Polymorphic Gradual Typing with Holes

Jaebyeog Kim Eijiro Sumii

In a live programming environment, we often work with incomplete programs that are syntactically ill-formed or have type errors [Omar+ 2017, 2019]. Since most languages do not provide semantics to such incomplete programs, they do not give an idea of how other "well written" parts of the programs work. To provide programmers continuous feedback about programs' typing and dynamic behavior, Omar et al. [2019] defined Hazelnut Live, a calculus that gives static and dynamic semantics to incomplete programs. They used *holes* $(\!|\ |\!)$ to represent missing parts of programs.

We extend this idea to polymorphic languages. Since typing of incomplete programs resembles gradual typing [Siek and Taha 2006], we adopt recent research [New+ 2020] on polymorphic gradual typing, extend its syntax with hole expressions, and do *not* stop a whole program by cast errors (which gradual typing does, while Hazelnut Live does not).

Extending polymorphic terms with holes calls for two major considerations. First, we address unspecified formal type parameters by assigning a fresh type variable X' for its *type* $\forall X'.A$, while introducing the *term* syntax $\Lambda(\!|\ |\!).M$ as it is, so as *not* to reduce incomplete programs like $(\Lambda(\!|\ |\!).M)[B]$. Second, we treat programs M containing unbound (term or type) variables by putting them inside holes $(\!|M|\!)$ and giving them different type environments.

With these ideas, we define a polymorphic calculus that gives static and dynamic semantics to incomplete programs.

1 Introduction

As discussed by Omar et al. [7], when we are in the middle of writing programs, we can not always have a complete program. Sometimes we need to write more code to make it syntactically right, or purposefully omit details to have a bigger structure first. Or maybe we just write something wrong like $2 + \mathsf{true}$. While we are amidst such editing, we still want to know behavior of our program even if it is partial and incomplete.

Most of the time, it is just the case that the compiler lets us know only which part of the program is wrong, or the runtime aborts as soon as it detects a dynamic error. This is because most programming languages do not assign meanings to incomplete programs. To attack this problem, Omar et al. [8] proposed a calculus, named Hazelnut, for functional programs where every editor state—even if intermediate—has some static meaning. They formalized users' edit actions as well as programs with *holes* that represent incomplete parts, and defined *static* semantics to the programs with holes.

Later, Omar et al. [7] also developed a calculus named Hazelnut Live. This calculus defines *dynamic* semantics for incomplete programs with holes, where evaluation continues arounds holes instead of stopping. It is closely related to gradual typing [11] for typing and evaluation of programs with type holes, where explicit casts are inserted before evaluation. Unlike gradual typing, however,

Jaebyeog Kim, Eijiro Sumii, 東北大学 大学院 情報科学研究科, Graduate School of Information Sciences, Tohoku University.

the evaluation does not stop even if a cast error occurs, and continues to other parts of the program.

We aim to extend this idea to other desirable features of programming languages, in particular parametric polymorphism. Since Hazelnut Live's semantics for handling types are based on gradual typing, we find a calculus that supports polymorphic gradual typing, and then extend it with static and dynamic semantics for hole expressions.

1.1 Polymorphic Gradual Typing with PolyG$^\nu$

Polymorphic gradual typing by itself is already challenging. Some previous studies [2] [4] [14] tried to extend gradual typing with explicit polymorphism like System F, but failed to meet criteria for parametric polymorphism and gradual typing [6].

For example, consider a function $f = (\Lambda X.\lambda x : X.(x ::?) :: \mathsf{Bool})$, where :: represents type ascription. Since we want ? to be consistent [11] with any other type, this term is well-typed and has type $\forall X.X \to \mathsf{Bool}$. Thus, we should able to apply this function to any type and a value of that type, and get a value of type Bool. Nonetheless, if we just substitute type variables as in System F, we find that f can accept only a value of the Bool type, as in the following examples (where $\rightsquigarrow$ represents elaboration into cast calculus and $\mapsto$ reduction):

$$
\begin{array}{ll}
 & f\ [\mathsf{Bool}]\ \mathsf{true} \\
\rightsquigarrow & (\Lambda X.\lambda x : X.\langle \mathsf{Bool} \Leftarrow ?\rangle\langle ? \Leftarrow X\rangle x)\ [\mathsf{Bool}]\ \mathsf{true} \\
\mapsto^* & \langle \mathsf{Bool} \Leftarrow ?\rangle\langle ? \Leftarrow \mathsf{Bool}\rangle \mathsf{true} \\
\mapsto^* & \mathsf{true} \\
 & f\ [\mathsf{Int}]\ 1 \\
\rightsquigarrow & (\Lambda X.\lambda x : X.\langle \mathsf{Bool} \Leftarrow ?\rangle\langle ? \Leftarrow X\rangle x)\ [\mathsf{Int}]\ 1 \\
\mapsto^* & \langle \mathsf{Bool} \Leftarrow ?\rangle\langle ? \Leftarrow \mathsf{Int}\rangle 1 \\
\mapsto^* & \mathit{error}
\end{array}
$$

This violates parametricity since behavior of the function differs depending on its type argument.

To ensure parametricity, Ahmed et al. [2] used sealing [5] [9] [13] in their polymorphic gradual language, later called λB [2]. When we apply a polymorphic function to a value, we seal the value with a fresh key. Then, the applied function can not inspect inside the sealed value. In λB, because they try to make a conservative extension of System F, the sealing and unsealing are implicit and are automatically inserted by type-directed coercion.

Since the insertion of sealing is type-directed, λB violates dynamic gradual guarantee [4]. The gradual guarantee is one of the "refined criteria" for gradually typed languages, presented by Siek et al. [12]. It captures the idea that replacing types with ? does not introduce more errors. λB does *not* satisfy (dynamic) gradual guarantee, as in:

$$
\begin{array}{ll}
 & ((\Lambda X.\lambda x : X.x :: X)\ \mathsf{Int}\ 1) + 1 \\
\mapsto^* & (\mathsf{unseal}_X(\mathsf{seal}_X 1)) + 1 \\
\mapsto^* & 2 \\
 & ((\Lambda X.\lambda x : X.x ::?)\ \mathsf{Int}\ 1) + 1 \\
\mapsto^* & (\mathsf{seal}_X 1) + 1 \\
\mapsto^* & \mathit{error}
\end{array}
$$

Other studies [4] [14] also have similar difficulties with gradual guarantee and parametricity. Ahmed et al. [6] pointed out these difficulties and avoided them by means of explicit sealing manually written by programmers. With this idea, they defined a polymorphic gradual language, PolyG$^\nu$.

Since we want our language to satisfy parametricity and dynamic gradual guarantee, we adopt PolyG$^\nu$ as our base language that we extend with typed holes.

1.2 Challenge of Incomplete Polymorphic Terms

Extending polymorphic calculus with holes brings two challenges about programs that are incomplete regarding polymorphic types: unspecified formal type parameters and unbound type variables.

Unspecified formal type parameters as in $\Lambda(\!|\,|\!).M$ are needed since they can *not* just be replaced with

fresh X' like $\Lambda X'.M$, since doing so breaks our intention that $\Lambda(\!|\!|).M$ is an incomplete program and cannot be applied, that is, its type application like $(\Lambda(\!|\!|).M)[\dots]$ cannot be reduced. We thus introduce $\Lambda(\!|\!|).M$ as distinct syntax, and give it a proper typing rule and *no* evaluation rule. (Similar considerations are required for type application in PolyG$^\nu$, which also binds type variables.) We also treat anonymous *term* abstraction $\lambda(\!|\!|).M$, which is not mentioned in Hazelnut [8] or Hazelnut Live [7] but is present in their implementation Hazel [1].

Unbound *term* variables are actually implemented in Hazel [1], but are not mentioned in either Hazelnut [8] or Hazelnut Live [7]. In Hazelnut Live, since a term inside a hole has to be well typed under the same type environment, it cannot contain an unbound (term) variable. (In Hazelnut, unbound variables cannot even be written with the editor.) By contrast, we let the type environment in holes extend the outside environment, so as to allow unbound variables inside.

These extensions apply to both term and type parameters and variables, but would be useful in particular for types, which programmers are more likely to omit or forget.

1.3 Contributions

We thus define our new language PolyG$^\nu$H that is extended from PolyG$^\nu$ with hole expressions and expressions with unspecified formal parameters. We also define an internal language PolyC$^\nu$H, where actual reduction occurs, and elaboration from PolyG$^\nu$H to PolyC$^\nu$.

2 Preliminaries

2.1 Hazelnut Live

Hazelnut Live [7] is a core calculus of Hazel [1] that evaluates incomplete expressions. The syntax of Hazelnut Live is shown in Figure 1. Terms are divided into two categories: external expressions e, and internal expressions d.

External expressions correspond to user input and are elaborated to internal expressions before evaluation. b represents base types such as Int and Bool, and c represents constants that have base types. Other than orthodox terms for functional programming languages, they added two main inventions to the external language: type holes $(\!|\!|)$, which coincide with dynamic types ? in gradual typing [11], and hole expressions, which represent the parts where terms of correct types are expected but missing. Hole expressions consist of empty holes $(\!|\!|)^u$ and non-empty holes $(\!|e|\!)^u$. The hole name u is given to each hole expression as an identifier. Empty and non-empty holes respectively represent missing expressions and expressions that have type inconsistency.

Figure 2 shows typing rules for the external language. The type system is defined in a bidirectional style [10]. The type synthesis judgement $\Gamma \vdash e \Rightarrow \tau$ means under typing context Γ, external expression e synthesizes type τ. The type analysis judgement $\Gamma \vdash e \Rightarrow \tau$ checks the expression e against τ. Hole expressions have the hole type $(\!|\!|)$ but non-empty holes require expressions inside to be well-typed. Most of the typing rules follow gradual typing [11], where type consistency relation $\sim$ and matching function $\mathit{fun}(\tau)$ are used instead of type equivalence.

External expressions are elaborated into internal according to Figure 3, where explicit casts $\langle \tau \Rightarrow \tau \rangle$ are inserted based on the type consistency relation. As a result, type consistency is used only in the typing rule for explicit casts in Figure 2. Omar et al. [7] also introduced a new term called failed cast $d\langle \Rightarrow (\!|\!|) \not\Rightarrow T \rangle$, which represents a dynamic cast error rather than abortion, so as to continue evaluation of other parts. For failed casts, we require the two types G_1 and G_2 are ground, that is, consist of only base types and $(\!|\!|) \to (\!|\!|)$. In both elaboration

$$
\begin{array}{llll}
\text{HTyp} & \tau & ::= & b \mid \tau \to \tau \mid \text{⦇⦈} \\
\text{HExp} & e & ::= & c \mid x \mid \lambda x : \tau . e \mid \lambda x . e \mid e\ e \mid \text{⦇⦈}^u \mid \text{⦇}e\text{⦈}^u \mid e : \tau \\
\text{IHExp} & d & ::= & c \mid x \mid \lambda x : \tau . d \mid d\ d \mid \text{⦇⦈}^u_\sigma \mid \text{⦇}d\text{⦈}^u_\sigma \mid d\langle \tau \Rightarrow \tau \rangle \mid d\langle \tau \Rightarrow \text{⦇⦈} \not\Rightarrow \tau \rangle
\end{array}
$$

$$
d\langle \tau_1 \Rightarrow \tau_2 \Rightarrow \tau_3 \rangle \stackrel{\text{def}}{=} d\langle \tau_1 \Rightarrow \tau_2 \rangle \langle \tau_2 \Rightarrow \tau_3 \rangle
$$

Fig. 1 Hazelnut Live Syntax

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

$$
\frac{}{\Gamma \vdash c \Rightarrow b} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x \Rightarrow \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash e \Rightarrow \tau_2}{\Gamma \vdash \lambda x : \tau_1 . e \Rightarrow \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \quad \mathit{fun}(\tau_1) = \tau_2 \to \tau \quad \Gamma \vdash e_2 \Leftarrow \tau_2}{\Gamma \vdash e_1\ e_2 \Rightarrow \tau}
$$

$$
\frac{}{\Gamma \vdash \text{⦇⦈}^u \Rightarrow \text{⦇⦈}} \qquad \frac{\Gamma \vdash e \Rightarrow \tau}{\Gamma \vdash \text{⦇}e\text{⦈}^u \Rightarrow \text{⦇⦈}} \qquad \frac{\Gamma \vdash e \Leftarrow \tau}{\Gamma \vdash e : \tau \Rightarrow \tau}
$$

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

$$
\frac{\mathit{fun}(\tau) = \tau_1 \to \tau_2 \quad \Gamma, x : \tau_1 \vdash e \Leftarrow \tau_2}{\Gamma \vdash \lambda x . e \Leftarrow \tau} \qquad \frac{\Gamma \vdash e \Rightarrow \tau \quad \tau \sim \tau'}{\Gamma \vdash e \Leftarrow \tau'}
$$

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

$$
\frac{}{\text{⦇⦈} \sim \tau} \qquad \frac{}{\tau \sim \text{⦇⦈}} \qquad \frac{}{\tau \sim \tau} \qquad \frac{\tau_1 \sim \tau_1' \quad \tau_2 \sim \tau_2'}{\tau_1 \to \tau_2 \sim \tau_1' \to \tau_2'}
$$

$$
\mathit{fun}(\text{⦇⦈}) = \text{⦇⦈} \to \text{⦇⦈} \qquad \mathit{fun}(\tau_1 \to \tau_2) = \tau_1 \to \tau_2
$$

Fig. 2 Hazelnut Live Typing

and typing rules, hole context Δ is used to ensure the consistency of environments among holes with the same name u.

Since Hazelnut Live receives incomplete programs, well-typed expressions do not always evaluate to values but may become irreducible because of hole expressions and failed casts. To represent such irreducible states, *indets* (indeterminate terms) are defined, and *finals* (irreducible expressions) are defined as either values or indets, as in Figure 5. As in gradual typing, values with an irreducible cast are also treated as values (originally called "boxed values" in [7]).

Reduction rules (defined only for internal expressions) are shown in Figure 6. First, there is beta reduction. Casts between identical types are removed. Casts between function types break down when the function is applied to an argument. Casts from or to hole type are extended to go through matching ground types, which are uniquely determined. In the case of a projection (a cast from ? to a ground type) after an injection (a cast from a ground type to ⦇⦈), the cast is removed, or else reduced to a failed cast term, according to whether those (ground) types are the same or not.

2.2 PolyG$^\nu$

As we discussed in the introduction, PolyG$^\nu$ [6] is a polymorphic gradual language with explicit sealing. Figure 7 is the syntax of PolyG$^\nu$. They write

$\boxed{\Gamma \vdash e \Rightarrow \tau \rightsquigarrow d \dashv \Delta}$ e synthesizes type τ and elaborates to d

$$\frac{}{\Gamma \vdash c \Rightarrow b \rightsquigarrow c \dashv \emptyset} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x \Rightarrow \tau \rightsquigarrow x \dashv \emptyset} \qquad \frac{\Gamma, x : \tau_1 \vdash e \Rightarrow \tau_2 \rightsquigarrow e \dashv \Delta}{\Gamma \vdash \lambda x : \tau_1 . e \Rightarrow \tau_1 \to \tau_2 \rightsquigarrow \lambda x : \tau_1 . d \dashv \Delta}$$

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

$$\frac{}{\Gamma \vdash (\!|\,|\!)^u \Rightarrow (\!|\,|\!) \rightsquigarrow (\!|\,|\!)^u_{\mathrm{id}(\Gamma)} \dashv u :: (\!|\,|\!)[\Gamma]} \qquad \frac{\Gamma \vdash e \Rightarrow \tau \rightsquigarrow d \dashv \Delta}{\Gamma \vdash (\!|e|\!)^u \Rightarrow (\!|\,|\!) \rightsquigarrow (\!|d|\!)^u_{\mathrm{id}(\Gamma)} \dashv \Delta, u :: (\!|\,|\!)[\Gamma]}$$

$$\frac{\Gamma \vdash e \Leftarrow \tau \rightsquigarrow d : \tau' \dashv \Delta}{\Gamma \vdash e : \tau \Rightarrow \tau \rightsquigarrow d \langle \tau' \Rightarrow \tau \rangle \dashv \Delta}$$

$\boxed{\Gamma \vdash e \Leftarrow \tau \rightsquigarrow d : \tau_2 \dashv \Delta}$ e analyzes against type τ_1 and elaborates to d of consistent type τ_2

$$\frac{\mathit{fun}(\tau) = \tau_1 \to \tau_2 \quad \Gamma, x : \tau_1 \vdash e \Leftarrow \tau_2 \dashv d : \tau_2' \dashv \Delta}{\Gamma \vdash \lambda x . e \Leftarrow \tau \rightsquigarrow \lambda x : \tau_1 . d : \tau_1 \to \tau_2' \dashv \Delta} \qquad \frac{e \neq (\!|\,|\!)^u \quad e \neq (\!|e'|\!)^u \qquad \Gamma \vdash e \Rightarrow \tau' \rightsquigarrow d \dashv \Delta \qquad \tau \sim \tau'}{\Gamma \vdash e \Leftarrow \tau \rightsquigarrow d : \tau' \dashv \Delta}$$

$$\frac{}{\Gamma \vdash (\!|\,|\!)^u \Leftarrow \tau \rightsquigarrow (\!|\,|\!)^u_{\mathrm{id}(\Gamma)} : \tau \dashv u :: \tau[\Gamma]} \qquad \frac{\Gamma \vdash e \Rightarrow \tau' \rightsquigarrow d \dashv \Delta}{\Gamma \vdash (\!|e|\!)^u \Leftarrow \tau \rightsquigarrow (\!|d|\!)^u_{\mathrm{id}(\Gamma)} : \tau \dashv \Delta, u :: \tau[\Gamma]}$$

Fig. 3 Hazelnut Live Elaboration

$\boxed{\Delta; \Gamma \vdash d : \tau}$ d has type τ

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

$$\frac{\Delta; \Gamma \vdash d : \tau_1 \quad \tau_1 \sim \tau_2}{\Delta; \Gamma \vdash d \langle \tau_1 \Rightarrow \tau_2 \rangle : \tau_2} \qquad \frac{\Delta; \Gamma \vdash d : G_1 \quad G_1 \neq G_2}{\Delta; \Gamma \vdash d \langle G_1 \Rightarrow (\!|\,|\!) \Rightarrow G_2 \rangle : G_2}$$

Fig. 4 Hazel Live Typing for internal language

$\forall^\nu$ and $\exists^\nu$ instead of $\forall$ and $\exists$ to emphasize freshness of the abstract types. Ground types are also extended with universal and existential types. For the terms, they have $\mathsf{is}(G)?M$ for dynamic type checking. It checks whether M returns a value that is consistent with the ground type G, and returns true or false accordingly.

Typing for PolyG$^\nu$ is presented in Figure 8. The environment Γ contains not only ordinary typing assumptions $x : A$ but also abstract type variable assumptions X and known type variable assumptions $X \cong A$. We assume all (term and type) variables in the domain of Γ are disjoint. To support explicit sealing, some type variables are exposed as in:

$$\mathsf{unseal}_X((\Lambda X . \lambda x : X . x)\ \{X \cong \mathsf{Bool}\}\ \mathsf{seal}_X \mathsf{true})$$

The type variable X in unseal_X, $\{X \cong \mathsf{Bool}\}$, and seal_X is out of the scope $\Lambda X . \lambda x : X . x$. To justify such extrusion, the typing relation of PolyG$^\nu$ takes the form $\Gamma \vdash M : A; \Gamma_o$, where a type application $\{X \cong A\}$ adds known the type variable assumption $X \cong A$ to Γ_o, which is then used for typing

$\boxed{d \text{ final}}$ d is final

$$\frac{d \text{ val}}{d \text{ final}} \qquad \frac{d \text{ indet}}{d \text{ final}}$$

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

$$\frac{}{c \text{ val}} \qquad \frac{}{\lambda x : \tau . d \text{ val}} \qquad \frac{\tau_1 \to \tau_2 \neq \tau_3 \to \tau_4 \quad d \text{ val}}{d\langle \tau_1 \to \tau_2 \Rightarrow \tau_3 \to \tau_4 \rangle \text{ val}} \qquad \frac{d \text{ val} \quad \tau \text{ ground}}{d\langle \tau \Rightarrow (\!|\!) \rangle \text{ val}}$$

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

$$\frac{}{(\!|\!)^u_\sigma \text{ indet}} \qquad \frac{d \text{ final}}{(\!|d|\!)^u_\sigma \text{ indet}} \qquad \frac{d_1 \neq d_1'\langle \tau_1 \to \tau_2 \Rightarrow \tau_3 \to \tau_4 \rangle}{d_1 \ d_2 \text{ indet}}$$

$$\frac{d \text{ indet} \quad \tau \text{ ground}}{d\langle \tau \Rightarrow (\!|\!) \rangle \text{ indet}} \qquad \frac{d \neq d'\langle \tau' \Rightarrow (\!|\!) \rangle \quad d \text{ indet} \quad \tau \text{ ground}}{d\langle (\!|\!) \Rightarrow \tau \rangle \text{ indet}}$$

$$\frac{\tau_1 \to \tau_2 \neq \tau_3 \to \tau_4 \quad d \text{ indet}}{d\langle \tau_1 \to \tau_2 \Rightarrow \tau_3 \to \tau_4 \rangle \text{ indet}} \qquad \frac{d \text{ final} \quad \tau_1 \text{ ground} \quad \tau_2 \text{ ground} \quad \tau_1 \neq \tau_2}{d\langle \tau_1 \Rightarrow (\!|\!) \nRightarrow \tau_2 \rangle \text{ indet}}$$

Fig. 5 Hazelnut Live Final Forms

$$\text{Evaluation Contexts} \quad E ::= [\,] \mid E\ d \mid f\ E \mid (\!|E|\!)^u_\sigma \mid E\langle \tau \Rightarrow \tau \rangle \mid E\langle \tau \Rightarrow (\!|\!) \nRightarrow \tau \rangle$$

$\boxed{d \mapsto d'}$ Reduction

$$\begin{array}{rcll}
E[(\lambda x : \tau . d)\ f] & \mapsto & E[d[f/x]] & \\
E[f\langle \tau \Rightarrow \tau \rangle] & \mapsto & E[f] & \\
E[(f_1\langle \tau_1 \to \tau_2 \Rightarrow \tau_3 \to \tau_4 \rangle)\ f_2] & \mapsto & E[f_1(f_2\langle \tau_3 \Rightarrow \tau_1 \rangle)\langle \tau_2 \Rightarrow \tau_4 \rangle] & \\
E[f\langle \tau \Rightarrow (\!|\!) \rangle] & \mapsto & E[f\langle \tau \Rightarrow G \Rightarrow (\!|\!) \rangle] & \text{if } \tau \neq (\!|\!), \tau \neq G, \tau \sim G \\
E[f\langle (\!|\!) \Rightarrow \tau \rangle] & \mapsto & E[f\langle (\!|\!) \Rightarrow G \Rightarrow \tau \rangle] & \text{if } \tau \neq (\!|\!), \tau \neq G, \tau \sim G \\
E[f\langle G \Rightarrow (\!|\!) \Rightarrow G \rangle] & \mapsto & E[f] & \\
E[f\langle G_1 \Rightarrow (\!|\!) \Rightarrow G_2 \rangle] & \mapsto & E[f\langle G \Rightarrow (\!|\!) \nRightarrow G_2 \rangle] & (\text{if } G_1 \neq G_2)
\end{array}$$

Substitution for hole expressions

$$\begin{array}{rcl}
(\!|\!)^u_\sigma[d/x] & = & (\!|\!)^u_{\sigma[d/x]} \\
(\!|d'|\!)^u_\sigma[d/x] & = & (\!|d'[d/x]|\!)^u_{\sigma[d/x]}
\end{array}$$

Fig. 6 Dynamic Semantics of the Internal Expressions

$$\begin{array}{lrcl}
\text{Types} & A & ::= & ? \mid X \mid \mathsf{Bool} \mid A \times A \mid A \to A \mid \exists^\nu X.A \mid \forall^\nu X.A \\
\text{Ground Types} & G & ::= & X \mid \mathsf{Bool} \mid ? \times ? \mid ? \to ? \mid \exists^\nu X.? \mid \forall^\nu X.? \\
\text{Terms} & M & ::= & x \mid M :: A \mid \mathsf{seal}_X M \mid \mathsf{unseal}_X M \mid \mathsf{is}(G)?M \mid \mathsf{true} \mid \mathsf{false} \\
 & & \mid & \mathsf{if}\ M\ \mathsf{then}\ M\ \mathsf{else}\ M \mid (M, M) \mid \mathsf{let}(x, x) = M; M \\
 & & \mid & M\ M \mid \lambda x : A.M \mid \mathsf{pack}^\nu(X \cong A, M) \mid \mathsf{unpack}(X, x) = M; M \\
 & & \mid & \Lambda^\nu X.M \mid M\{X \cong A\} \mid \mathsf{let}\ x = M; M
\end{array}$$

Fig. 7 Syntax of PolyG$^\nu$

$$\text{Environment} \quad \Gamma \quad ::= \quad \cdot \mid \Gamma, x : A \mid \Gamma, X \mid \Gamma, X \cong A$$

$$\frac{\Gamma \vdash M : A; \Gamma_o \quad A \sim B}{\Gamma \vdash (M :: B) : B; \Gamma_o} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x : A; \cdot} \qquad \frac{\Gamma \vdash M : A; \Gamma_M \quad \Gamma, \Gamma_M, x : A \vdash N : B; \Gamma_N}{\Gamma \vdash \text{let } x = M; N : B; \Gamma_M, \Gamma_N}$$

$$\frac{\Gamma \vdash M : B; \Gamma_o \quad X \cong A \in \Gamma, \Gamma_o \quad B \sim A}{\Gamma \vdash \mathsf{seal}_X M : X; \Gamma_o} \qquad \frac{\Gamma \vdash M : B; \Gamma_o \quad X \cong A \in \Gamma, \Gamma_o \quad B \sim X}{\Gamma \vdash \mathsf{unseal}_X M : A; \Gamma_o}$$

$$\frac{\Gamma \vdash M : A; \Gamma_o \quad \Gamma, \Gamma_o \vdash G}{\Gamma \vdash \mathsf{is}(G)?M : \mathsf{Bool}; \Gamma_o} \qquad \frac{}{\Gamma \vdash \mathsf{true} : \mathsf{Bool}; \cdot} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{Bool}; \cdot}$$

$$\frac{\Gamma \vdash M : A; \Gamma_M \quad \Gamma, \Gamma_M \vdash N_t : B_t; \Gamma_t \quad \Gamma, \Gamma_M \vdash N_f : B_f; \Gamma_f \quad A \sim \mathsf{Bool}}{\Gamma \vdash \mathsf{if}\ M\ \mathsf{then}\ N_t\ \mathsf{else}\ N_f : B_t \sqcap B_f; \Gamma_M, \Gamma_t \cap \Gamma_f}$$

$$\frac{\Gamma \vdash M_1 : A_1; \Gamma_1 \quad \Gamma, \Gamma_1 \vdash M_2 : A_2; \Gamma_2}{\Gamma \vdash (M_1, M_2) : A_1 \times A_2; \Gamma_1, \Gamma_2} \qquad \frac{\Gamma \vdash M : A; \Gamma_M \quad \Gamma, \Gamma_M, x : \pi_1(A), y : \pi_2(A) \vdash N : B; \Gamma_N}{\Gamma \vdash \mathsf{let}(x, y) = M; N : B; \Gamma_M, \Gamma_N}$$

$$\frac{\Gamma \vdash A \quad \Gamma, x : A \vdash M : B; \Gamma_o}{\Gamma \vdash \lambda x : A.M : A \to B; \cdot} \qquad \frac{\Gamma \vdash M : A; \Gamma_M \quad \Gamma, \Gamma_M \vdash N : B; \Gamma_N \quad \mathit{dom}(A) \sim B}{\Gamma \vdash M\ N : \mathit{cod}(A); \Gamma_M, \Gamma_N}$$

$$\frac{\Gamma, X \cong A \vdash M : B; \Gamma_o}{\Gamma \vdash \mathsf{pack}^\nu(X \cong A, M) : \exists^\nu X.B; \cdot} \qquad \frac{\begin{array}{c}\Gamma \vdash M : A; \Gamma_M \\ \Gamma, \Gamma_M, X, x : \mathit{un}\exists^\nu(A) \vdash N : B; \Gamma_N \qquad \Gamma, \Gamma_M, \Gamma_N|_X \vdash B\end{array}}{\Gamma \vdash \mathsf{unpack}(X, x) = M; N : B; \Gamma_M, \Gamma_N|_X}$$

$$\frac{\Gamma, X \vdash M : A; \Gamma_o}{\Gamma \vdash \Lambda^\nu X.M : \forall^\nu X.A; \cdot} \qquad \frac{\Gamma \vdash M : A; \Gamma_M \quad \Gamma, \Gamma_M \vdash B}{\Gamma \vdash M\{X \cong B\} : \mathit{un}\forall^\nu(A); \Gamma_M, X \cong B}$$

$$\frac{}{? \sim A} \qquad \frac{}{A \sim ?} \qquad \frac{}{\mathsf{Bool} \sim \mathsf{Bool}} \qquad \frac{}{X \sim X}$$

$$\frac{A_1 \sim B_1 \quad A_2 \sim B_2}{A_1 \to A_2 \sim B_1 \to B_2} \qquad \frac{A_1 \sim B_1 \quad A_2 \sim B_2}{A_1 \times A_2 \sim B_1 \times B_2} \qquad \frac{A \sim B}{\exists^\nu X.A \sim \exists^\nu X.B} \qquad \frac{A \sim B}{\forall^\nu X.A \sim \forall^\nu X.B}$$

$$\begin{aligned}
\mathit{dom}(A \to B) &= A & \qquad \mathit{un}\exists^\nu(\exists^\nu X.A) &= A \\
\mathit{dom}(?) &= ? & \qquad \mathit{un}\exists^\nu(?) &= ? \\
\mathit{cod}(A \to B) &= B & & \\
\mathit{cod}(?) &= ? & & \\
\pi_i(A_1 \times A_2) &= A_i & \qquad \cdot|_{\Gamma'} &= \cdot \\
\pi_i(?) &= ? & \qquad (X \cong A, \Gamma)|_{\Gamma'} &= X \cong A, (\Gamma|_{\Gamma'}) \quad (FV(A) \cap \Gamma' = \emptyset) \\
\mathit{un}\forall^\nu(\forall^\nu X.A) &= A & \qquad (X \cong A, \Gamma)|_{\Gamma'} &= \Gamma|_{\Gamma'} \quad (FV(A) \cap \Gamma' \neq \emptyset) \\
\mathit{un}\forall^\nu(?) &= ? &
\end{aligned}$$

Fig. 8 Static Semantics of PolyG$^\nu$

the continuation including seal_X and unseal_X.

For the syntax of the cast calculus PolyC$^\nu$, they add fresh seal σ to types as in Figure 9. For the terms, they remove type ascription $M :: A$ and add an error $\mathfrak{V}$ and explicit casts. Forms $\langle A^{\sqsubseteq}\rangle \upharpoonright M$ and $\langle A^{\sqsubseteq}\rangle \downarrow M$ represent upcasts and downcasts respectively. Instead of writing casts between two different types, they use witness expressions for type precision derived as in Figure 10. $\mathsf{inj}_G M$ represents injection, which is a cast from the ground type G to the dynamic type ?. PolyC$^\nu$ also has explicit type generation $\mathsf{hide}\ X \cong A; M$, packages with latent casts $\mathsf{pack}^\nu(X \cong A', [A^{\sqsubseteq}\ \updownarrow, \dots], M)$, and sealing and unsealing with seals (in addition to those with type variables).

Typing for PolyC$^\nu$, presented in Figure 11, is trivial except for Γ_o. Since there is a new form $\mathsf{hide}\ X \cong A; M$ that stops the scope of $X \cong A$ going further outwards, the bodies inside delayed thunks such as λ are expected to manually hide the bindings to make Γ_o empty.

Elaboration from PolyG$^\nu$ to PolyC$^\nu$ is defined in Figure 12. The ascription form $M :: A$ elaborates to upcast to ? followed by downcast to type A. The bodies of the thunks hide the type bindings that are generated inside, by using the function $\mathsf{hide}\ \Gamma \subseteq \Gamma'; M$ which hides all names in Γ' that are not present in Γ.

The dynamic semantics of PolyC$^\nu$ is presented in Figure 13. The relation is defined in the form of $\Sigma \triangleright M \mapsto \Sigma' \triangleright M'$. A store Σ consists of $\sigma : A$ which binds freshly generated type σ to a type A. When the step does not change Σ, we write $M \mapsto M'$ as an abbreviation of $\Sigma \triangleright M \mapsto \Sigma \triangleright M'$.

Unsealing a sealed value removes both unsealing and sealing, and gets the value inside. $\mathsf{is}(G)?V$ checks whether the type of the value V is consistent with G. hide generates a fresh seal σ, adds the binding $\sigma : A$ to Σ, and substitutes the type variable X with σ. unpack also generates a fresh seal and applies casts that have been accumulated in the second, extended parameter of pack. $\updownarrow$ is used to abbreviate rules for $\upharpoonright$ and $\downarrow$. Instantiation for a universal type is just a substitution because the binding is already generated by hide. The casts between pair types, function types, existential types, and universal types are decomposed and separately applied, where $\updownarrow^-$ represents the opposite arrow, i.e., $\upharpoonright^- = \downarrow$ and $\upharpoonright^- = \downarrow$.

3 External language

As we mentioned in the introduction, our goal is to introduce hole expressions to a polymorphic gradual language and to give static and dynamic semantics to incomplete programs with polymorphism. We define a new calculus PolyG$^\nu$H by extending PolyG$^\nu$ with hole expressions $(\!|\,|\!)$ and $(\!|M|\!)$ of Hazelnut Live. The syntax for those added terms are in Figure 14. Since we do not consider hole contexts, the hole names u are omitted.

Typing rules for the new terms are shown in Figure 15. An empty hole expression $(\!|\,|\!)$ has type ? and any Γ_o. For non-empty holes, the additional environment Γ' inside allows M to have unbound variables. This works for both term and type variables. Also, the outward type assumption Γ'_o is ignored and another Γ_o is just assumed, because we do not want wrongly written terms influence the rest of the program.

For functions $\lambda(\!|\,|\!) : A.\ M$ with an unspecified formal parameter, we could either introduce a fresh variable, or no variable at all, to the environment Γ. We choose the latter for simplicity.

By contrast, we *do* introduce a fresh type variable X' for the unspecified type parameter of $\Lambda^\nu(\!|\,|\!).\ M$, for the *type* binder $\forall^\nu X'$ instead of introducing yet another syntax $\forall^\nu(\!|\,|\!).\ A$.

Similarly, type applications $M\{(\!|\,|\!) \cong B\}$ introduce a fresh type variable X'.

Packing $\mathsf{pack}^\nu((\!|\,|\!) \cong A, M)$ is also typed by intro-

$$
\begin{array}{llll}
\text{Type names} & \alpha & ::= & \sigma \mid X \\
\text{Types} & A, B & + ::= & \sigma \\
\text{Ground Types} & G & ::= & \alpha \mid \mathsf{Bool} \mid ? \times ? \mid ? \to ? \mid \exists^{\nu} X.? \mid \forall^{\nu} X.? \\
\text{Precision derivations} & A^{\sqsubseteq}, B^{\sqsubseteq} & ::= & ? \mid \mathrm{tag}_G(A^{\sqsubseteq}) \mid \alpha \mid \mathsf{Bool} \mid A^{\sqsubseteq} \times A^{\sqsubseteq} \mid A^{\sqsubseteq} \to A^{\sqsubseteq} \\
 & & \mid & \exists^{\nu} X.A^{\sqsubseteq} \mid \forall^{\nu} X.A^{\sqsubseteq} \\
\text{Values} & V & ::= & \mathsf{seal}_{\alpha} V \mid \mathsf{true} \mid \mathsf{false} \mid (V, V) \mid \lambda x : A.M \\
 & & \mid & \Lambda^{\nu} X.M \mid \mathsf{inj}_G V \mid \langle A_1^{\sqsubseteq} \to A_2^{\sqsubseteq} \rangle \upharpoonright M \mid \langle A_1^{\sqsubseteq} \to A_2^{\sqsubseteq} \rangle \downharpoonright M \\
 & & \mid & \langle \forall^{\nu} X.A^{\sqsubseteq} \rangle \upharpoonright M \mid \langle \forall^{\nu} X.A^{\sqsubseteq} \rangle \downharpoonright M \\
 & & \mid & \mathsf{pack}^{\nu}(X \cong A', [A^{\sqsubseteq} \updownarrow], M) \\
\text{Expressions} & M, N & - ::= & (M :: A) \\
 & & + ::= & \mho \mid \langle A^{\sqsubseteq} \rangle \upharpoonright M \mid \langle A^{\sqsubseteq} \rangle \downharpoonright M \mid \mathsf{hide}\ X \cong A; M \mid \mathsf{inj}_G M \\
 & & \mid & \mathsf{pack}^{\nu}(X \cong A', [A \sqsubseteq^{\updownarrow}, \dots], M) \mid \mathsf{seal}_{\sigma} M \mid \mathsf{unseal}_{\sigma} M \\
\text{Evaluation Context} & E & ::= & [\,] \mid (E, M) \mid (V, E) \mid E[A] \mid E\ M \mid V\ E \mid \mathsf{inj}_G E \\
 & & \mid & \mathsf{if}\ E\ \mathsf{then}\ M\ \mathsf{else}\ M \mathsf{if}\ I\ \mathsf{then}\ M\ \mathsf{else}\ M \\
 & & \mid & \mathsf{let}(x, x) = E; M \mid \langle A^{\sqsubseteq} \rangle \upharpoonright E \\
 & & \mid & \mathsf{unpack}(X, x) = E; M \mid \mathsf{seal}_{\alpha} E \mid \mathsf{unseal}_{\alpha} E \mid \langle A^{\sqsubseteq} \rangle \downharpoonright E
\end{array}
$$

Fig. 9 Syntax of PolyC$^{\nu}$

$$
\frac{\Gamma \vdash A^{\sqsubseteq} : A \sqsubseteq G}{\Gamma \vdash \mathrm{tag}_G(A^{\sqsubseteq}) : A \sqsubseteq ?} \qquad \frac{}{\Gamma \vdash ? : ? \sqsubseteq ?} \qquad \frac{}{\Gamma \vdash \mathsf{Bool} : \mathsf{Bool} \sqsubseteq \mathsf{Bool}} \qquad \frac{X \in \Gamma}{\Gamma \vdash X : X \sqsubseteq X}
$$

$$
\frac{\Gamma \vdash A_1^{\sqsubseteq} : A_{l1} \sqsubseteq A_{r1} \quad \Gamma \vdash A_2^{\sqsubseteq} : A_{l2} \sqsubseteq A_{r2}}{\Gamma \vdash A_1^{\sqsubseteq} \times A_2^{\sqsubseteq} : A_{l1} \times A_{l2} \sqsubseteq A_{r1} \times A_{r2}} \qquad \frac{\Gamma \vdash A_1^{\sqsubseteq} : A_{l1} \sqsubseteq A_{r1} \quad \Gamma \vdash A_2^{\sqsubseteq} : A_{l2} \sqsubseteq A_{r2}}{\Gamma \vdash A_1^{\sqsubseteq} \to A_2^{\sqsubseteq} : A_{l1} \to A_{l2} \sqsubseteq A_{r1} \times A_{r2}}
$$

$$
\frac{\Gamma \vdash A^{\sqsubseteq} : A_l \sqsubseteq A_r}{\Gamma \vdash \exists^{\nu} A^{\sqsubseteq} : \exists^{\nu} A_l^{\sqsubseteq} \sqsubseteq \exists^{\nu} A_r^{\sqsubseteq}} \qquad \frac{\Gamma \vdash A^{\sqsubseteq} : A_l \sqsubseteq A_r}{\Gamma \vdash \forall^{\nu} A^{\sqsubseteq} : \forall^{\nu} A_l^{\sqsubseteq} \sqsubseteq \exists^{\nu} A_r^{\sqsubseteq}}
$$

Fig. 10 PolyC$^{\nu}$ Type Precision

ducing fresh X' as in the rule for Λ.

Since unpack binds two (term or type) variables, we add 3 rules for each combination of missing parameters. The case where only a term or type variable is unspecified is similar to the case of $\lambda(\!|\,|\!)$ or $\Lambda^{\nu}(\!|\,|\!)$. However, when *both* parameters are missing, we do *not* have to introduce fresh X' since it does not appear anywhere.

Sealing $\mathsf{seal}_{(\!|\,|\!)} M$ and unsealing $\mathsf{unseal}_{(\!|\,|\!)} M$ with an unspecified key are typed just as $\mathsf{seal}_X M$ and $\mathsf{unseal}_X M$ for some X.

Dynamic type check $(\mathsf{is}(\!|\,|\!)?M)$ is trivial.

There are also new let expressions with one or two unspecified bound (term) variables. Their typing rules are similar to that of λ.

4 Internal language

The cast calculus PolyC$^{\nu}$H (shown in Figure 16) is obtained by elaborating PolyG$^{\nu}$H like elebroration from PolyG$^{\nu}$ to PolyC$^{\nu}$, or, equivalently, adopting Hazelnut Live for PolyC$^{\nu}$ like we did for PolyG$^{\nu}$, and by replacing errors $\mho$ (abortion of the entire program) with indeterminate failed casts.

We added rules, shown in Figure 17, for elabora-

$$\frac{\Gamma \vdash M : A_l ; \Gamma_M \quad \Gamma \vdash A^{\sqsubseteq} : A_l \sqsubseteq A_r}{\Gamma \vdash \langle A^{\sqsubseteq} \rangle \updownarrow M : A_r ; \Gamma_M} \qquad \frac{\Gamma \vdash M : A_r ; \Gamma_M \quad \Gamma \vdash A^{\sqsubseteq} : A_l \sqsubseteq A_r}{\Gamma \vdash \langle A^{\sqsubseteq} \rangle \downarrow M : A_l ; \Gamma_M}$$

$$\frac{\Gamma \vdash M : \Gamma_M, X \cong A, \Gamma'_M \quad \Gamma, \Gamma_M \vdash \Gamma'_M}{\Gamma \vdash \mathsf{hide}\ X \cong A ; M ; \Gamma_M, \Gamma'_M}$$

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A ; \cdot} \qquad \frac{\Gamma \vdash M : A ; \Gamma_M \quad \Gamma, \Gamma_M, x : A \vdash N : B ; \Gamma_N}{\Gamma \vdash \mathsf{let}\ x = M ; N : B ; \Gamma_M, \Gamma_N}$$

$$\frac{\Gamma \vdash M : A ; \Gamma_o \quad X \cong A \in \Gamma, \Gamma_o}{\Gamma \vdash \mathsf{seal}_X M : X ; \Gamma_o} \qquad \frac{\Gamma \vdash M : A ; \Gamma_o \quad X \cong A \in \Gamma, \Gamma_o}{\Gamma \vdash \mathsf{unseal}_X M : A ; \Gamma_o}$$

$$\frac{\Gamma \vdash M : ? ; \Gamma_o \quad \Gamma, \Gamma_o \vdash G}{\Gamma \vdash \mathsf{is}(G)?M : \mathsf{Bool} ; \Gamma_o} \qquad \frac{}{\Gamma \vdash \mathsf{true} : \mathsf{Bool} ; \cdot} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{Bool} ; \cdot}$$

$$\frac{\Gamma \vdash M : \mathsf{Bool} ; \Gamma_o \quad \Gamma, \Gamma_M \vdash N_t : B ; \Gamma_N \quad \Gamma, \Gamma_M \vdash N_f : B ; \Gamma_N}{\Gamma \vdash \mathsf{if}\ M\ \mathsf{then}\ N_t\ \mathsf{else}\ N_f : B ; \Gamma_M, \Gamma_N}$$

$$\frac{\Gamma \vdash M_1 : A_1 ; \Gamma_1 \quad \Gamma, \Gamma_1 \vdash M_2 : A_2 ; \Gamma_2}{\Gamma \vdash (M_1, M_2) : A_1 \times A_2 ; \Gamma_1, \Gamma_2} \qquad \frac{\Gamma \vdash M : A_1 \times A_2 ; \Gamma_M \quad \Gamma, \Gamma_M, x : A_1, y : A_2 \vdash N : B ; \Gamma_N}{\Gamma \vdash \mathsf{let}(x, y) = M ; N : B ; \Gamma_M, \Gamma_N}$$

$$\frac{\Gamma \vdash A \quad \Gamma, x : A \vdash M : B ; \cdot}{\Gamma \vdash \lambda x : A.M : A \to B ; \cdot} \qquad \frac{\Gamma \vdash M : A \to B ; \Gamma_M \quad \Gamma, \Gamma_M \vdash N : A ; \Gamma_N}{\Gamma \vdash M\ N : B ; \Gamma_M, \Gamma_N}$$

$$\frac{\Gamma, X \cong A \vdash M : B ; \cdot}{\Gamma \vdash \mathsf{pack}^\nu(X \cong A, M) : \exists^\nu X.B ; \cdot} \qquad \frac{\begin{array}{c}\Gamma \vdash M : \exists^\nu X.A ; \Gamma_M \\ \Gamma, \Gamma_M, X, x : A \vdash N : B ; \Gamma_N \qquad \Gamma, \Gamma_M, \Gamma_N \vdash B\end{array}}{\Gamma \vdash \mathsf{unpack}(X, x) = M ; N : B ; \Gamma_M, \Gamma_N}$$

$$\frac{\Gamma, X \vdash M : A ; \cdot}{\Gamma \vdash \Lambda^\nu X.M : \forall^\nu X.A ; \cdot} \qquad \frac{\Gamma \vdash M : \forall^\nu X.A ; \Gamma_M \quad \Gamma, \Gamma_M \vdash B}{\Gamma \vdash M\{X \cong B\} : A ; \Gamma_M, X \cong B}$$

Fig. 11 PolyC$^\nu$ Typing

tion of hole expressions. For non-empty holes, we want to hide the variables that are only valid inside of the holes. Thus we extend $\mathsf{hide}\ \Gamma_s \subseteq \Gamma_b ; M$ for type environments containing term variables $x : A$ and abstract type variables X, which we bind to ?, in addition to known type variables $X \cong A$. We furthermore introduce known type variable assumptions that are present only outside (not inside) holes, by inserting dummy type applications.

To define dynamic semantics, we define final and indeterminate forms as in Figure 18. As we discussed in the introduction, we have added the new syntax $\lambda(\!|\,|\!) : A.M$ and $\lambda(\!|\,|\!).M$, rather than $\lambda x' : A.M$ and $\lambda X.M$ for fresh x' or X to treat them as indeterminate even if they are applied. The definition of values is unchanged from PolyC$^\nu$.

The dynamic semantics of PolyC$^\nu$H is defined in Figure 19. Most cases are obtained by replacing V with F from that of PolyC$^\nu$. The other differences are evaluation contexts for non-empty holes $(\!|E|\!)$ and failed casts $\langle G \Leftarrow ? \Leftarrow H \rangle E$, and the reduction rule for failed casts, all of which are essentially the same as in Hazelnut Live.

$$
\begin{array}{rcll}
(M :: B)^+ & = & \langle B?^{\sqsubseteq}\rangle \downarrow \langle A?^{\sqsubseteq}\rangle \uparrow M^+ & (\text{where } M : A \text{ and } A?^{\sqsubseteq} : A \sqsubseteq ?) \\
x^+ & = & x & \\
(\mathsf{let}\ x = M; N)^+ & = & \mathsf{let}\ x = M^+; N^+ & \\
(\mathsf{seal}_X M)^+ & = & \mathsf{seal}_X (M :: A)^+ & \\
(\mathsf{unseal}_X M)^+ & = & \mathsf{unseal}_X (X \wr M) & (\text{where } X \cong A) \\
(\mathsf{is}(G)?M)^+ & = & \mathsf{is}(G)?(\langle A?^{\sqsubseteq}\rangle \uparrow M^+) & (\text{where } M : A \text{ and } A?^{\sqsubseteq} : A \sqsubseteq ?) \\
b^+ & = & b & (\text{where } b \in \mathsf{true}, \mathsf{false}) \\
(\mathsf{if}\ M\ \mathsf{then}\ N_t\ \mathsf{else}\ N_f) & = & \mathsf{if}\ \mathsf{Bool} \wr M\ \mathsf{then}\ \langle B_t^{\sqsubseteq}\rangle \downarrow \mathsf{hide}\ \Gamma_t \subseteq \Gamma_t \cap \Gamma_f; N_t^+ & \\
 & & \quad \mathsf{else}\ \langle B_f^{\sqsubseteq}\rangle \downarrow \mathsf{hide}\ \Gamma_f \subseteq \Gamma_t \cap \Gamma_f; N_f^+ & \\
 & & (\text{where } \Gamma \vdash \mathsf{if}\ M\ \mathsf{then}\ N_t\ \mathsf{else}\ N_f : B_t \sqcap B_f; \Gamma_M, \Gamma_t \cap \Gamma_f) & \\
 & & \quad (\text{and } B_t^{\sqsubseteq} : B_t \sqcap B_f \sqsubseteq) B_t,\ B_f^{\sqsubseteq} : B_t \sqcap B_f \sqsubseteq B_f) & \\
(M_1, M_2)^+ & = & (M_1^+, M_2^+) & \\
(\mathsf{let}(x,y) = M; N) & = & \mathsf{let}(x,y) = ? \times ? \wr M; N^+ & \\
(\lambda x : A.M)^+ & = & \lambda x : A.\mathsf{hide}\ \cdot \subseteq \Gamma_o; M^+ & (\text{where } M : A; \Gamma_o) \\
(M\ N)^+ & = & (? \to ? \wr M)(N :: dom(A))^+ & (\text{where } M : A) \\
(\mathsf{pack}^\nu (X \cong A, M))^+ & = & \mathsf{pack}^\nu (X \cong A, \mathsf{hide}\ \cdot \subseteq \Gamma_o, M^+) & (\text{where } M : B; \Gamma_o) \\
(\mathsf{unpack}(X,x) = M; N)^+ & = & \mathsf{unpack}(X,x) = \exists^\nu X.? \wr M; \mathsf{hide}\ \Gamma_{N|X} \subseteq \Gamma_N; N^+ & \\
\Lambda^\nu X.M^+ & = & \Lambda^\nu X.\mathsf{hide}\ \cdot \subseteq \Gamma_o; M^+ & \\
M\{X \cong B\}^+ & = & (\forall^\nu X.? \wr M)\{X \cong B\} &
\end{array}
$$

$$
\begin{array}{rcll}
G \wr M & = & \langle \mathsf{tag}_G(G)\rangle \downarrow M^+ & (\text{when } M : ?) \\
G \wr M & = & M^+ & (\text{otherwise})
\end{array}
$$

$$
\begin{array}{rcll}
\mathsf{hide}\ \Gamma_s \subseteq (\Gamma_b, X \cong A); M & = & \mathsf{hide}\ \Gamma_s \subseteq \Gamma_b; \mathsf{hide} X \cong A; M & (X \notin \Gamma_s) \\
\mathsf{hide}\ (\Gamma_s, X \cong A) \subseteq (\Gamma_b, X \cong A); M & = & \mathsf{hide}\ \Gamma_s \subseteq \Gamma_b; M & \\
\mathsf{hide}\ \cdot \subseteq \cdot; M & = & M &
\end{array}
$$

Fig. 12 Elaboration from PolyG$^\nu$ to PolyC$^\nu$

5 Properties

In this chapter, we prove the type safety (progress and preservation) of our new calculus PolyC$^\nu$H.[†1] To prove progress, we prove canonical forms lemma according to the definition of values.

Lemma 5.1 (Canonical Forms for Values)**.** *If* $\Sigma; \cdot \vdash V : A; \cdot$ *then:*

- $A = \mathsf{Bool}$ *and* $V = \mathsf{true}$ *or* false
- $A = \sigma$ *and* $V = \mathsf{seal}_\sigma V'$
- $A = A_1 \times A_2$ *and* $V = (V_1, V_2)$
- $A = A_1 \to A_2$ *and* $V = \lambda x : A_1.M'$ *or* $\langle A_1^{\sqsubseteq} \to A_2^{\sqsubseteq}\rangle \updownarrow V'$
- $A = \forall^\nu X.A'$ *and* $V = \Lambda^\nu X.M'$ *or* $\langle \forall^\nu X.A'^{\sqsubseteq}\rangle \updownarrow V'$
- $A = \exists^\nu X.A'$ *and* $V = \mathsf{pack}^\nu (X \cong A', [A^{\sqsubseteq} \updownarrow], M')$
- $A = ?$ *and* $V = \mathsf{inj}_G V'$

□

We define a well-typed closed term as $\Sigma_1; \cdot \vdash M_1 : A; \cdot$, where the type judgements $\Sigma; \Gamma \vdash$

†1 Proving type preservation of the elaboration from PolyG$^\nu$H to PolyC$^\nu$H is out of the scope of the present paper and is left for future work.

$$
\begin{array}{lcl}
E[\mho] & \mapsto & \mho \text{ where } E \neq [\,] \\
E[\mathsf{unseal}_\sigma(\mathsf{seal}_\sigma V)] & \mapsto & E[V] \\
E[\mathsf{is}(G)?(\mathsf{inj}_G V)] & \mapsto & E[\mathsf{true}] \\
E[\mathsf{is}(G)?(\mathsf{inj}_H V)] & \mapsto & E[\mathsf{false}] \text{ where } G \neq H \\
\Sigma \triangleright E[\mathsf{hide}\ X \cong A; M] & \mapsto & \Sigma, \sigma : A \triangleright E[M[\sigma/X]] \\
\Sigma \triangleright E\left[\begin{array}{l}\mathsf{unpack}(X,x) = \mathsf{pack}^\nu(X \cong A', [\overline{A^{\sqsubseteq}\updownarrow}], M); \\ N\end{array}\right] & \mapsto & \Sigma, \sigma : A' \triangleright E\left[\begin{array}{l}\mathsf{let}\ x = \overline{\langle A^{\sqsubseteq}\rangle\updownarrow} M[\sigma/X]; \\ N[\sigma/X]\end{array}\right] \\
E[\mathsf{pack}^\nu(X \cong A, M)] & \mapsto & E[\mathsf{pack}^\nu(X \cong A', [], M)] \\
E[(\Lambda^\nu X.M)\{\sigma \cong A\}] & \mapsto & E[M[\sigma/X]] \\
E[\langle A^{\sqsubseteq}\rangle \updownarrow V] & \mapsto & E[V] \text{ where } A^{\sqsubseteq} \in \{\mathsf{Bool}, \sigma, ?\} \\
E[\langle A_1^{\sqsubseteq} \times A_2^{\sqsubseteq}\rangle \updownarrow (V_1, V_2)] & \mapsto & E[(\langle A_1^{\sqsubseteq}\rangle \updownarrow V_1, \langle A_2^{\sqsubseteq}\rangle \updownarrow V_2)] \\
E[(\langle A_1^{\sqsubseteq} \to A_2^{\sqsubseteq}\rangle \updownarrow V_1)\ V_2] & \mapsto & E[\langle A_2^{\sqsubseteq}\rangle \updownarrow (V_1\ \langle A_2^{\sqsubseteq}\rangle \updownarrow^- V_2)] \\
E[\langle \exists^\nu X.A^{\sqsubseteq}\rangle \updownarrow \mathsf{pack}^\nu(X \cong A, [A'^{\sqsubseteq} \updownarrow', \ldots], M)] & \mapsto & E[\mathsf{pack}^\nu(X \cong A, [A^{\sqsubseteq} \updownarrow, A'^{\sqsubseteq} \updownarrow', \ldots], M)] \\
E[\langle \forall^\nu X.A^{\sqsubseteq}\rangle \updownarrow V\{\sigma \cong A\}] & \mapsto & E[\langle A^{\sqsubseteq}[\sigma/X]\rangle \updownarrow (V\{\sigma \cong A\})] \\
E[\langle \mathrm{tag}_G(A^{\sqsubseteq})\rangle \upharpoonright V] & \mapsto & E[\mathsf{inj}_G\langle A^{\sqsubseteq}\rangle \upharpoonright V] \\
E[\langle \mathrm{tag}_G(A^{\sqsubseteq})\rangle \downarrow \mathsf{inj}_G V] & \mapsto & E[\langle A^{\sqsubseteq}\rangle \downarrow V] \\
E[\langle \mathrm{tag}_G(A^{\sqsubseteq})\rangle \downarrow \mathsf{inj}_H V] & \mapsto & \mho \text{ where } H \neq G
\end{array}
$$

Fig. 13 Dynamic Semantics PolyC$^\nu$

$$
\begin{array}{lrcl}
\text{Terms} & M & +::= & (\!|\,|\!) \mid (\!|M|\!) \\
& & \mid & \lambda(\!|\,|\!) : A.\ M \\
& & \mid & \Lambda^\nu(\!|\,|\!).\ M \mid M\{(\!|\,|\!) \cong A\} \\
& & \mid & \mathsf{pack}^\nu((\!|\,|\!) \cong A, M) \\
& & \mid & \mathsf{unpack}((\!|\,|\!), (\!|\,|\!)) = M; N \\
& & \mid & \mathsf{unpack}(X, (\!|\,|\!)) = M; N \\
& & \mid & \mathsf{unpack}((\!|\,|\!), x) = M; N \\
& & \mid & \mathsf{seal}_{(\!|\,|\!)} M \mid \mathsf{unseal}_{(\!|\,|\!)} M \mid \mathsf{is}((\!|\,|\!))?M \\
& & \mid & \mathsf{let}\ (\!|\,|\!) = M; N \mid \mathsf{let}\ ((\!|\,|\!), (\!|\,|\!)) = M; N \\
& & \mid & \mathsf{let}\ (x, (\!|\,|\!)) = M; N \mid \mathsf{let}\ ((\!|\,|\!), y) = M; N
\end{array}
$$

Fig. 14 PolyG$^\nu$H Syntax

$M : A; \Gamma_o$ with store typing Σ can be defined as in PolyC$^\nu$ [`http://www.ccs.neu.edu/home/amal/papers/gradparam-tr.pdf`, Fig. 23].

Progress theorem ensures irreducible terms are either a value or an indet, i.e., final.

Theorem 5.2 (Progress)**.** *If* $\Sigma_1; \cdot \vdash M_1 : A; \cdot$ *then either* $\Sigma_1 \triangleright M_1 \mapsto \Sigma_2 \triangleright M_2$*, or* $M_1 = V$*, or* $M_1 = I$*.*

Proof. Induction on the derivation of $\Sigma_1; \cdot \vdash M_1 : A; \cdot$. Most cases amount to checking that our definition of indets covers irreducible non-values. □

As for preservation, we need to prove substitution lemmas for terms and types.

Lemma 5.3 (Substitution Lemma)**.** *If* $\Sigma; \Gamma, x : A \vdash M_1 : B; \Gamma_o$ *and* $\Sigma; \Gamma \vdash M_2 : A; \cdot$ *then* $\Sigma; \Gamma \vdash M_1[M_2/x] : B; \Gamma_o$

Proof. Induction on derivation of $\Sigma; \Gamma, x : A \vdash M_1 :$

$$\frac{}{\Gamma \vdash (\!|\,|\!) : ? ; \Gamma_o} \qquad \frac{\Gamma, \Gamma' \vdash M : A ; \Gamma'_o}{\Gamma \vdash (\!|M|\!) : ? ; \Gamma_o}$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash M : B ; \Gamma_o}{\Gamma \vdash \lambda (\!|\,|\!) : A . M : A \to B ; \cdot}$$

$$\frac{\Gamma, X' \vdash M : A ; \Gamma_o \quad X' \notin \Gamma, \Gamma_o}{\Gamma \vdash \Lambda (\!|\,|\!) . M : \forall^\nu X' . A ; \cdot} \qquad \frac{\Gamma \vdash M : A ; \Gamma_M \quad \Gamma, \Gamma_M \vdash B \quad X' \notin \Gamma, \Gamma_M}{\Gamma \vdash M \ \{(\!|\,|\!) \cong B\} : un \forall^\nu (A)[X'/X] ; \Gamma_M, X' \cong B}$$

$$\frac{\Gamma, X' \cong A, \vdash M : B ; \Gamma_o \quad X' \notin \Gamma, \Gamma_o}{\Gamma \vdash \mathsf{pack}^\nu ((\!|\,|\!) \cong A, M) : \exists^\nu X' . B ; \cdot}$$

$$\frac{\Gamma \vdash M : A ; \Gamma_M \quad \Gamma, \Gamma_M, X \vdash N : B ; \Gamma_N \quad \Gamma, \Gamma_M, \Gamma_{N|X} \vdash B}{\Gamma \vdash \mathsf{unpack}(X, (\!|\,|\!)) = M ; N : B ; \Gamma_M, \Gamma_{N|X}}$$

$$\frac{\Gamma \vdash M : A ; \Gamma_M \quad \Gamma, \Gamma_M, X', x : un \exists^\nu (A)[X'/X] \vdash N : B ; \Gamma_N \quad \Gamma, \Gamma_M, \Gamma_{N|X'} \vdash B}{\Gamma \vdash \mathsf{unpack}((\!|\,|\!), x) = M ; N : B ; \Gamma_M, \Gamma_{N|X'}}$$

$$\frac{\Gamma \vdash M : A ; \Gamma_M \quad \Gamma, \Gamma_M \vdash N : B ; \Gamma_N}{\Gamma \vdash \mathsf{unpack}((\!|\,|\!), (\!|\,|\!)) = M ; N : B ; \Gamma_M, \Gamma_N} \qquad \frac{\Gamma \vdash M : A ; \Gamma_o}{\Gamma \vdash \mathsf{is}((\!|\,|\!))? M : \mathsf{Bool} ; \Gamma_o}$$

$$\frac{\Gamma \vdash M : B ; \Gamma_o \quad X \cong A \in \Gamma, \Gamma_o \quad A \sim B}{\Gamma \vdash \mathsf{seal}_{(\!|\,|\!)} M : X ; \Gamma_o} \qquad \frac{\Gamma \vdash M : B ; \Gamma_o \quad X \cong A \in \Gamma, \Gamma_o \quad B \sim X}{\Gamma \vdash \mathsf{unseal}_{(\!|\,|\!)} M : A ; \Gamma_o}$$

$$\frac{\Gamma \vdash M : A ; \Gamma_M \quad \Gamma, \Gamma_M \vdash N : B ; \Gamma_N}{\Gamma \vdash \mathsf{let}\ (\!|\,|\!) = M ; N : B ; \Gamma_M, \Gamma_N} \qquad \frac{\Gamma \vdash M : A ; \Gamma_M \quad \Gamma, \Gamma_M \vdash N : B ; \Gamma_N}{\Gamma \vdash \mathsf{let}\ ((\!|\,|\!), (\!|\,|\!)) = M ; N : B ; \Gamma_M, \Gamma_N}$$

$$\frac{\Gamma \vdash M : A ; \Gamma_M \quad \Gamma, \Gamma_M, x : \pi_1(A) \vdash N : B ; \Gamma_N}{\Gamma \vdash \mathsf{let}\ (x, (\!|\,|\!)) = M ; N : B ; \Gamma_M, \Gamma_N} \qquad \frac{\Gamma \vdash M : A ; \Gamma_M \quad \Gamma, \Gamma_M, y : \pi_2(A) \vdash N : B ; \Gamma_N}{\Gamma \vdash \mathsf{let}\ ((\!|\,|\!), y) = M ; N : B ; \Gamma_M, \Gamma_N}$$

Fig. 15 PolyG$^\nu$H Typing

$B ; \Gamma_o$. □

Lemma 5.4 (Substitution Lemma for Type)**.** *If* $\Sigma ; \Gamma, X \cong A \vdash M_1 : B ; \Gamma_o$ *or* $\Sigma ; \Gamma, X \vdash M_1 : B ; \Gamma_o$ *or* $\Sigma ; \Gamma \vdash M_1 : B ; \Gamma_o, X \cong A$ *then* $\Sigma, \sigma : A ; \Gamma[\sigma/X] \vdash M_1[\sigma/X] : B[\sigma/X] ; \Gamma_o[\sigma/X]$

Proof. Induction on type derivation of M_1. □

We do not need to change the definition of preservation even though substitution of a type variable also substitutes the resulting type, because the type of a closed term does not contain type variables.

Theorem 5.5 (Preservation)**.** *If* $\Sigma_1 ; \cdot \vdash M_1 : A ; \cdot$ *and* $\Sigma_1 \triangleright M_1 \mapsto \Sigma_2 \triangleright M_2$ *then* $\Sigma_2 ; \cdot \vdash M_2 : A ; \cdot$

Proof. Induction on derivation of $\Sigma_1 ; \cdot \vdash M_1 : A ; \cdot$. □

6 Conclusion

By combining two calculi, one with hole expressions and one with polymorphic gradual typing, we defined a calculus that can give dynamic meaning to incomplete programs with polymorphism.

In this paper, we adopted explicit parametric polymorphism like System F and explicit sealing (and unsealing) of PolyG$^\nu$. To reduce program-

Syntax

$$\begin{array}{rrcl} \text{Terms} & M & - ::= & (M :: A) \\ & & + ::= & \langle A^{\sqsubseteq} \rangle \upharpoonright M \mid \langle A^{\sqsubseteq} \rangle \downarrow M \mid \mathsf{hide}\ X \cong A; M \mid \mathsf{inj}_G M \\ & & \mid & \mathsf{pack}^{\nu}(X \cong A', [A \sqsubseteq^{\updownarrow}, \ldots], M) \mid \mathsf{seal}_{\sigma} M \mid \mathsf{unseal}_{\sigma} M \\ & & \mid & \langle G \nLeftarrow ? \Leftarrow H \rangle M \end{array}$$

Typing

$$\frac{}{\Gamma \vdash (\!|\,|\!) : ?; \cdot} \qquad \frac{\Gamma \vdash M : A; \Gamma_o}{\Gamma \vdash (\!|M|\!) : ?; \Gamma_o}$$

$$\frac{\Gamma \vdash M : H; \Gamma_o}{\Gamma \vdash \langle G \nLeftarrow ? \Leftarrow H \rangle M : G; \Gamma_o}$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash M : B; \cdot}{\Gamma \vdash \lambda(\!|\,|\!) : A.M : A \to B; \cdot}$$

$$\frac{\Gamma \vdash M : A; \cdot \quad X' \notin \Gamma}{\Gamma \vdash \Lambda(\!|\,|\!).M : \forall^{\nu} X'.A; \cdot} \qquad \frac{\Gamma \vdash M : \forall^{\nu} X'.A; \Gamma_M \quad \Gamma, \Gamma_M \vdash B \quad X' \notin \Gamma}{\Gamma \vdash M\ \{(\!|\,|\!) \cong B\} : A; \Gamma_M, X' \cong B}$$

$$\frac{\Gamma, X' \cong A, M \vdash M : B; \cdot \quad X' \notin \Gamma}{\Gamma \vdash \mathsf{pack}^{\nu}((\!|\,|\!) \cong A, M) : \exists^{\nu} X'.B; \cdot}$$

$$\frac{\Gamma \vdash M : \exists^{\nu} X.A; \Gamma_M \quad \Gamma, \Gamma_M, X \vdash N : B; \Gamma_N \quad \Gamma, \Gamma_M, \vdash \Gamma_N \quad \Gamma, \Gamma_M, \Gamma_N \vdash B}{\Gamma \vdash \mathsf{unpack}(X, (\!|\,|\!)) = M; N : B; \Gamma_M, \Gamma_N}$$

$$\frac{\Gamma \vdash M : \exists^{\nu} X'.A; \Gamma_M \quad \Gamma, \Gamma_M, X', x : A \vdash N : B; \Gamma_N \quad \Gamma, \Gamma_M, \vdash \Gamma_N \quad \Gamma, \Gamma_M, \Gamma_N \vdash B \quad X' \notin \Gamma}{\Gamma \vdash \mathsf{unpack}((\!|\,|\!), x) = M; N : B; \Gamma_M, \Gamma_N}$$

$$\frac{\Gamma \vdash M : \exists^{\nu} X'.A; \Gamma_M \quad \Gamma, \Gamma_M, X' \vdash N : B; \Gamma_N \quad \Gamma, \Gamma_M, \vdash \Gamma_N \quad X' \notin \Gamma}{\Gamma \vdash \mathsf{unpack}((\!|\,|\!), (\!|\,|\!)) = M; N : B; \Gamma_M, \Gamma_N}$$

$$\frac{\Gamma \vdash M : A; \Gamma_o \quad X \cong A \in \Gamma, \Gamma_o}{\Gamma \vdash \mathsf{seal}_{(\!|\,|\!)} M : X; \Gamma_o} \qquad \frac{\Gamma \vdash M : X; \Gamma_o \quad X \cong A \in \Gamma, \Gamma_o}{\Gamma \vdash \mathsf{unseal}_{(\!|\,|\!)} M : A; \Gamma_o} \qquad \frac{\Gamma \vdash M : ?; \Gamma_o \quad \Gamma \vdash G}{\Gamma \vdash \mathsf{is}((\!|\,|\!))?M : \mathsf{Bool}; \Gamma_o}$$

$$\frac{\Gamma \vdash M : A; \Gamma_M \quad \Gamma, \Gamma_M \vdash N : B; \Gamma_N}{\Gamma \vdash \mathsf{let}\ (\!|\,|\!) = M; N : B; \Gamma_M, \Gamma_N} \qquad \frac{\Gamma \vdash M : A_1 \times A_2; \Gamma_M \quad \Gamma, \Gamma_M \vdash N : B; \Gamma_N}{\Gamma \vdash \mathsf{let}\ ((\!|\,|\!), (\!|\,|\!)) = M; N : B; \Gamma_M, \Gamma_N}$$

$$\frac{\Gamma \vdash M : A_1 \times A_2; \Gamma_M \quad \Gamma, \Gamma_M, x : A_1 \vdash N : B; \Gamma_N}{\Gamma \vdash \mathsf{let}\ (x, (\!|\,|\!)) = M; N : B; \Gamma_M, \Gamma_N} \qquad \frac{\Gamma \vdash M : A_1 \times A_2; \Gamma_M \quad \Gamma, \Gamma_M, y : A_2 \vdash N : B; \Gamma_N}{\Gamma \vdash \mathsf{let}\ ((\!|\,|\!), y) = M; N : B; \Gamma_M, \Gamma_N}$$

Fig. 16 Syntax and Static Semantics of PolyC$^{\nu}$H

mers' burden, more implicit polymorphic gradual typing as in [3] [15] would be desirable.

One of the languages that are mainly used in this paper is Hazelnut Live, but it is rather a core calculus than a surface language. It is part of Hazel, live functional programming environment that can

$$
\begin{array}{rcll}
(\!|\!|)^+ & = & \mathsf{hide}\ \Gamma_o \subseteq \cdot; (\!|\!|) & (\text{where } \Gamma \vdash (\!|\!|) : ?; \Gamma_o) \\
(\!|M|\!)^+ & = & (\!|\mathsf{hide}\ \Gamma \subseteq \Gamma, \Gamma'; \mathsf{hide}\ \Gamma_o \subseteq \Gamma'_o; M^+|\!) & (\text{where } \Gamma, \Gamma' \vdash M : A; \Gamma_o) \\
(\mathsf{let}\ (\!|\!|) = M; N)^+ & = & \mathsf{let}\ (\!|\!|) = M^+; N^+ & \\
(\mathsf{seal}_{(\!|\!|)} M)^+ & = & \mathsf{seal}_{(\!|\!|)}(M :: X)^+ & (\text{where } M : X) \\
(\mathsf{unseal}_{(\!|\!|)} M)^+ & = & \mathsf{unseal}_{(\!|\!|)}(X \wr M) & \\
 & & & (\text{where } \Gamma \vdash M : A; \Gamma_o,\ A \sim X \text{ and } X \cong B \in \Gamma, \Gamma_o) \\
(\mathsf{is}(\!|\!|)?M)^+ & = & \mathsf{is}(\!|\!|)?(\langle A?^{\sqsubseteq} \rangle \upharpoonright M^+) & (\text{where } M : A \text{ and } A?^{\sqsubseteq} : A \sqsubseteq ?) \\
(\mathsf{let}\ (\!|\!|, y) = M; N) & = & \mathsf{let}(\!|\!|, y) = ? \times ? \wr M; N^+ & \\
(\mathsf{let}\ (x, (\!|\!|)) = M; N) & = & \mathsf{let}(x, (\!|\!|)) = ? \times ? \wr M; N^+ & \\
(\mathsf{let}\ (\!|\!|, (\!|\!|)) = M; N) & = & \mathsf{let}(\!|\!|, (\!|\!|)) = ? \times ? \wr M; N^+ & \\
(\lambda (\!|\!|) : A.M)^+ & = & \lambda (\!|\!|) : A.\mathsf{hide}\ \cdot \subseteq \Gamma_o; M^+ & (\text{where } M : A; \Gamma_o) \\
(M\ N)^+ & = & (? \to ? \wr M)(N :: dom(A))^+ & (\text{where } M : A) \\
(\mathsf{pack}^\nu\ ((\!|\!|) \cong A, M))^+ & = & \mathsf{pack}^\nu((\!|\!|) \cong A, \mathsf{hide}\ \cdot \subseteq \Gamma_o, M^+) & (\text{where } M : B; \Gamma_o) \\
(\mathsf{unpack}\ ((\!|\!|), x) = M; N)^+ & = & \mathsf{unpack}((\!|\!|), x) = \exists^\nu X'.? \wr M; \mathsf{hide}\ \Gamma_{N|X'} \subseteq \Gamma_N; N^+ & (X' \notin \Gamma, \Gamma_o) \\
(\mathsf{unpack}\ (X, (\!|\!|)) = M; N)^+ & = & \mathsf{unpack}(X, (\!|\!|)) = \exists^\nu X.? \wr M; \mathsf{hide}\ \Gamma_{N|X} \subseteq \Gamma_N; N^+ & \\
(\mathsf{unpack}((\!|\!|), (\!|\!|)) = M; N)^+ & = & \mathsf{unpack}((\!|\!|), (\!|\!|)) = \exists^\nu X'.? \wr M; \mathsf{hide}\ \Gamma_{N|X'} \subseteq \Gamma_N; N^+ & (X' \notin \Gamma, \Gamma_o) \\
\Lambda^\nu (\!|\!|).M^+ & = & \Lambda^\nu (\!|\!|).\mathsf{hide}\ \cdot \subseteq \Gamma_o; M^+ & \\
M\{(\!|\!|) \cong B\}^+ & = & (\forall^\nu X'.? \wr M)\{(\!|\!|) \cong B\} & (X' \notin \Gamma, \Gamma_o) \\
 & & & \\
G \wr M & = & \langle \mathsf{tag}_G(G) \rangle \downarrow M^+ & (\text{when } M : ?) \\
G \wr M & = & M^+ & (\text{otherwise}) \\
 & & & \\
\mathsf{hide}\ \Gamma_s \subseteq (\Gamma_b, X \cong A); M & = & \mathsf{hide}\ \Gamma_s \subseteq \Gamma_b; \mathsf{hide} X \cong A; M & (X \notin \Gamma_s) \\
\mathsf{hide}\ \Gamma_s \subseteq (\Gamma_b, X); M & = & \mathsf{hide}\ \Gamma_s \subseteq \Gamma_b; \mathsf{hide}\ X \cong ?; M & (X \notin \Gamma_s) \\
\mathsf{hide}\ \Gamma_s \subseteq (\Gamma_b, x : A); M & = & \mathsf{hide}\ \Gamma_s \subseteq \Gamma_b; \mathsf{let}\ x = (\!|\!|); M & (x \notin \Gamma_s) \\
\mathsf{hide}\ (\Gamma_s, X \cong A) \subseteq \Gamma_b; M & = & \mathsf{hide}\ \Gamma_s \subseteq \Gamma_b; \mathsf{let}\ x' = (\Lambda^\nu X.\mathsf{true})\{X \cong A\}; M & \\
 & & & (X \notin \Gamma_b \text{ and } x' \notin \Gamma, \Gamma_o)
\end{array}
$$

Fig. 17 Elaboration from PolyG$^\nu$H to PolyC$^\nu$H

type-check and run programs with holes. Hazel is based on two core calculi, Hazelnut [8] and Hazelnut Live [7]. Hazelnut defines edit actions that automatically insert holes and ensure every editor state has static meaning. Hazelnut Live provides dynamic semantics to those programs to inform programmers how their (incomplete) program will work. As future work, we may also consider edit actions for our polymorphic gradual language, after which we may be able to extend Hazel with parametric polymorphism. The biggest challenge would be how to determine the environments around non-empty holes, which at preset are just assumed to be given like oracles.

Acknowledgements We thank Prof. Igarashi for information about the state-of-the-art of polymorphc gradual typing including PolyG$^\nu$, and Dr. New for e-mail communication about details of PolyG$^\nu$. This work was partially supported by JSPS KAKENHI Grant Numbers JP15H02681 and JP20H04161.

$$
\begin{array}{lrcll}
\text{Finals} & F & ::= & V \mid I & \\
\text{Indets} & I & ::= & (\!|\!|) \mid (\!|F|\!) \mid \langle G \Leftarrow ? \Leftarrow H \rangle F & \\
& & \mid & \mathsf{is}(G)?\, F & \text{where } F \neq \mathsf{inj}_H F \\
& & \mid & \lambda(\!|\!| : A.\ M \mid \Lambda^\nu (\!|\!|.\ M \mid M\{(\!|\!| \cong A\} & \\
& & \mid & \mathsf{pack}^\nu((\!|\!| \cong A, M) \mid \mathsf{unpack}((\!|\!|, (\!|\!|) = M; N & \\
& & \mid & \mathsf{unpack}(X, (\!|\!|) = M; N \mid \mathsf{unpack}((\!|\!|, x) = M; N & \\
& & \mid & \mathsf{seal}_{(\!|\!|} M \mid \mathsf{unseal}_{(\!|\!|} M \mid \mathsf{is}((\!|\!|)?M & \\
& & \mid & \mathsf{let}\ (\!|\!| = M; N \mid \mathsf{let}\ ((\!|\!|, (\!|\!|) = M; N & \\
& & \mid & \mathsf{let}\ (x, (\!|\!|) = M; N \mid \mathsf{let}\ ((\!|\!|, y) = M; N & \\
& & \mid & \mathsf{seal}_\alpha I \mid (I, F) \mid (V, I) \mid \mathsf{inj}_G I & \\
& & \mid & \mathsf{if}\ I\ \mathsf{then}\ M\ \mathsf{else}\ M & \\
& & \mid & \mathsf{unseal}_\alpha I & \text{where } I \neq \mathsf{seal}_\alpha M \\
& & \mid & I\ F & \text{where } I \neq \langle A_1^{\sqsubseteq} \to A_2^{\sqsubseteq} \rangle \updownarrow M \\
& & \mid & I\ \{X \cong A\} & \text{where } I \neq \langle \forall^\nu X.A^{\sqsubseteq} \rangle \updownarrow M \\
& & \mid & \mathsf{unpack}(X, x) = I; M & \\
& & \mid & \mathsf{let}(x, x) = I; M & \text{where } I \neq (M, M) \\
& & \mid & \langle A_1^{\sqsubseteq} \to A_2^{\sqsubseteq} \rangle \updownarrow I \mid \langle \forall^\nu X.A^{\sqsubseteq} \rangle \updownarrow I \mid \langle \exists^\nu X.A^{\sqsubseteq} \rangle \updownarrow I & \\
& & \mid & \langle A_1^{\sqsubseteq} \times A_2^{\sqsubseteq} \rangle \updownarrow I & \text{where } I \neq (M, M) \\
& & \mid & \langle \mathsf{tag}_G(A^{\sqsubseteq}) \rangle \downarrow I & \text{where } I \neq \mathsf{inj}_{G'} M \\
& & \mid & \mathsf{is}(G)?\, I & \text{where } I \neq \mathsf{inj}_{G'} M
\end{array}
$$

Fig. 18 Final Forms of PolyC$^\nu$H

Evaluation Context

$$E \quad + ::= \quad (\!|E|\!) \mid \langle G \Leftarrow ? \Leftarrow H \rangle E$$

Dynamic Semantics

$$E[\langle \mathsf{tag}_G(A^{\sqsubseteq}) \rangle \downarrow \mathsf{inj}_H F] \quad \mapsto \quad E[\langle A^{\sqsubseteq} \rangle \downarrow \langle G \Leftarrow ? \Leftarrow H \rangle F]$$

Fig. 19 Dynamic Semantics of PolyC$^\nu$H

参 考 文 献

[1] : Hazel, a live functional programming environment featuring typed holes. `https://hazel.org/`.

[2] Ahmed, A., Findler, R. B., Siek, J. G., and Wadler, P.: Blame for all, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011*, 2011, pp. 201–214.

[3] Garcia, R. and Cimini, M.: Principal Type Schemes for Gradual Programs, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Rajamani, S. K. and Walker, D.(eds.), ACM, 2015, pp. 303–315.

[4] Igarashi, Y., Sekiyama, T., and Igarashi, A.: On polymorphic gradual typing, *PACMPL*, Vol. 1, No. ICFP(2017), pp. 40:1–40:29.

[5] Morris Jr., J. H.: Protection in Programming Languages, *Commun. ACM*, Vol. 16, No. 1(1973), pp. 15–21.

[6] New, M. S., Jamner, D., and Ahmed, A.: Graduality and parametricity: together again for the first time, *PACMPL*, Vol. 4, No. POPL(2020), pp. 46:1–46:32.

[7] Omar, C., Voysey, I., Chugh, R., and Hammer, M. A.: Live functional programming with typed holes, *PACMPL*, Vol. 3, No. POPL(2019), pp. 14:1–14:32.

[8] Omar, C., Voysey, I., Hilton, M., Aldrich, J., and Hammer, M. A.: Hazelnut: A Bidirectionally Typed Structure Editor Calculus, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, New York, NY,

USA, Association for Computing Machinery, 2017, pp. 86–99.

[9] Pierce, B. and Sumii, E.: Relating Cryptography and Polymorphism, 2000. Manuscript.

[10] Pierce, B. C. and Turner, D. N.: Local Type Inference, *ACM Trans. Program. Lang. Syst.*, Vol. 22, No. 1(2000), pp. 1–44.

[11] Siek, J. and Taha, W.: Gradual typing for functional languages, *Scheme and Functional Programming Workshop*, (2006), pp. 81–92.

[12] Siek, J. G., Vitousek, M. M., Cimini, M., and Boyland, J. T.: Refined Criteria for Gradual Typing, *1st Summit on Advances in Programming Languages (SNAPL 2015)*, 2015, pp. 274–293.

[13] Sumii, E. and Pierce, B. C.: A bisimulation for dynamic sealing, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004,*, 2004, pp. 161–172.

[14] Toro, M., Labrada, E., and Tanter, É.: Gradual parametricity, revisited, *PACMPL*, Vol. 3, No. POPL(2019), pp. 17:1–17:30.

[15] Xie, N., Bi, X., and d. S. Oliveira, B. C.: Consistent Subtyping for All, *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, Ahmed, A.(ed.), Lecture Notes in Computer Science, Vol. 10801, Springer, 2018, pp. 3–30.