory. ACM Trans. Comput. Log. **9**(3), 23:1–23:49 (2008). https://doi.org/10.1145/1352582.1352591

17. New, M.S., Jamner, D., Ahmed, A.: Graduality and parametricity: together again for the first time. Proc. ACM Program. Lang. **4**(POPL), 46:1–46:32 (2020). https://doi.org/10.1145/3371114
18. Omar, C., Voysey, I., Chugh, R., Hammer, M.A.: Live functional programming with typed holes. Proc. ACM Program. Lang. **3**(POPL), 14:1–14:32 (2019). https://doi.org/10.1145/3290327
19. Omar, C., Voysey, I., Hilton, M., Aldrich, J., Hammer, M.A.: Hazelnut: a bidirectionally typed structure editor calculus. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, 18–20 January 2017, pp. 86–99. ACM (2017). https://doi.org/10.1145/3009837.3009900
20. Reynolds, J.C.: Towards a theory of type structure. In: Robinet, B.J. (ed.) Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, 9–11 April 1974. LNCS, vol. 19, pp. 408–423. Springer, Cham (1974). https://doi.org/10.1007/3-540-06859-7_148
21. Siek, J., Taha, W.: Gradual typing for functional languages, January 2006
22. Siek, J.G., Vitousek, M.M., Cimini, M., Boyland, J.T.: Refined criteria for gradual typing. In: Ball, T., Bodík, R., Krishnamurthi, S., Lerner, B.S., Morrisett, G. (eds.) 1st Summit on Advances in Programming Languages, SNAPL 2015, 3–6 May 2015, Asilomar, California, USA. LIPIcs, vol. 32, pp. 274–293. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2015). https://doi.org/10.4230/LIPICS.SNAPL.2015.274
23. Siek, J.G., Vitousek, M.M., Cimini, M., Tobin-Hochstadt, S., Garcia, R.: Monotonic references for efficient gradual typing. In: Vitek, J. (ed.) ESOP 2015. LNCS, vol. 9032, pp. 432–456. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46669-8_18
24. Tanimoto, S.L.: A perspective on the evolution of live programming. In: Burg, B., Kuhn, A., Parnin, C. (eds.) Proceedings of the 1st International Workshop on Live Programming, LIVE 2013, San Francisco, California, USA, 19 May 2013, pp. 31–34. IEEE Computer Society (2013). https://doi.org/10.1109/LIVE.2013.6617346
25. Vouillon, J., Belat, V.: From bytecode to javascript: the js_of_ocaml compiler. Softw. Pract. Experience **44**(8), 951–972 (2014). https://doi.org/10.1002/spe.2187
26. Wakeling, D.: Spreadsheet functional programming. J. Funct. Program. **17**(1), 131–143 (2007). https://doi.org/10.1017/S0956796806006186
27. Xie, N., Bi, X., D. S. Oliveira, B.C., Schrijvers, T.: Consistent subtyping for all. ACM Trans. Program. Lang. Syst. **42**(1), 2:1–2:79 (2020). https://doi.org/10.1145/3310339
28. Zhao, E., Maroof, R., Dukkipati, A., Pan, Z., Omar, C.: Total type error localization and recovery with holes. Proc. ACM Program. Lang. **8**(POPL), 2041–2068 (2024). https://doi.org/10.1145/3632910