Polymorphism with Typed Holes

Adam Chen[1], Thomas Porter[2], and Cyrus Omar[3]

[1] Stevens Institute of Technology, achen19@stevens.edu
[2] Unaffiliated, thomasporter2019@gmail.com
[3] University of Michigan, 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 or wrap missing or 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, pointing out 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. By extending forthcoming work on mark (i.e. error hole) insertion in Hazel, we develop a system in which the programmer may omit explicit type applications, and the editor (rather than downstream tools like the typechecker or compiler as is traditional) implicitly inserts and fills them, allowing the user to see and override these actions as needed.

Submission Remarks. This is a draft paper submission. The work to be completed and revised for formality are marked as conjectures. This is a student paper; Adam Chen and Thomas Porter are students and are the main authors.

1 Introduction & Background

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 [25], 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.

The Hazel programming language and environment [18] seeks to solve this problem of reasoning about incomplete programs by define a formal semantics for expressions that can contain *holes* in expressions, patterns, and types (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; 19], and Hazel is unique in assigning rich static and dynamic meaning to every editor state [19].

Hazel is based on a series of core calculi – Hazelnut and the marked lambda calculus, which define a static semantics for programs with expression and type holes, and Hazelnut Live, which defines a dynamic semantics for programs with expression and type holes. These calculi combine and extend ideas from contextual modal type theory [16] and the gradually typed lambda calculus [21].

The problem that motivates this work is that Hazelnut Live did not consider abstraction over types, which is key to practical typed functional programming. Sec. 3 presents *Polymorphic Hazelnut Live*, a polymorphic extension of Hazelnut Live [18] and Sec. 4 establishes that the key metatheoretic properties of Hazelnut Live, suitably modified to account for type variables, are conserved, including type safety (in the presence of holes). We also describe our mechanization of these metatheoretic properties in Agda, and discuss the implementation into the Hazel programming environment. Sec. 5 discusses further important metatheoretic properties that were not previously considered for Hazelnut 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 strict parametricity that we conjecture should hold for our system. In practice, it is cumbersome to explicitly apply type abstractions and most major functional languages support implicitly inferred type applications. Sec. 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

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; 22]. 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 is exemplified in the success of TypeScript [3]. Siek et al. [22] later formalize some properties that hold of the gradually-typed lambda calculus that make working in a gradually-typed system intuitive for programmers. These properties collectively are known as the gradual guarantee, and are the gold standard properties that are desirable for extensions of the gradually-typed lambda calculus.

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. Intuitively, 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 causes some polymorphic functions to encounter cast error on all types when any instantiation would error. (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 type. Gradual System F 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. [26], 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

The Hazelnut live system [18], in the tradition of gradually typed systems, presents a user-facing gradually typed calculus that elaborates (i.e. compiles) into a cast calculus. We extend both calculi from Hazelnut live with type variables, universal types, polymorphic abstraction, and type application.

$$\begin{array}{rrcl}
\textsf{Type} & \tau & ::= & \ldots \mid \alpha \mid \forall\alpha.\ \tau \\
\textsf{UExp} & e & ::= & \ldots \mid \Lambda\alpha.\ e \mid e\ [e] \\
\textsf{IHExp} & d & ::= & \ldots \mid \Lambda\alpha.\ d \mid d\ [d]
\end{array}$$

Fig. 1: Syntax extension of polymorphic Hazel.

With the introduction of the new type form, we define a corresponding matching judgement to allow for expanding the gradual type into a forall form. Note that the gradual type is identified with the type hole and is denoted ?.

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

$$\textsc{MFHole}\ \frac{}{?\blacktriangleright_{\forall}\forall\alpha.\ ?} \qquad \textsc{MFForall}\ \frac{}{\forall\alpha.\ \tau\blacktriangleright_{\forall}\forall\alpha.\ \tau}$$

Fig. 2: Matched forall types.

3.1 Gradually Typed Calculus

We extend the bidirectional typing and elaboration rules as shown in Fig. 3 and Fig. 4. We augment typing judgements with type variable contexts Σ, which are sets of type variables in scope. Notably, type functions may be typed both analytically and synthetically. Annotated term functions synthesize types, while unannotated term functions only analyze against types. Type functions may be thought of as having properties of both annotated and unannotated functions, and therefore admit both analytic and synthetic rules. This is a deviation from Dunfield and Krishnaswami [6]'s "bidirectional recipe", as we only have a single introduction form for type functions in the type assignment rules. We include both rules because doing so improves the power of the system (for example, being able to synthesize the type of the polymorphic identity function $\Lambda\alpha.\ \lambda x : \alpha.\ x$), without compromising on its properties (refer to Sec. 4)[4].

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

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

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

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

Fig. 3: Bidirectional typing rules.

[4] These rules match the polymorphic rules in [27], however we show a different set of properties, such as elaborability, type safety, etc. The previous work also does not deal with the elaborated internal expressions from these forms.

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

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

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

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

ATLam
$$\frac{e \text{ is not a hole} \qquad \tau_1 \blacktriangleright_{\forall} \forall\alpha.\, \tau_2 \qquad \Sigma, \alpha; \Gamma \vdash e \Leftarrow \tau_2 \rightsquigarrow d : \tau_3 \dashv \Delta}{\Sigma; \Gamma \vdash \Lambda\alpha.\, e \Leftarrow \tau_1 \rightsquigarrow \Lambda\alpha.\, d : \forall\alpha.\, \tau_4 \dashv \Delta}$$

Fig. 4: Elaboration rules from external expressions to internal expressions.

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

WFBase
$$\frac{}{\Sigma \vdash b}$$

WFHole
$$\frac{}{\Sigma \vdash ?}$$

WFVar
$$\frac{\alpha \in \Sigma}{\Sigma \vdash \alpha}$$

WFArr
$$\frac{\Sigma \vdash \tau_1 \qquad \Sigma \vdash \tau_2}{\Sigma \vdash \tau_1 \to \tau_2}$$

WFForall
$$\frac{\Sigma, \alpha \vdash \tau}{\Sigma \vdash \forall\alpha.\, \tau}$$

Fig. 5: Well-formedness rules.

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

3.2 Elaboration & Cast Calculus

We extend the typed elaboration rules. The added rules are shown in Fig. 4. Type function applications are elaborated in a way analogous to function applications. The function is cast to the output of the matched type judgement to accommodate the gradual type, and the function is analyzed against this type. We can omit any sort of type consistency check since the language syntax guarantees that only types appear as the arguments to type applications.

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

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

Fig. 6: Type assignment for internal expressions.

3.3 Dynamics & Final Forms

The universal type creates a new ground type case (Fig. 7). The ground type judgment is used to simplify the range of final forms, presented in Fig. 8. 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, and further, indet forms cannot occur in the absence of expression holes. This is formalized in Sec. 4 and extends the analogous results in [18].

$\boxed{\tau \text{ ground}}$ τ is a ground type $\qquad\qquad$ $\boxed{\tau \blacktriangleright_{\text{ground}} \tau'}$ τ has matched ground type τ'

$$\text{GForall}\quad \frac{}{\forall\alpha.\ ?\ \text{ground}} \qquad\qquad \text{MGForall}\quad \frac{\forall\alpha.\, \tau \neq \forall\alpha.\ ?}{\forall\alpha.\, \tau \blacktriangleright_{\text{ground}} \forall\alpha.\ ?}$$

Fig. 7: Ground and matched ground rules. The matched ground judgment is used in the ITGround and ITExpand instruction transitions, which are not presented here as they are not directly modified.

$\boxed{d \text{ val}}$ d is a value

$$\textsc{VTLam}\quad \frac{}{\Lambda\alpha.\, d \text{ val}}$$

$\boxed{d \text{ boxedval}}$ d is a boxed value

$$\textsc{VTLam}\quad \frac{\forall\alpha_1.\, \tau_1 \neq \forall\alpha_2.\, \tau_2 \qquad d \text{ boxedval}}{d\langle\forall\alpha_1.\, \tau_1 \Rightarrow \forall\alpha_2.\, \tau_2\rangle \text{ boxedval}}$$

$\boxed{d \text{ indet}}$ d is indeterminate

$$\textsc{ITAp}\quad \frac{d \neq d'\langle\forall\alpha_1.\, \tau_1 \Rightarrow \forall\alpha_2.\, \tau_2\rangle \qquad d \text{ indet}}{d\ [\tau] \text{ indet}}$$

$$\textsc{ICastForall}\quad \frac{\forall\alpha_1.\, \tau_1 \neq \forall\alpha_2.\, \tau_2 \qquad d \text{ indet}}{d\langle\forall\alpha_1.\, \tau_1 \Rightarrow \forall\alpha_2.\, \tau_2\rangle \text{ indet}}$$

Fig. 8: Final form rules.

Operational semantics are given in the approach of contextual semantics (TODO: cite for what this is). Evaluation contexts must be extended to allow for the new syntactic forms. All that is required is to extend evaluation contexts into type function application, and this is shown in Fig. 9.

Finally, we add new transition rules. Type functions applied to type are evaluated in System F style of type substitution. 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 Sec. 5. 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. Such a rule may be written analogously to the rule for term functions, albeit without any sort of consistency constraint on the type, which is syntactically guaranteed to be valid. The instruction transitions are shown in Fig. 10. 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.

$$\text{EvalCtx } \mathcal{E} ::= \ldots \mid \mathcal{E}\ [\tau]$$

$\boxed{d = \mathcal{E}\{d'\}}$ d is obtained by placing d' at the mark in $\mathcal{E}$.

$$\textsc{FHTAp}\quad \frac{d = \mathcal{E}\{d'\}}{d\ [\tau] = \mathcal{E}\ [\tau]\{d'\}}$$

Fig. 9: Changes to evaluation contexts.

$\boxed{d \longrightarrow d'}$ d takes an instruction transition to d'

$$\textsc{ITTLam}\quad \frac{}{(\Lambda\alpha.\ d)\ [\tau] \longrightarrow [\tau/\alpha]d} \qquad \textsc{ITTApCast}\quad \frac{}{d\langle \forall\alpha_1.\ \tau_1 \Rightarrow \forall\alpha_2.\ \tau_2\rangle\ [\tau] \longrightarrow d\ [\tau]\langle [\tau/\alpha_1]\tau_1 \Rightarrow [\tau/\alpha_2]\tau_2\rangle}$$

Fig. 10: Instruction transitions.

4 Type Safety

4.1 Type Safety

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 1. *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 exists an* IHExp d *such that* $\Sigma;\Gamma \vdash e \Rightarrow \tau \rightsquigarrow d \dashv \Delta$.
  2. *If* $\Sigma;\Gamma \vdash e \Leftarrow \tau$ *then there exists an* IHExp d *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 [sufficient in order to what?] to reason solely about the cast calculus. We prove that the system with the instruction transitions defined in Fig. 10 is type safe:

Theorem 2. *The system presented in Sec. 3.2 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 say 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 3. *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. These properties along with determinism (which is guaranteed by our contextual semantics) establish strong normalization of complete programs, with the val predicate defining normal forms.

Note that while complete elaboration does not directly prove that all casts in elaborated complete programs will be identity casts, since any cast between types that are not the gradual type (which is not present due to completeness) is indet, and the system is strongly normalizing to val, it cannot be the case that we evaluate casts between different types.

4.2 Agda Mechanization

The system and a proof of all of the properties[5] in this section are mechanized in Agda. The code is available at https://github.com/hazelgrove/hazelnut-polymorphism-agda.

In the mechanization we are forced to deal with some details about type variable binding names and alpha equivalence that we may gloss over in the formalization. We make the simplifying assumption that binding names are unique, so that we do not have to deal with capture-avoiding substitution. However, even when doing so, we found some cases where the rules must explicitly allow for alpha variation to make sense, and thus some results, like progress, are up to alpha equivalence.

Consider the following example:

$$d\langle \forall \alpha_1.\, \alpha_1 \Rightarrow \forall \alpha_2.\, \alpha_2 \rangle$$

It would not make sense for this example to be indet – the two types represented are the same under alpha equivalence, while being unequal in syntax! But should this

[5] As of writing of this draft, the proofs of preservation and complete preservation are incomplete for some details on substitution typing. As we omit fill-and-resume and thus substitution typing from our results, we will likely edit this out of the Agda prior to publication.

step to d, the type changes from $\forall\alpha_2.\ \tau_2$ to $\forall\alpha_1.\ \tau_1$, and so preservation is up to alpha. Furthermore, this term can arise from a function application with unique binding names, so simple assumptions about binding names are insufficient.

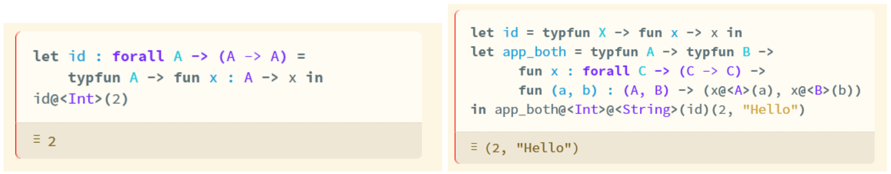
Specifically, the rules that must allow for alpha variation, even assuming binder names are advantageous, are:

- TAAp: In the typing term function application, the argument's type may vary up to alpha.
- TACast: When typing a type cast, the type being cast from may vary up to alpha (this is implied by type consistency, which is itself up to alpha).
- TAFailedCast: When typing a failing type cast, the type being cast from may vary up to alpha.
- ICastID: As in the motivating example above, when transitioning an identity cast, the type may vary up to alpha.
- ICastSucceed: Similar to the previous case, but with a cast to ? in-between.

4.3 Implementation

We have fully implemented the polymorphic system into the Hazel programming environment. 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. The implementation is done in ReasonML and uses js_of_ocaml to compile to a website. The Hazel project is described, with a link to the source code, at https://hazel.org.

The Hazel user interface is shown in Fig. 11. The user-facing gradually typed calculus is input via a gradual structure editor that uses obligations (see [15]); for example, inserting a `typfun` creates an obligation for a `->` and inserts the appropriate expression hole. Type function application is denoted with `@< >`.



```
let id : forall A -> (A -> A) =
    typfun A -> fun x : A -> x in
id@<Int>(2)
```

```
≡ 2
```

```
let id = typfun X -> fun x -> x in
let app_both = typfun A -> typfun B ->
      fun x : forall C -> (C -> C) ->
      fun (a, b) : (A, B) -> (x@<A>(a), x@<B>(b))
in app_both@<Int>@<String>(id)(2, "Hello")
```

```
≡ (2, "Hello")
```

Fig. 11: Screenshots of the current Hazel UI showing off polymorphic functions, including the polymorphic identity and a rank-2 polymorphic function.

[5] As of the writing of this draft, the code is still in a pull request, but it is soon-to-be merged. A live demo of the implemented system is available at https://hazel.org/build/poly-adt-after2/.

5 Metatheoretic Properties

5.1 Parametricity

As mentioned before, previous works imposed additional restrictions on the system 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 [10] is thus:

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

Noting that

$$\begin{aligned} &f\ [\mathsf{num}]\ 1 &&\mapsto^* 1 \\ &f\ [\mathsf{bool}]\ \mathsf{true} &&\mapsto^* \mathsf{true}\langle \mathsf{num} \Rightarrow \llparenthesis\rrparenthesis \not\Rightarrow \mathsf{bool}\rangle \end{aligned}$$

where 1 $\mathsf{boxedval}$, and $\mathsf{true}\langle \mathsf{num} \Rightarrow \llparenthesis\rrparenthesis \not\Rightarrow \mathsf{bool}\rangle$ indet. Indeterminate forms correspond to blame / errors in other calculi. By using type bindings, it would become the case that:

$$f\ [\mathsf{num}]\ 1 \mapsto^* 1\langle \mathsf{num} \Rightarrow \llparenthesis\rrparenthesis \not\Rightarrow \alpha\rangle$$

We argue that creating errors from otherwise sensibly executable programs is against the spirit of gradual typing, and we would like to avoid doing so. Igarashi et al. [10] 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 out 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 [12]) only enforces parametricity by introducing unnecessary error states, we instead focus on weakening parametricity. We follow the construction of parametricity presented in [5]. The necessary definition and lemma are reproduced here, adapted to our setting:

Definition 1. $=_0$ *is a relation between terms taken to be the least congruence such that for any* d, τ, *and* τ' *we have that* $\lambda x : \tau.\ d =_0 \lambda x : \tau'.\ d$ *and* $d\ [\tau] =_0 d\ [\tau']$.

Proposition 1 (Parametricity Lemma). *If* $d_1 =_0 d_2$ *then:*

1. *If* $d_1 \mapsto^* v_1$ *and* v_1 $\mathsf{boxedval}$ *then there exists* v_2 *such that* $d_2 \mapsto^* v_2$ *and* v_2 $\mathsf{boxedval}$ *and* $v_1 =_0 v_2$.
2. *If* $d_1 \mapsto^* v_1$ *and* $d_2 \mapsto^* v_2$ *and* v_1 $\mathsf{boxedval}$ *and* v_2 final *then* v_2 $\mathsf{boxedval}$ *and* $v_1 =_0 v_2$

In this draft paper, we give two conjectures based off the form of the parametricity lemma that express the metatheoretic properties we believe to hold of our system:

Conjecture 1. If e $\mathsf{complete}$ and $\Sigma; \Gamma \vdash e \Rightarrow \tau \rightsquigarrow d_1 \dashv \Delta$ and $d_1 =_0 d_2$ then:

1. If $d_1 \mapsto^* v_1$ and v_1 boxedval then there exists v_2 such that $d_2 \mapsto^* v_2$ and v_2 boxedval and $v_1 =_0 v_2$.
2. If $d_1 \mapsto^* v_1$ and $d_2 \mapsto^* v_2$ and v_1 boxedval and v_2 final then v_2 boxedval and $v_1 =_0 v_2$

That is, if d is the result of typed elaboration of a complete program, then d is parametric. This asserts that if we do not deal with graduality then we are able to ensure full parametricity. We believe this to be true since typeable complete programs contain only identity casts and thus may be embedded in System F, which is parametric.

Conjecture 2. If $d_1 =_0 d_2$ then:

1. If $d_1 \mapsto^* v_1$ and v_1 boxedval then there exists v_2 such that $d_2 \mapsto^* v_2$ and either (v_2 boxedval and $v_1 =_0 v_2$) or v_2 indet.
2. If $d_1 \mapsto^* v_1$ and $d_2 \mapsto^* v_2$ and v_1 boxedval and v_2 boxedval then $v_1 =_0 v_2$

In other words, all terms exhibit parametricity *up to successful termination*. Note the added disjunct in item 1. This conjecture means that, should we know the program successfully terminates, then we recover parametricity. Thus the only violations we get of parametricity are run-time cast errors, and thus solely from not erroring otherwise successfully executing programs.

5.2 Gradual Guarantee

Suppose we are given a notion of precision between external terms and types $\sqsubseteq$. We take the statement of the gradual guarantee from Siek et al. [22] and adapt it to our notation:

Definition 2. *Suppose* $\emptyset;\emptyset \vdash e \Rightarrow \tau$ *and* $\emptyset;\emptyset \vdash e \Rightarrow \tau \rightsquigarrow d \dashv \Delta$. *We say* $e \Downarrow v$ *when* $d \mapsto^* v$ *and* v boxedval. *We say* $e \Uparrow$ *when there does not exist a* v *such that* $d \mapsto^* v$ *and* v final *(in other words,* d *diverges).*

Proposition 2 (The Gradual Guarantee). *Suppose* $e \sqsubseteq e'$ *and* $\emptyset;\emptyset \vdash e \Rightarrow \tau$.

1. *There exists a* τ' *such that* $\emptyset;\emptyset \vdash e' \Rightarrow \tau'$ *with* $\tau \sqsubseteq \tau'$.
2. *If* $e \Downarrow v$ *with* v boxedval *then* $e' \Downarrow v'$ *with* v' boxedval *and* $v \sqsubseteq v'$.
   If $e \Uparrow$ *then* $e' \Uparrow$.
3. *If* $e' \Downarrow v'$ *with* v' boxedval *then* $e \Downarrow v$ *such that either* v boxedval *and* $v \sqsubseteq v'$ *or* v indet.
   If $e' \Uparrow$ *then* $e \Uparrow$ *or* $e \Downarrow v$ *with* v indet.

Of course, what the guarantee establishes depends on the definition of $\sqsubseteq$. Recall that our setting is that of a live programming environment. Thus we would like become more precise to coincide with the intuitive idea of the programmer filling in type holes. Thus we do not want a definition similar to System F_G [10]'s, but rather one closer to GSF [12]'s. Such a definition is provided in Fig. 12[6].

[6] Note that in the PForall rule, for convenience, we assume the binding names on the left and right match. For a more general treatment of binding names, we would introduce binding environments.

$$\textsc{PBase}\quad \frac{}{b \sqsubseteq b} \qquad \textsc{PHole}\quad \frac{}{\tau \sqsubseteq ?} \qquad \textsc{PTVar}\quad \frac{}{\alpha \sqsubseteq \alpha}$$

$$\textsc{PArr}\quad \frac{\tau_1 \sqsubseteq \tau_1' \qquad \tau_2 \sqsubseteq \tau_2'}{\tau_1 \to \tau_2 \sqsubseteq \tau_1' \to \tau_2'} \qquad \textsc{PForall}\quad \frac{\tau \sqsubseteq \tau'}{\forall \alpha.\, \tau \sqsubseteq \forall \alpha.\, \tau'}$$

Fig. 12: Precision between types

Conjecture 3. The gradual guarantee holds of Polymorphic Hazelnut Live with the provided precision relation.

Intuitively, if the previous result about parametricity holds, then the only cases in which programs do not behave like their type erased version are when they evaluate to indet, i.e. a cast failure. Due to the canonical forms, casts fail when inconsistent types are cast to each other. But since the gradual type is consistent with any type, replacing types with less precise types can only create cases where previously failing casts succeed, rather than making previously successful casts fail. Conversely, if we replace types with more precise types (intuitively, replacing type holes with concrete types), we can only turn previously successful casts, either a composition of casts through the gradual type or the identity cast on the gradual type, into now failing casts. These correspond to items 2 and 3 of the gradual guarantee, sometimes referred to as the dynamic gradual guarantee. In short, since we never introduce types from outside the program syntax, we have good reason to believe that the gradual guarantee holds. Furthermore, our system is very similar to GSF, which is known to satisfy the gradual guarantee.

6 Towards Implicit Polymorphism

For practical programming purposes, it is burdensome to write explicit type applications for each use of a polymorphic term. Therefore usable 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 [7], which we could have chosen to gradualize in the same manner as we gradualized explicit System F above.

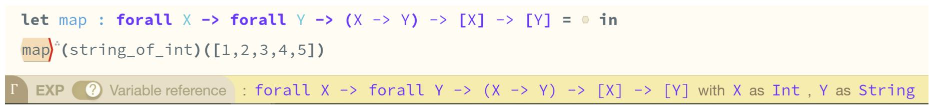
Instead we propose to take advantage of the structured editing capabilities of Hazel. The widespread practice of implicit language features represents a compromise between the interests of the language user and the language developer. The user benefits by typing and seeing less code, and by achieving 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.

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 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 blue type applications into the program below, supposing that $f : A \to B$ and $l : A$ list.

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

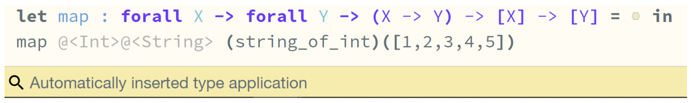
These type applications could be folded by default to avoid cluttering the screen with useless information. Despite this diminution of the type application forms, the proposed strategy is distinct from usual implicit schemes because the persisted program will retain the inferred type applications, and because the user will be able to see and edit the type arguments if needed. Figures 13 - 16 display mock ups of various editor states in a hypothetical version of Hazel with implicit polymorphism.



```
let map : forall X -> forall Y -> (X -> Y) -> [X] -> [Y] = in
map(string_of_int)([1,2,3,4,5])
```

Γ EXP Variable reference : forall X -> forall Y -> (X -> Y) -> [X] -> [Y] with X as Int , Y as String

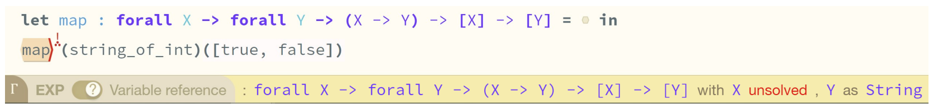
Fig. 13: Hypothetical behavior: when a type application insertion succeeds, the type arguments are listed along with the type of the polymorphic term. An ellipsis mark indicates folded code and provides a way to examine it.



```
let map : forall X -> forall Y -> (X -> Y) -> [X] -> [Y] = in
map @<Int>@<String> (string_of_int)([1,2,3,4,5])
```

Automatically inserted type application

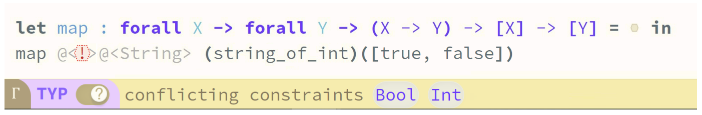
Fig. 14: Hypothetical behavior: when the ellipsis mark is selected, it expands to reveal the explicit type applications and arguments that have been inserted by the editor. If this code is edited by the user, it becomes fully explicit and is colored accordingly.



```
let map : forall X -> forall Y -> (X -> Y) -> [X] -> [Y] = in
map(string_of_int)([true, false])
```

Γ EXP Variable reference : forall X -> forall Y -> (X -> Y) -> [X] -> [Y] with X unsolved , Y as String

Fig. 15: Hypothetical behavior: when a type application insertion fails, the conflicted type arguments are indicated. The ellipsis mark signals an error.



```
let map : forall X -> forall Y -> (X -> Y) -> [X] -> [Y] = ⬡ in
map @<!>@<String> (string_of_int)([true, false])
```

```
conflicting constraints Bool Int
```

Fig. 16: 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.

6.1 Mark Insertion

A forthcoming paper ([27]) describes the “mark insertion" component of the Hazel architecture. Hazel programs begin as unmarked expressions e, corresponding exactly to the external expressions in [18] except that they do not contain nonempty holes. Next comes a bidirectionally typed mark insertion phase, with judgement forms $\Sigma; \Gamma \vdash e \looparrowright \check{e} \Rightarrow \tau$ and $\Sigma; \Gamma \vdash e \looparrowright \check{e} \Leftarrow \tau$. 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 first described in [18] and extended to include polymorphism in Sec. 3.

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

$$\ldots \qquad \frac{\Sigma, \alpha; \Gamma \vdash e \looparrowright \check{e} \Rightarrow \check{\tau}}{\Sigma; \Gamma \vdash \Lambda\alpha.\, e \looparrowright \Lambda\alpha.\, \check{e} \Rightarrow \forall\alpha.\, \check{\tau}}\ \text{MKSTypeLam} \qquad \frac{\Sigma; \Gamma \vdash e \looparrowright \check{e} \Rightarrow \check{\tau} \qquad \Sigma \vdash \tau_2 \looparrowright \check{\tau}_2 \qquad \check{\tau} \blacktriangleright_{\forall} \forall\alpha.\, \check{\tau}_1}{\Sigma; \Gamma \vdash e\,[\tau_2] \looparrowright \check{e}\,[\check{\tau}_2] \Rightarrow \check{\tau}_1[\check{\tau}_2/\alpha]}\ \text{MKSTypeAp1}$$

$$\frac{\Sigma; \Gamma \vdash e \looparrowright \check{e} \Rightarrow \check{\tau} \qquad \Sigma \vdash \tau_2 \looparrowright \check{\tau}_2 \qquad \check{\tau} \blacktriangleright_{\not\forall}}{\Sigma; \Gamma \vdash e\,[\tau_2] \looparrowright (\!|\check{e}|\!)^{\Rightarrow}_{\blacktriangleright_{\not\forall}}\,[\check{\tau}_2] \Rightarrow ?}\ \text{MKSTypeAp2}$$

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

$$\ldots \qquad \frac{\check{\tau} \blacktriangleright_{\forall} \forall\alpha.\, \check{\tau}' \qquad \Sigma, \alpha; \Gamma \vdash e \looparrowright \check{e} \Leftarrow \check{\tau}'}{\Sigma; \Gamma \vdash \Lambda\alpha.\, e \looparrowright \Lambda\alpha.\, \check{e} \Leftarrow \check{\tau}}\ \text{MKATypeLam1} \qquad \frac{\check{\tau} \blacktriangleright_{\not\forall} \qquad \Sigma, \alpha; \Gamma \vdash e \looparrowright \check{e} \Leftarrow ?}{\Sigma; \Gamma \vdash \Lambda\alpha.\, e \looparrowright (\!|\Lambda\alpha.\, \check{e}|\!)^{\Leftarrow}_{\blacktriangleright_{\not\forall}} \Leftarrow \check{\tau}}\ \text{MKATypeLam2}$$

Fig. 17: Polymoprhic Mark Insertion Rules

Fig. 17 contains the mark insertion rules for the polymorphic fragment, which can also be found in the appendix of [27]. 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.

The mark insertion judgement satisfies desirable metatheoretic properties. These properties include totality and unicity, which mean that the insertion operation is a total function on unmarked expressions. Mark insertion also satisfies the property that it only generates well-typed terms, and that erasing marks from the marked term recovers the original term. The mark insertion process is also guaranteed to not affect terms that already type check, and to insert at least one mark into terms that do not type check.

6.2 Type Application Insertion

We enrich the mark insertion phase so that it also inserts type applications with a type hole as the argument, in locations deemed necessary by the bidirectional typing flow. 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 marked nonempty hole. The new rules for these insertions are described in Fig. 18. These rules replace the previous rules for applications, projections, and subsumption presented in [27]. The insertion of type applications could have been presented equivalently as a phase that follows mark insertion, and only transforms programs at the location of certain marks. However, the synthesized and analyzed types would be the same in both phases, so presenting them as separate would imply a large degree of redundant computation.

The insertion rules for polymorphism are derived from the mark insertion rules in [27]. The rules that have been updated are those with a premise of the form $\tau \blacktriangleright_{\rightarrow} \tau_1 \rightarrow \tau_2$, $\tau \blacktriangleright_{\not\rightarrow}$, $\tau \blacktriangleright_{\times} \tau_1 \times \tau_2$, $\tau \blacktriangleright_{\not\times}$, $\tau \sim \tau_1$, or $\tau \nsim \tau_1$, where τ and none of the other types involved is 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 we have omitted such a discussion for brevity and because 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 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 updated properties state

$\boxed{\forall^{\Box}(\tau_1) = \tau_2}$ Prenex erasure operation

PEForall
$$\frac{}{\forall^{\Box}(\forall\alpha.\ \tau) = \forall^{\Box}(\tau[?/\alpha])}$$

PEUnknown
$$\frac{}{\forall^{\Box}(?) = ?}$$

PENotMatched
$$\frac{\tau \blacktriangleright_{\not\forall}}{\forall^{\Box}(\tau) = \tau}$$

$\boxed{\Sigma; \Gamma \vdash e \looparrowright \check{e} \Rightarrow \tau}$ Synthetic marking judgement

InsertSAp1
$$\frac{\Sigma; \Gamma \vdash e_1 \looparrowright \check{e}_1 \Rightarrow \tau_1 \qquad \tau_1 \blacktriangleright_{\rightarrow} \tau_2 \rightarrow \tau_3 \qquad \Sigma; \Gamma \vdash e_2 \looparrowright \check{e}_2 \Leftarrow \tau_2}{\Sigma; \Gamma \vdash e_1\ e_2 \looparrowright \check{e}_1\ \check{e}_2 \Rightarrow \tau_3}$$

InsertSAp2
$$\frac{\Sigma; \Gamma \vdash e_1 \looparrowright \check{e}_1 \Rightarrow \tau \qquad \tau \blacktriangleright_{\not\rightarrow} \qquad \forall^{\Box}(\tau) \blacktriangleright_{\rightarrow} \tau_3 \rightarrow \tau_4 \qquad \Sigma; \Gamma \vdash e_1\ [?]\ e_2 \looparrowright \check{e}_3 \Rightarrow \tau_2}{\Sigma; \Gamma \vdash e_1\ e_2 \looparrowright \check{e}_3 \Rightarrow \tau_2}$$

InsertSAp3
$$\frac{\Sigma; \Gamma \vdash e_1 \looparrowright \check{e}_1 \Rightarrow \tau \qquad \forall^{\Box}(\tau) \blacktriangleright_{\not\rightarrow} \qquad \Sigma; \Gamma \vdash e_2 \looparrowright \check{e}_2 \Leftarrow ?}{\Sigma; \Gamma \vdash e_1\ e_2 \looparrowright \llparenthesis \check{e}_1 \rrparenthesis^{\Rightarrow}_{\blacktriangleright_{\not\rightarrow}}\ \check{e}_2 \Rightarrow ?}$$

InsertSProj1
$$\frac{\Sigma; \Gamma \vdash e \looparrowright \check{e} \Rightarrow \tau \qquad \tau \blacktriangleright_{\times} \tau_1 \times \tau_2}{\Sigma; \Gamma \vdash \pi_1 e \looparrowright \pi_1 \check{e} \Rightarrow \tau_1}$$

InsertSProj2
$$\frac{\Sigma; \Gamma \vdash e \looparrowright \check{e}_1 \Rightarrow \tau \qquad \tau \blacktriangleright_{\not\times} \qquad \forall^{\Box}(\tau) \blacktriangleright_{\times} \tau_3 \times \tau_4 \qquad \Sigma; \Gamma \vdash \pi_1(e\ [?]) \looparrowright \check{e}_2 \Rightarrow \tau_2}{\Sigma; \Gamma \vdash \pi_1 e \looparrowright \check{e}_2 \Rightarrow \tau_2}$$

InsertSProj3
$$\frac{\Sigma; \Gamma \vdash e \looparrowright \check{e} \Rightarrow \tau \qquad \forall^{\Box}(\tau) \blacktriangleright_{\not\times}}{\Sigma; \Gamma \vdash \pi_1 e \looparrowright \pi_1 \llparenthesis \check{e} \rrparenthesis^{\Rightarrow}_{\blacktriangleright_{\not\times}} \Rightarrow ?}$$

$\boxed{\Sigma; \Gamma \vdash e \looparrowright \check{e} \Leftarrow \tau}$ Analytic marking judgement

InsertASubsume1
$$\frac{\Sigma; \Gamma \vdash e \looparrowright \check{e} \Rightarrow \tau_1 \qquad \tau_1 \sim \tau_2 \qquad e \text{ subsumable}}{\Sigma; \Gamma \vdash e \looparrowright \check{e} \Leftarrow \tau_2}$$

InsertASubsume2
$$\frac{\begin{array}{c} \Sigma; \Gamma \vdash e \looparrowright \check{e}_1 \Rightarrow \tau_1 \\ \tau_1 \nsim \tau_2 \qquad \forall^{\Box}(\tau_1) \sim \tau_2 \qquad \Sigma; \Gamma \vdash e\ [?] \looparrowright \check{e}_2 \Leftarrow \tau_2 \qquad e \text{ subsumable} \end{array}}{\Sigma; \Gamma \vdash e \looparrowright \check{e}_2 \Leftarrow \tau_2}$$

InsertAInconsistentTypes
$$\frac{\Sigma; \Gamma \vdash e \looparrowright \check{e} \Rightarrow \tau_1 \qquad \tau_1 \nsim \tau_2 \qquad \forall^{\Box}(\tau_1) \nsim \tau_2 \qquad e \text{ subsumable}}{\Sigma; \Gamma \vdash e \looparrowright \llparenthesis \check{e} \rrparenthesis_{\nsim} \Leftarrow \tau_2}$$

Fig. 18: Implicit insertion.

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 [27], which uses constraints on type holes during the mark insertion phase to generate the possible fillings for the type holes that appear in the program. 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 [27] 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 uncomputable. We therefore cannot immediately implement the system of implicit polymorphism outlined in this section. 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.

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

7 Related and Future Work

Polymorphic Type Hole Inference. While it is impossible to completely solve the higher order unification problem of inferring type hole values in polymorphic Hazel, it may be possible to devise a satisfactory partial solution. Just as the general undecidability of type inference for System F does not preclude effective partial solutions based on heuristics, it seems plausible that an incomplete yet practical type hole inference mechanism may be developed to enable implicit polymorphic workflows in Hazel. In this case, the suggestions of the editor will not be complete in all cases, but will be sound and often helpful. We leave the development of such an extension to future work.

Fill and Resume. Hazelnut Live [18] 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. These substitutions were a part of the cast calculus, and their validity were 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 provided 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. [26], implicit polymorphism may force instantiations that cause error 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. The authors 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, one may populate the type $\forall\alpha.\ \alpha$ ref with the term $\Lambda\alpha.$ ref$(\llparenthesis\rrparenthesis)$, which is not possible otherwise. 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 [10].

Acknowledgements. 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.

Bibliography

[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, January 26-28, 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, July 28 - August 1, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8586, pp. 257–281. Springer (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 '99), Paris, France, September 27-29, 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. Ed.). Cambridge University Press (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., Ragan-Kelley, B., Pérez, F., Granger, B.E., Bussonnier, M., Frederic, J., Kelley, K., Hamrick, J.B., Grout, J., Corlay, S., Ivanov, P., Avila, D., Abdalla, S., Willing, C., Team, J.D.: 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, June 7-9, 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., Fitzgerald, S., Lewandowski, G., Murphy, L., Simon, B., Thomas, L., Zander, C.: 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, March 29 - April 6, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4962, pp. 350–364. Springer (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, October 3-6, 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, January 18-20, 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, April 9-11, 1974. Lecture Notes in Computer Science, vol. 19, pp. 408–423. Springer (1974). https://doi.org/10.1007/3-540-06859-7_148

[21] Siek, J., Taha, W.: Gradual typing for functional languages (01 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, May 3-6, 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.) Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings. Lecture Notes in Computer Science, vol. 9032, pp. 432–456. Springer (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, May 19, 2013. pp. 31–34. IEEE Computer Society (2013). https://doi.org/10.1109/LIVE.2013.6617346

[25] Wakeling, D.: Spreadsheet functional programming. J. Funct. Program. **17**(1), 131–143 (2007). https://doi.org/10.1017/S0956796806006186

[26] 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

[27] Zhao, E., Maroof, R., Dukkipati, A., Pan, Z., Omar, C.: Total type error localization and recovery with holes. Proc. ACM Program. Lang. **8**(POPL) (2024). https://doi.org/10.1145/3371114