Operational Semantics $\boxed{e \hookrightarrow e}$

$$\frac{v \models \phi}{\square : [v : b \mid \phi] \hookrightarrow v}\ \textsc{EHole} \qquad \frac{op\ \overline{v} \equiv v_y}{\texttt{let}\ y = op\ \overline{v}\ \texttt{in}\ e \hookrightarrow e[y \mapsto v_y]}\ \textsc{EAppOp}$$

$$\frac{e_1 \hookrightarrow e_1'}{\texttt{let}\ y = e_1\ \texttt{in}\ e_2 \hookrightarrow \texttt{let}\ y = e_1'\ \texttt{in}\ e_2}\ \textsc{ELetE1} \qquad \frac{}{\texttt{let}\ y = v\ \texttt{in}\ e \hookrightarrow e[y \mapsto v]}\ \textsc{ELetE2}$$

$$\frac{}{\texttt{let}\ y = \lambda x{:}t.e_1\ v_x\ \texttt{in}\ e_2 \hookrightarrow \texttt{let}\ y = e_1[x \mapsto v_x]\ \texttt{in}\ e_2}\ \textsc{ELetAppLam}$$

$$\frac{}{\texttt{let}\ y = \mathsf{fix} f{:}t.\lambda x{:}t_x.e_1\ v_x\ \texttt{in}\ e_2 \hookrightarrow \texttt{let}\ y = (\lambda f{:}t.e_1[x \mapsto v_x])\ (\mathsf{fix} f{:}t.\lambda x{:}t_x.e_1)\ \texttt{in}\ e_2}\ \textsc{ELetAppFix}$$

$$\frac{}{\texttt{match}\ d_i\ \overline{v_j}\ \texttt{with}\ \overline{d_i\ \overline{y_j} \to e_i} \hookrightarrow e_i[\overline{y_j \mapsto v_j}]}\ \textsc{EMatch}$$

Fig. 15. Small Step Operational Semantics of $\boldsymbol{\lambda}^{TG}$++.

A Operational Semantics

Fig. 15 give the reduction rules for $\boldsymbol{\lambda}^{TG}$++'s small standard operational semantics.

Well-Formedness $\boxed{\Gamma \vdash^{\mathbf{WF}} \tau}$

$$\frac{\begin{array}{c}\Gamma \equiv \overline{x_i{:}\{v:b_{x_i} \mid \phi_{x_i}\}, y_j{:}[v:b_{y_j} \mid \phi_{y_j}], z{:}(a{:}\tau_a \to \tau_b)} \\ (\overline{\forall x_i{:}b_{x_i}, \exists y_j{:}b_{y_i}}, \forall v{:}b, \phi) \text{ is a Boolean predicate} \quad \forall j, \texttt{err} \notin [\![[v:b_{y_j} \mid \phi_{y_j}]]\!]_\Gamma\end{array}}{\Gamma \vdash^{\mathbf{WF}} [v:b \mid \phi]} \textsc{WfBase}$$

$$\frac{\Gamma, x{:}\{v:b \mid \phi\} \vdash^{\mathbf{WF}} \tau}{\Gamma \vdash^{\mathbf{WF}} x{:}\{v:b \mid \phi\} \to \tau} \textsc{WfArg} \qquad \frac{\Gamma \vdash^{\mathbf{WF}} (a{:}\tau_a \to \tau_b) \qquad \Gamma \vdash^{\mathbf{WF}} \tau}{\Gamma \vdash^{\mathbf{WF}} (a{:}\tau_a \to \tau_b) \to \tau} \textsc{WfRes}$$

Subtyping $\boxed{\Gamma \vdash \tau_1 <: \tau_2}$

$$\frac{[\![[v:b \mid \phi_1]]\!]_\Gamma \subseteq [\![[v:b \mid \phi_2]]\!]_\Gamma}{\Gamma \vdash [v:b \mid \phi_1] <: [v:b \mid \phi_2]} \textsc{SubUBase} \qquad \frac{\begin{array}{l}[\![\{v:b \mid \phi_1\}]\!]_\Gamma \subseteq \\ [\![\{v:b \mid \phi_2\}]\!]_\Gamma\end{array}}{\Gamma \vdash \{v:b \mid \phi_1\} <: \{v:b \mid \phi_2\}} \textsc{SubOBase}$$

$$\frac{\Gamma \vdash \tau_{21} <: \tau_{11} \qquad \Gamma, x{:}\tau_{21} \vdash \tau_{12} <: \tau_{22}}{\Gamma \vdash x{:}\tau_{11} \to \tau_{12} <: x{:}\tau_{21} \to \tau_{22}} \textsc{SubArr}$$

Disjunction $\boxed{\Gamma \vdash \tau_1 \vee \tau_2 = \tau_3}$

$$\frac{[\![\tau_1]\!]_\Gamma \cap [\![\tau_2]\!]_\Gamma = [\![\tau_3]\!]_\Gamma}{\Gamma \vdash \tau_1 \vee \tau_2 = \tau_3} \textsc{Disjunction}$$

Fig. 16. Auxillary typing relations

B Type System

The full set of typing rules for $\boldsymbol{\lambda}^{TG}$++ is shown in Fig. 17.

B.1 Subset Relation of Denotation under Type Context

We define the subset relation between the denotation of two refinement types τ_1 and τ_2 under a type context Γ (written $[\![\tau_1]\!]_\Gamma \subseteq [\![\tau_1]\!]_\Gamma$) as:

$$\begin{aligned}
[\![\tau_1]\!]_\emptyset \subseteq [\![\tau_2]\!]_\emptyset &\doteq [\![\tau_1]\!] \subseteq [\![\tau_2]\!] \\
[\![\tau_1]\!]_{x:\tau_x,\Gamma} \subseteq [\![\tau_1]\!]_{x:\tau_x,\Gamma} &\doteq \forall v_x \in [\![\tau_x]\!], \\
&\qquad [\![\tau_1[x \mapsto v_x]]\!]_{\Gamma[x \mapsto v_x]} \subseteq [\![\tau_2[x \mapsto v_x]]\!]_{\Gamma[x \mapsto v_x]} \quad \text{if } \tau \equiv \{v:b \mid \phi\} \\
[\![\tau_1]\!]_{x:\tau_x\Gamma} \subseteq [\![\tau_2]\!]_{x:\tau_x\Gamma} &\doteq \exists \hat{e}_x \in [\![\tau_x]\!], \forall v_x, \hat{e}_x \hookrightarrow^* v_x \implies \\
&\qquad [\![\tau_1[x \mapsto v_x]]\!]_{\Gamma[x \mapsto v_x]} \subseteq [\![\tau_2[x \mapsto v_x]]\!]_{\Gamma[x \mapsto v_x]} \quad \text{otherwise}
\end{aligned}$$

The way we interpret the type context Γ here is the same as the definition of the type denotation under the type context, but we keep the denotation of τ_1 and τ_2 as the subset relation under the same interpretation of Γ, that is under the *same* substitution $[x \mapsto v_x]$. This constraint is also required by other refinement type systems, which define the denotation of the type context Γ as a set of substitutions, with the subset relation of the denotation of two types holding under the *same* substitution. However, our type context is more complicated, since it has both under- and overapproximate types. The denotations of these types use both existential and universal quantifiers, and cannot simply be interpreted as a set of substitutions. Thus, we define a subset relation over denotations under a type context to ensure the same substitution is applied to both types.

Typing $\boxed{\Gamma \vdash e : \tau}$

$$\frac{\Gamma \vdash^{\mathbf{WF}} [v{:}b \mid \phi]}{\Gamma \vdash \Box \;:\; [v{:}b \mid \phi] : [v{:}b \mid \phi]}\;\textsc{THole} \qquad \frac{\Gamma \vdash^{\mathbf{WF}} [v{:}b \mid \bot]}{\Gamma \vdash \mathsf{err} : [v{:}b \mid \bot]}\;\textsc{TErr}$$

$$\frac{\Gamma \vdash^{\mathbf{WF}} \mathsf{Ty}(c)}{\Gamma \vdash c : \mathsf{Ty}(c)}\;\textsc{TConst} \qquad \frac{\Gamma \vdash^{\mathbf{WF}} \mathsf{Ty}(op)}{\Gamma \vdash op : \mathsf{Ty}(op)}\;\textsc{TOp}$$

$$\frac{\Gamma \vdash^{\mathbf{WF}} [v{:}b \mid v = x]}{\Gamma \vdash x : [v{:}b \mid v = x]}\;\textsc{TVarBase} \qquad \frac{\Gamma(x) = (a{:}\tau_a \to \tau_b) \quad \Gamma \vdash^{\mathbf{WF}} a{:}\tau_a \to \tau_b}{\Gamma \vdash x : (a{:}\tau_a \to \tau_b)}\;\textsc{TVarFun}$$

$$\frac{\Gamma, x{:}\tau_x \vdash e : \tau \quad \Gamma \vdash^{\mathbf{WF}} x{:}\tau_x \to \tau}{\Gamma \vdash \lambda x{:}\lfloor \tau_x \rfloor.e : (x{:}\tau_x \to \tau)}\;\textsc{TFun}$$

$$\frac{\Gamma \vdash \lambda x{:}b.\lambda f{:}(b \to \lfloor \tau \rfloor).e : (x{:}\{v{:}b \mid \phi\} \to f{:}(x{:}\{v{:}b \mid v \prec x \land \phi\} \to \tau) \to \tau) \quad \Gamma \vdash^{\mathbf{WF}} x{:}\{v{:}b \mid \phi\} \to \tau}{\Gamma \vdash \mathsf{fix} f{:}(b \to \lfloor \tau \rfloor).\lambda x{:}b.e : (x{:}\{v{:}b \mid \phi\} \to \tau)}\;\textsc{TFix}$$

$$\frac{\begin{array}{c}\emptyset \vdash \tau <: \tau' \quad \emptyset \vdash e : \tau \\ \Gamma \vdash^{\mathbf{WF}} \tau'\end{array}}{\Gamma \vdash e : \tau'}\;\textsc{TSub} \qquad \frac{\begin{array}{c}\Gamma \vdash \tau' <: \tau \quad \Gamma \vdash \tau <: \tau' \\ \Gamma \vdash e : \tau \quad \Gamma \vdash^{\mathbf{WF}} \tau'\end{array}}{\Gamma \vdash e : \tau'}\;\textsc{TEq}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau_1 \quad \Gamma \vdash e : \tau_2 \\ \Gamma \vdash \tau_1 \lor \tau_2 = \tau \quad \Gamma \vdash^{\mathbf{WF}} \tau\end{array}}{\Gamma \vdash e : \tau}\;\textsc{TMerge} \qquad \frac{\begin{array}{c}\Gamma \vdash e_x : \tau_x \quad \Gamma, x{:}\tau_x \vdash e : \tau \\ \Gamma \vdash^{\mathbf{WF}} \tau\end{array}}{\Gamma \vdash \mathtt{let}\; x = e_x \;\mathtt{in}\; e : \tau}\;\textsc{TLetE}$$

$$\frac{\begin{array}{c}\Gamma \vdash op : \overline{a_i{:}\{v{:}b_i \mid \phi_i\}} \to \tau_x \\ \forall i, \Gamma \vdash v_i : [v{:}b_i \mid [\phi_i]] \\ \Gamma, x{:}\tau_x[\overline{a_i \mapsto v_i}] \vdash e : \tau \\ \Gamma \vdash^{\mathbf{WF}} \tau\end{array}}{\Gamma \vdash \mathtt{let}\; x = op\; \overline{v_i} \;\mathtt{in}\; e : \tau}\;\textsc{TAppOp} \qquad \frac{\begin{array}{c}\Gamma \vdash v_1 : (\tau_1 \to \tau_2) \to \tau_x \\ \Gamma \vdash v_2 : \tau_1 \to \tau_2 \quad \Gamma, x{:}\tau_x \vdash e : \tau \\ \Gamma \vdash^{\mathbf{WF}} \tau\end{array}}{\Gamma \vdash \mathtt{let}\; x = v_1\; v_2 \;\mathtt{in}\; e : \tau}\;\textsc{TAppFun}$$

$$\frac{\begin{array}{c}\Gamma \vdash v_1 : a{:}\{v{:}b \mid \phi\} \to \tau_x \\ \Gamma \vdash v_2 : [v{:}b \mid \phi] \\ \Gamma, x{:}\tau_x[a \mapsto v_2] \vdash e : \tau \quad \Gamma \vdash^{\mathbf{WF}} \tau\end{array}}{\Gamma \vdash \mathtt{let}\; x = v_1\; v_2 \;\mathtt{in}\; e : \tau}\;\textsc{TApp} \qquad \frac{\begin{array}{c}\Gamma \vdash v : \tau_v \quad \Gamma \vdash^{\mathbf{WF}} \tau \quad \Gamma, \overline{y{:}\tau_y} \vdash d_i(\overline{y}) : \tau_v \\ \Gamma, \overline{y{:}\tau_y} \vdash e_i : \tau\end{array}}{\Gamma \vdash (\mathtt{match}\; v \;\mathtt{with}\; \overline{d_i\; \overline{y} \to e_i}) : \tau}\;\textsc{TMatch}$$

Fig. 17. Full Typing Rules

C Abduction Algorithm

Algorithm 3 lists the Abduce subroutine that Repair uses to infer ψ_{need}, the qualifier of the type we use to capture an input generator's missing coverage. The first argument, Γ, is the typing context of the body of the generator. The final two arguments of Abduce include the current and target coverage of the test input generator, ψ_{cur} and ψ respectively. Given these inputs, the goal of Abduce is to output the weakest formula ψ_{need} in the hypothesis space that ensures $\Gamma \vdash [v{:}b \mid \phi_{\mathsf{cur}} \lor \phi_{\mathsf{need}}] <: [v{:}b \mid \phi]$.

Abduce does so by adapting an existing algorithm for inferring a maximally weak specification in the context of safety verification [65] to our coverage type setting.

Algorithm 3: Inferring missing coverage (Abduce)

Inputs : Φ: a set of atomic formulas, Γ: typing context, $[v:b \mid \psi_{\mathsf{cur}}]$: current coverage, $[v:b \mid \psi]$: target coverage

Output : Formula ψ_{need} such that $\Gamma \vdash [v:b \mid \psi_{\mathsf{cur}} \vee \psi_{\mathsf{need}}] <: [v:b \mid \psi]$

1 **if** $\Gamma \vdash [v:b \mid \phi_{\mathsf{cur}}] <: [v:b \mid \phi]$ **then return** $\perp$;

2 $\Psi \leftarrow \bigcup_{\phi \subseteq \Phi} \{\psi \mid \psi = \bigwedge_{\alpha \in \phi} \alpha \quad \bigwedge_{\alpha \in \Phi - \phi} \neg\alpha\}$;

3 $\Psi^- \leftarrow \{\psi \in \Psi \mid \psi_{\mathsf{cur}} \implies \psi\}$; $\Psi^? \leftarrow \Psi - \Psi^-$; $\Psi^+ \leftarrow \emptyset$; $\psi_{\mathsf{need}} \leftarrow \top$;

4 **while** $\exists \psi_? \in \Psi^?$ **do**

5     $\Psi^? \leftarrow \Psi^? \setminus \{\psi_?\}$;

6     $\psi_{\mathsf{need}} \leftarrow \mathsf{Learn}(\Psi^+ \cup \Psi^?, \Psi^- \cup \{\psi_?\})$;

7   **if** $\Gamma \vdash [v:b \mid \psi_{\mathsf{cur}} \vee \psi_{\mathsf{need}}] <: [v:b \mid \psi]$ **then**

8         $\Psi^- \leftarrow \Psi^- \cup \{\psi_?\}$;

9   **else**

10         $\Psi^+ \leftarrow \Psi^+ \cup \{\psi_?\}$;

11         $\psi_{\mathsf{need}} \leftarrow \mathsf{Learn}(\Psi^+, \Psi^- \cup \Psi^?)$;

12   **if** $\Gamma \vdash [v:b \mid \psi_{\mathsf{cur}} \vee \psi_{\mathsf{need}}] <: [v:b \mid \psi]$ **then**

13   **return** ψ_{need};

14 **return** ψ_{need};

This algorithm maintains three disjoint sets, Ψ^-, Ψ^+, and $\Psi^?$, which cumulatively contain all conjunctions of the method predicates in P and their negations (line 2). Intuitively, each element of these sets is a formula that defines a distinct subset of the generator's outputs: the formula $\neg\mathsf{empty}(l) \wedge \mathsf{hd}(l) = 1$, for example, characterizes all nonempty lists that begin with 1. The set Ψ^+ captures values that the input generator does not output but needs to, Ψ^- includes values that can safely omitted from the output of the repaired generator, and $\Psi^?$ includes those values which have not been definitively placed into either category. Abduce initializes these sets by moving all values currently covered by ψ_{cur} into Ψ^- and placing the remaining elements into $\Psi^?$ (line 3). The candidate solution maintained by Abduce, ψ_{need}, contains all the elements of Ψ^+ and $\Psi^?$. The key invariant of Abduce is that the disjunction of ψ_{need} and ψ_{cur} is always a subtype of ψ, i.e., ψ_{need} captures a superset of the outputs that need to be added to a generator.

The algorithm's main loop (lines 4-13) attempts to place all the members of $\Psi^?$ into either Ψ^- or Ψ^+. Each iteration of the loop checks if it is safe to move an element of $\Psi^?$, $\psi_?$, to Ψ^-, adding it to Ψ^+ if not. The loop first uses an auxiliary function, Learn, to construct a candidate solution, ψ_{need}, that distinguishes Ψ^+ and $\Psi^?$ from those of $\Psi^- \cup \{\psi_?\}$ (line 5). The loop then uses a subtype check to see whether ψ_{need} is still sufficient to complete ϕ_{cur} (line 7), updating Ψ^- if so (line 9). If not, $\psi_?$ is added to Ψ^+ and we check to see if all the remaining elements of $\Psi^?$ can be safely moved to Ψ^-, terminating if so (lines 11-13). If not, the loop continues until $\Psi^?$ has no more elements.

Algorithm 4: Insert repair locations (Localize)

Inputs : s: incomplete generator, Γ: typing context, $[v{:}b \mid \phi_{\mathsf{need}}]$: missing coverage

Output : An updated sketch i' containing j holes and a set of typing contexts for each hole $\overline{\Gamma_j \vdash \square_j \ :\ [v{:}b \mid \phi_j]}$

1 **match** s **with**

2 $v \Rightarrow$ **return** $(v \oplus \square\ :\ [v{:}b \mid \phi_{\mathsf{need}}], \{\Gamma \vdash \square\ :\ [v{:}b \mid \phi_{\mathsf{need}}]\})$;

3 $\mid$ err $\Rightarrow$ **return** $(\square\ :\ [v{:}b \mid \phi_{\mathsf{need}}], \{\Gamma \vdash \square\ :\ [v{:}b \mid \phi_{\mathsf{need}}]\})$;

4 $\mid \square\ :\ [v{:}b \mid \phi] \Rightarrow$ **return** $(\square, \{\Gamma \vdash \square\ :\ [v{:}b \mid \phi], \Gamma \vdash \square\ :\ [v{:}b \mid \phi_{\mathsf{need}}]\})$;

5 $\mid \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \Rightarrow$

6 $(i_2, \Gamma') \leftarrow \mathtt{Localize}(\Gamma; x : \mathtt{TyInfer}(\Gamma, e_1), e_2, [v{:}b \mid \phi_{\mathsf{need}}])$;

7 **return** $(\mathtt{let}\ x = c_1\ \mathtt{in}\ i_2, \Gamma')$

8 $\mid \mathtt{let}\ x = op\ \overline{v}\ \mathtt{in}\ e_2 \Rightarrow$

9 $(i_2, \Gamma') \leftarrow \mathtt{Localize}(\Gamma; x : \mathtt{TyInfer}(\Gamma, op\ \overline{v}), e_2, [v{:}b \mid \phi_{\mathsf{need}}])$;

10 **return** $(\mathtt{let}\ x = op\ \overline{v}\ \mathtt{in}\ i_2, \Gamma')$

11 $\mid \mathtt{let}\ x = v\ v\ \mathtt{in}\ e_2 \Rightarrow$

12 $(i_2, \Gamma') \leftarrow \mathtt{Localize}(\Gamma; x : \mathtt{TyInfer}(\Gamma, v\ v), e_2, [v{:}b \mid \phi_{\mathsf{need}}])$;

13 **return** $(\mathtt{let}\ x = v\ v\ \mathtt{in}\ i_2, \Gamma')$

14 $\mid \mathtt{match}\ v\ \mathtt{with}\ \overline{d_k\ \overline{y_k} \longrightarrow e_k} \Rightarrow$

15 $\Gamma' \leftarrow \emptyset$;

16 **for** $m \in \{0, \ldots, \mathrm{k}\}$ **do**

17 $\overline{y{:}\{v{:}b \mid \phi\}} \rightarrow [v{:}b_m \mid \psi_m] \leftarrow \mathsf{Ty}(d_m)$;

18 $\Gamma'_j \leftarrow \overline{y{:}\{v{:}b \mid \phi\}},\ a{:}[v{:}b_m \mid v = v \land \psi_m]$;

19 $(i_m, \Gamma_m) \leftarrow \mathtt{Localize}(\Gamma; \Gamma'_j, e_j, [v{:}b \mid \phi_{\mathsf{need}}])$;

20 $\Gamma' \leftarrow \Gamma' \cup \Gamma_m$;

21 **return** $(\mathtt{match}\ v\ \mathtt{with}\ \overline{d_k\ \overline{y_k} \longrightarrow i_k},\ \Gamma')$

D Localization Algorithm

Starting from an initial typing context, Γ, Localize recurses over the AST of the input program (line 1). In the base cases, a hole is inserted and the current typing context is attached to it (lines 2-5). Localize replaces err expressions with a new hole, since errors never contribute any coverage (line 3). If there is already a hole, Localize adds an additional hole with the target coverage, and adds both holes to the set of repair locations. In the recursive cases, Γ is updated according to the typing context used for each subterm in the corresponding typing rule: the recursive call on line 10, for example, extends the input typing context with a binding for the result of $op\ \overline{v}$. When applied to a match expression, Algorithm 4 recursively inserts holes into each branch (line 18).

E Case Study: Well-Typed Lambda Calculus Terms

Table 2. Results for the Simply Typed Lambda Calculus case study.

name	#Holes	Repair Size	#Queries	#Terms	Abduction Time(s)	Synthesis Time(s)	Total Time(s)
STLC 1	1	9	99	9	7.34	8.16	109.6
STLC 3	1	30	694	276	3.06	160.34	165.35
STLC 2	1	59	4443	1555	5.96	2760.8	2778.68