A Operational Semantics

Operational Semantics

$\boxed{e \hookrightarrow e}$

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

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

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

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

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

Fig. 15. Small Step Operational Semantics of λ^{TG}++.

Fig. 15 give the reduction rules for λ^{TG}++'s small standard operational semantics.

B Type System

Well-Formedness

$\boxed{\Gamma \vdash^{\mathbf{WF}} \tau}$

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

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

Subtyping

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

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

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

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

The full set of typing rules for λ^{TG}++ is shown in Fig. 17.

B.1 Type Denotations

Assuming a standard typing judgement for basic types, $\emptyset \vdash_t e : t$, a type denotation for a type τ, $[\![\tau]\!]$, is a set of closed expressions:

$$\begin{aligned}
[\![\{v:b \mid \phi\}]\!] &\equiv \{v \mid \emptyset \vdash_t v : b \wedge \phi[v \mapsto v]\} \\
[\![[v:b \mid \phi]]\!] &\equiv \{e \mid \emptyset \vdash_t e : b \wedge \forall v{:}b, \phi[v \mapsto v] \implies e \hookrightarrow^* v\} \\
[\![x{:}\tau_x \to \tau]\!] &\equiv \{f \mid \emptyset \vdash_t f : \lfloor \tau_x \to \tau \rfloor \wedge \forall v_x \in [\![\tau_x]\!] \implies f\ v_x \in [\![\tau[x \mapsto v_x]]\!]\}
\end{aligned}$$

In the case of an overapproximate refinement type, $\{v:b \mid \phi\}$, the denotation is simply the set of all values of type b whose elements satisfy the type's refinement predicate (ϕ), when substituted for all free occurrences of v in ϕ.[12] Dually, the denotation of an underapproximate coverage type is the set of expressions that evaluate to v whenever $\phi[v \mapsto v]$ holds, where ϕ is the type's refinement predicate. Thus, every expression in such a denotation serves as a witness to a feasible, type-correct, execution. The denotation for a function type is defined in terms of the denotations of the function's argument and result in the usual way, ensuring that our type denotation is a logical predicate.

[12]The denotation of an overapproximate refinement type is more generally $\{e{:}b \mid \emptyset \vdash e : b \wedge \forall v{:}b, e \hookrightarrow^* v \implies \phi[x \mapsto v]\}$. However, because such types are only used for function parameters, and our language syntax only admits values as arguments, our denotation uses the simpler form.

Typing

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

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

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

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

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

$$\frac{\Gamma,\ x{:}\{v:b \mid \phi\},\ f{:}x{:}\{v:b \mid v \prec x \ \wedge \ \phi\} \to \tau \ \vdash \ e : \tau \quad \Gamma \vdash^{\mathbf{WF}} x{:}\{v:b \mid \phi\} \to \tau}{\Gamma \vdash \mathtt{fix}\ f(x{:}b) : \tau \coloneqq e \ : \ x{:}\{v:b \mid \phi\} \to \tau}\ \textsc{TFix}$$

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

$$\frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash e : \tau_2 \quad \Gamma \vdash \tau_1 \vee \tau_2 = \tau \quad \Gamma \vdash^{\mathbf{WF}} \tau}{\Gamma \vdash e : \tau}\ \textsc{TMerge} \qquad \frac{\Gamma \vdash e_x : \tau_x \quad \Gamma, x{:}\tau_x \vdash e : \tau \quad \Gamma \vdash^{\mathbf{WF}} \tau}{\Gamma \vdash \mathtt{let}\ x \coloneqq e_x\ \mathtt{in}\ e : \tau}\ \textsc{TLetE}$$

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

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

Fig. 17. Full Typing Rules

The denotation of a refinement type τ under a type context Γ (written $[\![\tau]\!]_\Gamma$) is:[13],

$$[\![\tau]\!]_\emptyset \equiv [\![\tau]\!]$$

$$[\![\tau]\!]_{x:\tau_x,\Gamma} \equiv \{e \mid \forall v_x \in [\![\tau_x]\!].\mathtt{let}\ x \coloneqq v_x\ \mathtt{in}\ e \in [\![\tau[x \mapsto v_x]]\!]_{\Gamma[x \mapsto v_x]}\} \qquad \text{if } \tau \equiv \{v:b \mid \phi\}$$

$$[\![\tau]\!]_{x:\tau_x,\Gamma} \equiv \{e \mid \exists \hat{e}_x \in [\![\tau_x]\!].\forall e_x \in [\![\tau_x]\!].\ \mathtt{let}\ x \coloneqq e_x\ \mathtt{in}\ e \in \bigcap_{\hat{e}_x \hookrightarrow^* v_x} [\![\tau[x \mapsto v_x]]\!]_{\Gamma[x \mapsto v_x]}\} \qquad \text{otherwise}$$

[13] In the last case, since $\hat{e}_x$ may nondeterministically reduce to multiple values, we employ intersection (not union), similar to the Disjunction rule.

The denotation of an overapproximate refinement type under a type context is mostly unsurprising, other than our presentation choice to use a let-binding, rather than substitution, to construct the expressions included in the denotations.

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:

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

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.

Theorem B.1 (Type Soundness [70]). *A well-typed test generator of type* $\vdash f : \overline{x_i : \{\nu : b_i \mid \phi_i\}} \to [\nu : b \mid \phi]$*, when applied to well-typed arguments* $\overline{\vdash v_i : \{\nu : b_i \mid \phi_i\}}$*, can evaluate every value satisfying* $\phi[\overline{x_i \mapsto v_i}] : \forall v.\ \phi[\overline{x_i \mapsto v_i}, \nu \mapsto v] \implies f\ \overline{v_i} \hookrightarrow^* v$

Proof. We proceed by specializing the general type soundness theorem for coverage types [70] to our setting. The general theorem states that for any type context Γ, term e, and coverage type τ, if $\Gamma \vdash e : \tau$, then $e \in [\![\tau]\!]_\Gamma$.

In our case, the generator f is well-typed in the empty context:

$$\vdash f : \overline{x_i : \{\nu : b_i \mid \phi_i\}} \to [\nu : b \mid \phi]$$

and each argument v_i is well-typed:

$$\forall i.\ \vdash v_i : \{\nu : b_i \mid \phi_i\}$$

By the soundness theorem, this means:

$$
\begin{aligned}
&f \in [\![\overline{x_i : \{\nu : b_i \mid \phi_i\}} \to [\nu : b \mid \phi]]\!] \\
&\forall i.\ v_i \in [\![\{\nu : b_i \mid \phi_i\}]\!]
\end{aligned}
$$

By the definition of the denotation of function types, we have:

$$\forall \overline{v_i} \in [\![\{\nu : b_i \mid \phi_i\}]\!].\quad f\ \overline{v_i} \in [\![[\nu : b \mid \phi][\overline{x_i \mapsto v_i}]]\!]$$

That is, for any choice of well-typed arguments $\overline{v_i}$, the application $f\ \overline{v_i}$ is in the denotation of the result type under the substitution $[x_i \mapsto v_i]$.

Now, by the definition of the denotation of the coverage base type $[\nu : b \mid \phi]$, we have:

$$[\![[\nu : b \mid \phi][\overline{x_i \mapsto v_i}]]\!] = \{e \mid \forall v.\ \phi[\overline{x_i \mapsto v_i}, \nu \mapsto v] \implies e \hookrightarrow^* v\}$$

Therefore, for any v such that $\phi[\overline{x_i \mapsto v_i}, \nu \mapsto v]$ holds, it must be that $f\ \overline{v_i} \hookrightarrow^* v$.

This is precisely the statement of the theorem:

$$\forall v.\ \phi\,[\overline{x_i \mapsto v_i}, \nu \mapsto v] \implies f\ \overline{v_i} \hookrightarrow^* v$$

Thus, the result follows directly. □

C Abduction Algorithm

Algorithm 3: Inferring missing coverage (Abduce)

Inputs : Γ: typing context, $[v : b \mid \psi_{\text{cur}}]$: current coverage, $[v : b \mid \psi]$: target coverage
Output : Formula ψ_{need} such that $\Gamma \vdash [v : b \mid \psi_{\text{cur}} \vee \psi_{\text{need}}] <: [v : b \mid \psi]$

1 **if** $\Gamma \vdash [v : b \mid \phi_{\text{cur}}] <: [v : b \mid \phi]$ **then return** $\bot$;
2 $\Psi \leftarrow \bigcup_{\phi \subseteq \Phi} \{\psi \mid \psi = \bigwedge_{\alpha \in \phi} \alpha \wedge \bigwedge_{\alpha \in \Phi - \phi} \neg\alpha\}$;
3 $\Psi^- \leftarrow \{\psi \in \Psi \mid \psi_{\text{cur}} \implies \psi\}$; $\Psi^? \leftarrow \Psi - \Psi^-$; $\Psi^+ \leftarrow \emptyset$; $\psi_{\text{need}} \leftarrow \top$;
4 **while** $\exists \psi_? \in \Psi^?$ **do**
5 $\quad \Psi^? \leftarrow \Psi^? \setminus \{\psi_?\}$;
6 $\quad \psi_{\text{need}} \leftarrow \texttt{Learn}(\Psi^+ \cup \Psi^?, \Psi^- \cup \{\psi_?\})$;
7 $\quad$ **if** $\Gamma \vdash [v : b \mid \psi_{\text{cur}} \vee \psi_{\text{need}}] <: [v : b \mid \psi]$ **then**
8 $\quad\quad \Psi^- \leftarrow \Psi^- \cup \{\psi_?\}$;
9 $\quad$ **else**
10 $\quad\quad \Psi^+ \leftarrow \Psi^+ \cup \{\psi_?\}$;
11 $\quad\quad \psi_{\text{need}} \leftarrow \texttt{Learn}(\Psi^+, \Psi^- \cup \Psi^?)$;
12 $\quad\quad$ **if** $\Gamma \vdash [v : b \mid \psi_{\text{cur}} \vee \psi_{\text{need}}] <: [v : b \mid \psi]$ **then**
13 $\quad\quad\quad$ **return** ψ_{need};
14 **return** ψ_{need};

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 algorithm is parameterized over a set of atomic formulas, Φ, and take three arguments: the first argument, Γ, is the typing context of the body of the generator, and the final two arguments 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 $\Gamma \vdash [v : b \mid \phi_{\text{cur}} \vee \phi_{\text{need}}] <: [v : b \mid \phi]$.

Abduce does so by adapting an existing algorithm for inferring a maximally weak specification in the context of safety verification [69] to our coverage type setting.

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 $\neg\texttt{empty}(l) \wedge \texttt{hd}(l) = 1$, 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.

Theorem C.1 (Abduce is Sound). *Given a typing context* Γ*, the current coverage* $[v:b \mid \psi_{\mathsf{cur}}]$*, and the target coverage* $[v:b \mid \psi]$*,* Abduce$(\Phi, \Gamma, [v:b \mid \psi_{\mathsf{cur}}], [v:b \mid \psi])$ *produces a* $\psi_{\mathsf{need}} \in \{\psi' \mid \psi' = \bigvee(\bigwedge \overline{\phi} \wedge \bigwedge \overline{\neg\phi}) \wedge \psi\}$ *such that* $\Gamma \vdash [v:b \mid \psi_{\mathsf{cur}} \vee \psi_{\mathsf{need}}] <: [v:b \mid \psi]$*. Moreover,* ψ_{need} *is a minimal solution in the solution space considered by* Abduce*:* $\neg\exists\, \psi' \in \{\psi' \mid \psi' = \bigvee(\bigwedge \overline{\phi} \wedge \bigwedge \overline{\neg\phi}) \wedge \psi\}$*.* $\Gamma \vdash [v:b \mid \psi_{\mathsf{cur}} \vee \psi'] <: [v:b \mid \psi] \wedge \psi' \implies \psi_{\mathsf{need}}$.

Proof. We first prove that the produced ψ_{need} always satisfies $\Gamma \vdash [v:b \mid \psi_{\mathsf{cur}} \vee \psi_{\mathsf{need}}] <: [v:b \mid \psi]$. The algorithm can only return in two cases: either at line 13, which guarantees that the subtyping relation holds, or at line 14, when all Boolean combinations of atomic formulas (i.e., $\bigwedge_{\alpha\in\phi} \alpha \wedge \bigwedge_{\alpha\in\Phi-\phi} \neg\alpha$) are labeled as positive (i.e., Ψ^+) or negative (i.e., Ψ^-). If ψ_{need} does not make the subtyping relation hold, then there exists a Boolean combination ψ_{ex} such that

$$\Gamma \vdash \psi_{\mathsf{ex}} \implies \psi \text{ and } \Gamma \nvdash \psi_{\mathsf{ex}} \implies \psi_{\mathsf{need}} \vee \psi_{\mathsf{cur}}$$

Moreover, for any formula in the solution space, we have

$$\Gamma \nvdash \psi_{\mathsf{ex}} \implies \psi' \text{ implies that } \Gamma \nvdash [v:b \mid \psi'] <: [v:b \mid \psi]$$

According to the definition of the learning procedure Learn, the formula ψ_{ex} can only belong to the set Ψ^-. This can only happen on line 8, where ψ_{need} makes the subtyping relation hold even when labeling ψ_{ex} as negative, which conflicts with the assumption. Thus, the returned ψ_{need} always satisfies $\Gamma \vdash [v:b \mid \psi_{\mathsf{cur}} \vee \psi_{\mathsf{need}}] <: [v:b \mid \psi]$.

Next, we prove minimality. If ψ_{need} is not minimal, then there exists a formula ψ' different from ψ_{need} in the solution space such that

$$\Gamma \vdash [v:b \mid \psi_{\mathsf{cur}} \vee \psi'] <: [v:b \mid \psi] \wedge \psi' \implies \psi_{\mathsf{need}}$$

Then, there exists a Boolean combination ψ_{max} such that

$$\Gamma \vdash \psi_{\mathsf{max}} \implies \psi_{\mathsf{need}} \text{ and } \Gamma \nvdash \psi_{\mathsf{max}} \implies \psi'$$

According to the definition of the learning procedure Learn, the formula ψ_{max} can only belong to the set Ψ^+. This can only happen on line 10, where ψ_{need} fails to make the subtyping relation hold even when labeling ψ_{max} as negative. This means that

$$\Gamma \vdash \psi_{\mathsf{max}} \implies \psi$$

which is a contradiction with the assumption that $\Gamma \nvdash \psi_{\mathsf{max}} \implies \psi'$ and $\Gamma \vdash [v:b \mid \psi_{\mathsf{cur}} \vee \psi'] <: [v:b \mid \psi]$. Thus, ψ_{need} is a minimal solution. □

D Localization Algorithm

Algorithm 4: Insert repair locations (`Localize`)

Inputs : s: incomplete generator, Γ: typing context, $[v : b \mid \phi_{\text{need}}]$: 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 $\quad v \Rightarrow$ **return** $(v \oplus \square \ : \ [v : b \mid \phi_{\text{need}}], \{\Gamma \vdash \square \ : \ [v : b \mid \phi_{\text{need}}]\})$;
3 $\quad \mid \texttt{err} \Rightarrow$ **return** $(\square \ : \ [v : b \mid \phi_{\text{need}}], \{\Gamma \vdash \square \ : \ [v : b \mid \phi_{\text{need}}]\})$;
4 $\quad \mid \square \ : \ [v : b \mid \phi] \Rightarrow$ **return** $(\square, \{\Gamma \vdash \square \ : \ [v : b \mid \phi], \Gamma \vdash \square \ : \ [v : b \mid \phi_{\text{need}}]\})$;
5 $\quad \mid \texttt{let}\ x \coloneqq e_1\ \texttt{in}\ e_2 \Rightarrow$
6 $\quad\quad (i_2, \Gamma') \leftarrow \texttt{Localize}(\Gamma; x : \texttt{TyInfer}(\Gamma, e_1), e_2, [v : b \mid \phi_{\text{need}}])$;
7 $\quad\quad$ **return** $(\texttt{let}\ x \coloneqq c_1\ \texttt{in}\ i_2, \Gamma')$
8 $\quad \mid \texttt{let}\ x \coloneqq op\ \overline{v}\ \texttt{in}\ e_2 \Rightarrow$
9 $\quad\quad (i_2, \Gamma') \leftarrow \texttt{Localize}(\Gamma; x : \texttt{TyInfer}(\Gamma, op\ \overline{v}), e_2, [v : b \mid \phi_{\text{need}}])$;
10 $\quad\quad$ **return** $(\texttt{let}\ x \coloneqq op\ \overline{v}\ \texttt{in}\ i_2, \Gamma')$
11 $\quad \mid \texttt{let}\ x \coloneqq v\ v\ \texttt{in}\ e_2 \Rightarrow$
12 $\quad\quad (i_2, \Gamma') \leftarrow \texttt{Localize}(\Gamma; x : \texttt{TyInfer}(\Gamma, v\ v), e_2, [v : b \mid \phi_{\text{need}}])$;
13 $\quad\quad$ **return** $(\texttt{let}\ x \coloneqq v\ v\ \texttt{in}\ i_2, \Gamma')$
14 $\quad \mid \texttt{match}\ v\ \texttt{with}\ \overline{d_k\ \overline{y_k} \rightarrow e_k} \Rightarrow$
15 $\quad\quad \Gamma' \leftarrow \emptyset$;
16 $\quad\quad$ **for** $m \in \{0, \ldots, \mathsf{k}\}$ **do**
17 $\quad\quad\quad \overline{y{:}\{v : b \mid \phi\}} \rightarrow [v : b_m \mid \psi_m] \leftarrow \mathsf{Ty}(d_m)$;
18 $\quad\quad\quad \Gamma'_j \leftarrow \overline{y{:}\{v : b \mid \phi\}},\ a{:}[v : b_m \mid v = v \wedge \psi_m]$;
19 $\quad\quad\quad (i_m, \Gamma_m) \leftarrow \texttt{Localize}(\Gamma; \Gamma'_j, e_j, [v : b \mid \phi_{\text{need}}])$;
20 $\quad\quad\quad \Gamma' \leftarrow \Gamma' \cup \Gamma_m$;
21 $\quad\quad$ **return** $(\texttt{match}\ v\ \texttt{with}\ \overline{d_k\ \overline{y_k} \rightarrow i_k},\ \Gamma')$

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 Correctness Proofs for Repair

Here are all the parameters that define the set of patches explored by Repair:

- A finite set of atomic formulas ϕ used by Abduce;
- A collection of typed *seeds* and *components* that Synthesize uses to enumerate terms;
- A set of method predicates used in the types of those components and by Abduce to characterize missing coverage;
- Axioms characterizing the semantics of method predicates;
- A cost function used by genExp to prioritize certain terms.

The soundness proofs for Repair are based on the following three lemmas.

Lemma E.1 (Soundness of TyInfer). *For a given type context* Γ *and incomplete program* s, $\texttt{TyInfer}(\Gamma, s)$ *returns a coverage type* $[\nu : b \mid \psi]$ *such that* $\Gamma \vdash s \ : \ [\nu : b \mid \psi]$.

Proof. This follows directly from the type soundness theorem for coverage types [70]. □

Lemma E.2 (Soundness of Localize). *For a given type context* Γ, *incomplete program* s *that covers* $[\nu : b \mid \psi_{\mathsf{cur}}]$, *and coverage type* $[\nu : b \mid \psi_{\mathsf{need}}]$, Localize *returns a new program sketch* s' *and set of typed holes* $\overline{\Gamma_j \vdash \square_j : [\nu : b \mid \psi_j]}$ *such that for all terms* $\overline{e_j}$ *where* $\overline{\Gamma_j \vdash e_j : [\nu : b \mid \psi_j]}$, $s'[\overline{e_j}]$ *is a repair of* s *that covers* $[\nu : b \mid \psi_{\mathsf{cur}} \vee \psi_{\mathsf{need}}]$, *i.e.,* $\Gamma \vdash s'[\overline{e_j}] \ : \ [\nu : b \mid \psi_{\mathsf{cur}} \vee \psi_{\mathsf{need}}]$.

Proof. By induction on the structure of the program s. □

Lemma E.3 (Soundness of Synthesize). *For a given type context* Γ *and program sketch* s' *with typed holes* $\overline{\Gamma_j \vdash \square_j : [\nu : b \mid \psi_j]}$, *such that for all terms* $\overline{e_j}$ *where* $\overline{\Gamma_j \vdash e_j : [\nu : b \mid \psi_j]}$, $\Gamma \ \vdash \ s'[\overline{e_j}] \ : \ [\nu : b \mid \psi]$, Synthesize *returns a complete program with type* $[\nu : b \mid \psi]$*:*
$\Gamma \vdash \texttt{Synthesize}(\Gamma, s', \overline{\Gamma_j \vdash \square_j \ : \ [\nu : b \mid \psi_j]}, [\nu : b \mid \psi]) \ : \ [\nu : b \mid \psi]$.

Proof. Synthesize only exits immediately after the main loop if the current completion of s' has the required coverage type: $\Gamma \vdash s'[\overline{e_j}] \ : \ [\nu : b \mid \psi]$. Otherwise, for the i^{th} hole in s', the main body of Synthesize will have only produced a patch e_i whose type precisely matches $\Gamma_i \vdash e_i : [\nu : b \mid \psi_i]$ (lines 12-13), and the second loop will have filled any hole $\square_k$ that remains with a term e_k that is a subtype of $[\nu : b \mid \psi_k]$, and thus $\Gamma_k \vdash e_k : [\nu : b \mid \psi_k]$ – in the worst case, this term will be the default generator for the base type b, which is always included in our set of components. Thus, if Synthesize exits on line 21, $\overline{\Gamma_j \vdash e_j : [\nu : b \mid \psi_j]}$; by assumption Synthesize will return a completed sketch with the required coverage type $\Gamma \vdash s'[\overline{e_j}] \ : \ [\nu : b \mid \psi]$. □

Theorem E.4 (Repair is Sound). *Given a program* s *that is well typed under typing context* Γ, $\Gamma \vdash s \ : \ b$, *and target coverage type* $[\nu : b \mid \psi]$, *if* Repair *terminates, it returns a coverage complete repaired program* s', $\texttt{Repair}(\Gamma, \ s, \ [\nu : b \mid \psi]) = s' \implies \Gamma \vdash s' \ : \ [\nu : b \mid \psi]$.

Proof. We proceed by following the structure of the Repair algorithm and applying the soundness of its subroutines.

Step 1: Type Inference. By Lemma E.1, applying $\texttt{TyInfer}(\Gamma, s)$ yields a coverage type $[\nu : b \mid \psi_{\mathsf{cur}}]$ such that $\Gamma \vdash s : [\nu : b \mid \psi_{\mathsf{cur}}]$.

Step 2: Abduction. By Theorem C.1, the call to $\texttt{Abduce}(\Gamma, [\nu : b \mid \psi_{\mathsf{cur}}], [\nu : b \mid \psi])$ produces a formula ψ_{need} such that

$$\Gamma \vdash [\nu : b \mid \psi_{\mathsf{cur}} \vee \psi_{\mathsf{need}}] <: [\nu : b \mid \psi]$$

That is, any program covering $\psi_{\mathsf{cur}} \vee \psi_{\mathsf{need}}$ is a subtype of the target coverage ψ.

Step 3: Localization. By Lemma E.2, the call to `Localize`$(\Gamma, s, [v:b \mid \psi_{\mathsf{need}}])$ produces a program sketch s' with holes, and for any choice of terms $\overline{e_j}$ such that $\overline{\Gamma \vdash e_j : [v:b \mid \psi_j]}$, the filled program $s'[\overline{e_j}]$ satisfies

$$\Gamma \vdash s'[\overline{e_j}] : [v:b \mid \psi_{\mathsf{cur}} \vee \psi_{\mathsf{need}}]$$

Step 4: Synthesis. By Lemma E.3, the call `Synthesize`$(\Gamma, s', \overline{\Gamma_j \vdash \square_j : [v:b \mid \psi_j]}, \psi_{\mathsf{cur}} \vee \psi_{\mathsf{need}})$ will produce a term $s'[\overline{e_j}]$ such that $\Gamma \vdash s'[\overline{e_j}] : [v:b \mid \psi_{\mathsf{cur}} \vee \psi_{\mathsf{need}}]$.

Step 5: Subsumption. By the typing subsumption rule and the result of Step 2, since

$$\Gamma \vdash s[\overline{e_j}] : [v:b \mid \psi_{\mathsf{cur}} \vee \psi_{\mathsf{need}}] \quad \text{and} \quad \Gamma \vdash [v:b \mid \psi_{\mathsf{cur}} \vee \psi_{\mathsf{need}}] <: [v:b \mid \psi]$$

we conclude

$$\Gamma \vdash s[\overline{e_j}] : [v:b \mid \psi]$$

as required.

Therefore, if `Repair` terminates, it returns a program e such that $\Gamma \vdash e : [v:b \mid \psi]$, establishing the soundness of `Repair`. □

The completeness proofs for `Repair` are based on the following four lemmas.

Lemma E.5 (Termination of `TyInfer`). *For a given type context Γ and incomplete program s, `TyInfer` always terminates.*

Proof. This follows from the type completeness theorem for coverage types [70]. □

Lemma E.6 (Termination of `Abduce`). *For a given type context Γ, coverage type $[v:b \mid \psi_{\mathsf{cur}}]$ and $[v:b \mid \psi]$, if there exists $[v:b \mid \psi_{\mathsf{need}}]$ such that $\Gamma \vdash [v:b \mid \psi_{\mathsf{cur}} \vee \psi_{\mathsf{need}}] <: [v:b \mid \psi]$, `Abduce` always terminates.*

Proof. The main body of the `Abduce` subroutine is a refinement loop. We need to show that this loop terminates. The loop assigns each Boolean combination from the set $\Psi^?$ either to Ψ^+ (line 10) or to Ψ^- (line 8). The set $\Psi^?$ is finite because it is derived from a finite set of atomic formulas (Φ). Thus, the `Abduce` subroutine terminates. □

Lemma E.7 (Termination of `Localize`). *For a given type context Γ, incomplete program s that covers $[v:b \mid \psi_{\mathsf{cur}}]$, and coverage type $[v:b \mid \psi_{\mathsf{need}}]$, if there exists a program sketch s' such that $\Gamma \vdash s' : [v:b \mid \psi_{\mathsf{cur}} \vee \psi_{\mathsf{need}}]$, `Localize` always terminates.*

Proof. By induction on the structure of the program s. □

Lemma E.8 (Termination of `Synthesize`). *For a given type context Γ and program sketch s' and terms $\overline{e_j}$ where $\overline{\Gamma \vdash e_j : [v:b \mid \psi_j]}$, `Synthesize` always terminates.*

Proof. The main body of the `Synthesize` subroutine is an iterative loop with an increasing cost bound. Each step of the loop is guaranteed to terminate, as there is a finite number of holes, $\overline{\square_j}$, and `genExp` will only generate a finite number of terms at each cost level. The entire loop must eventually hit the upper cost bound and terminate. The subsequent loop also iterates over at most j unfilled holes, and will also terminate. Therefore, `Synthesize` always terminates. □

Theorem E.9 (`Repair` Terminates). `Repair` *always terminates.*

Proof. This is a direct consequence of the previous three lemmas:

- By Lemma E.5, `TyInfer` always terminates for any input s and context Γ.
- Since a valid repair exists, there must exist a non-bottom type $[v:b \mid \psi_{\mathsf{need}}]$ such that $\Gamma \vdash [v:b \mid \psi_{\mathsf{cur}}] \vee [v:b \mid \psi_{\mathsf{need}}] <: [v:b \mid \psi]$. By Lemma E.6, `Abduce` always terminates.
- Given s' and $[v:b \mid \psi_{\mathsf{need}}]$, Lemma E.7 ensures that `Localize` always terminates.

- Given s' and the terms $\overline{e_j}$, Lemma E.8 ensures that `Synthesize` always terminates.

Therefore, `Repair` always terminates. □

Theorem E.10 (`Repair` is Sound and Total). *Given a program s that is well typed under typing context Γ, $\Gamma \vdash s \ : \ b$, and target coverage type $[v : b \mid \psi]$, `Repair` returns a coverage complete generator, $\Gamma \vdash$ `Repair`$(s,\ \Gamma,\ [v : b \mid \psi])\ :\ [v : b \mid \psi]$.*

Proof. Follows immediately from Theorem E.4 and Theorem E.9. □

F Unique generated values

Generator	Unique_Count	Total_Duplicates	Duplicate_Count	None_Count
Sized List	16759	3241	1	0
Unique List	18939	1061	1	0
Even List	20000	0	0	0
Sorted List	778	1030	1	18192
Duplicate List	18875	1125	1	0
Red-Black Tree	17921	2079	1	0
Complete Tree	18441	1559	1	0
Sized Tree	15124	4876	1	0

The table above presents an analysis of the number of unique and duplicate values produced by the Cobb-repaired generators from Figure 11. Each value that is output precisely once is reported as a unique element under the 'Unique_Count' column. The 'Total_Duplicates' column includes the total number of repeated values. 'Duplicate_Count' reports the number of values that are repeated more than once. Note that for most benchmarks the last value is 1, as the only duplicate value generated tends is always either the leaf node or the empty list: Even List has a non-nil base case and therefore doesn't admit any duplicates. Finally, the 'None_Count' column presents the number of runs on which the generator threw an error – as described in Section the Sorted List benchmark is the only one that does so.