Technical Report (PRL-2020-01)
Context-Aware and Data-Driven Program Repair

Dowon Song
dowon_song@korea.ac.kr
Korea University

Hakjoo Oh
hakjoo_oh@korea.ac.kr
Korea University

1 Problem Definition

In this section, we define our program-repair problem. We first define a program model that captures key aspects of ML-like languages and introduce notations that allow us to formalize our algorithm in the next section.

To formalize our approach, we consider an idealized functional language similar to the core of ML, with the additional property that we *label* all expressions. Our target language features algebraic data types and recursive functions. A program is an expression defined as follows:

$e \in Exp$	(Expressions)	$x \in VId$	(Variables)
$f \in FId$	(Functions)	$Id = VId \cup FId$	(Identifiers)
$\ell \in Label$	(Labels)	$\tau \in Type$	(Types)

$$
\begin{array}{rcl}
e & ::= & n \mid x \mid \lambda x.e \mid e_1 \oplus e_2 \mid e_1\ e_2 \mid \kappa(e_1, \cdots, e_{a(\kappa)}) \\
& \mid & \kappa^{-i}(e) \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid \mathsf{let\ rec}\ f(x) = e_1\ \mathsf{in}\ e_2 \\
& \mid & \mathsf{match}\ e\ \mathsf{with}\ \overline{p_i \rightarrow e_i}^{k} \\
p & ::= & \kappa(x_1, \cdots, x_n) \mid \_ \qquad \tau ::= \mathsf{int} \mid T \mid \tau_1 \rightarrow \tau_2
\end{array}
$$

We assume each expression is associated with a unique label. Expression e associated with a label $\ell \in \mathcal{L}$ is denoted by e^{ℓ}. For the sake of better readability, we will often elide ℓ when the label is not necessary for discussion. In addition, when we determine equality of two expressions, we do not consider their labels and only check if they are syntactically equivalent.

The syntax of expressions is standard: application is written $e_1\ e_2$, κ ranges over data type constructors, $a(\kappa)$ denotes the arity of κ, κ^{-i} denotes a destructor which extracts the i-th subcomponent of a constructor κ, and let bindings for variables and recursive functions are allowed. For conciseness, we assume that all functions take a single argument and are not mutually recursive. We use ML-style pattern match expressions in which each pattern p binds subcomponents of a constructor κ, or the underscore (_) called the wildcard pattern. We use $\overline{p_i \rightarrow e_i}^{k}$ to denote $p_1 \rightarrow e_1 \mid \cdots \mid p_k \rightarrow e_k$. Types include the integer type int, user-defined algebraic data types T, and function types $\tau_1 \rightarrow \tau_2$.

We will use some notations regarding expressions throughout the remaining sections. We use $\longrightarrow^*$ to denote the standard multi-step call-by-value operational relation. To denote the set of all subexpressions of expression e, we will use $\mathsf{Sub}(e)$. The size of expression e will be denoted by $|e|$. Identifiers of functions defined in an expression e will be denoted by $\mathsf{functions}(e)$ (i.e., $\mathsf{functions}(e) = \{f \in FId \mid \mathsf{let\ rec}\ f(x) = e_1\ \mathsf{in}\ e_2 \in \mathsf{Sub}(e)\}$). Identifiers used in an expression e will be denoted by $\mathsf{vars}(e)$.

We assume that each program $P \in Exp$ has no type error, and is in the following form:

$$\mathsf{let\ rec}\ f(x) = e\ \mathsf{in}\ f\ x_\iota$$

where f is a top-level function, and x_ι is a special variable which we call *input variable*. For convinience, we also assume that all labels and identifiers in the pile of entire programs are unique with exception of the input variable that all the programs have in common. This assumption enables to use the following global functions: (1) $\mathsf{body} : FId \rightarrow Exp$ that returns the body of a function, (2) $\mathsf{param} : FId \rightarrow VId$ that returns the parameter variable of a function, and (3) $\mathsf{type} : (Label \cup Id) \rightarrow Type$ that returns the type of an expression with a given label or a variable.

Assuming some (possibly infinite) set Val of values, a set of test cases $\mathcal{T} \subseteq Val \times Val$ is used to determine the correctness of each program. The program P is correct (denoted $\mathsf{correct}(P, \mathcal{T})$) iff

$$\forall (i, o) \in \mathcal{T}.\ (\lambda x_\iota.\ P)\ i \longrightarrow^* o.$$

Otherwise, the program is buggy.

Our problem is defined as follows: given a buggy program $P_b \in Exp$, a corpus of correct programs $\mathcal{P}_c \subseteq Exp$, and a set of test cases $\mathcal{T}$, derive a correct program $P \notin \mathcal{P}_c$ from P_b with minimal changes (the notion of the minimality will be detailed in Section **??**).

2 Repair Algorithm

In this section, we describe our repair algorithm. Section **??** formalizes calling contexts and how to find a matching relation between functions. Section **??** describes the repair algorithm that extracts repair templates from reference functions and uses them to correct a buggy program.

2.1 Context-Aware Matching

We formally define the notion of context-aware matching. From each function call in all the given programs, we collect calling contexts. A context $\delta \in Ctx$ is a path condition on the input variable under which a function is invoked. Formally, path conditions are defined below:

$$\delta ::= \mathsf{true} \mid \mathsf{false} \mid e = e \mid \neg\delta \mid \delta \wedge \delta.$$

Calling contexts are defined as follows:

Definition 2.1 (Calling context)**.** A calling context is a triple $\langle f, \delta, g\rangle \in FId \times Ctx \times FId$ where f is a function, g is another function called in the body of f, and δ is a path condition under which the function call happens.

We first perform a *path-sensitive 0-CFA* on all the programs to obtain calling contexts. Like the standard 0-CFA, the analysis information at any given expression is the set of possible evaluation results of the expression. Here, evaluation results are expressions in which the input variable is a free variable. We add path sensitivity by making our analysis track information separately for different execution paths. The analysis computes a dataflow state

$$\sigma \in (Id \cup \mathcal{L}) \times Ctx \to \mathcal{P}(Exp) \cup \{\top\}.$$

Figure **??** depicts the constraint generation rules. The judgement $\delta \vdash [\![e]\!]^{\ell} \hookrightarrow C$ can be read as "the analysis of expression e with label ℓ generates set constraints C over dataflow state σ under a current path condition δ". While solving the constraints via a least-fix point computation, four kinds of special constraints of the forms

$$\mathbf{fn}_{\delta}\ \ell_1 : \ell_2 \implies \ell \qquad \mathbf{cn}^{\kappa}_{\delta}\ \ell_1, \cdots, \ell_{a(\kappa)} \implies \ell$$
$$\mathbf{cn}^{\kappa^{-i}}_{\delta}\ \ell' \implies \ell \qquad \mathbf{pat}_{\delta}\ \ell_0 : \overline{p_i \to e_i^{\ell_i}}^{k} \implies \ell$$

are interpreted by the constraint solver to generate additional concrete constraints by referring to an intermediate analysis result. For example, from a constraint $\mathbf{fn}_{\delta}\ \ell_1 : \ell_2 \implies \ell$, for every function $\lambda x.e_0^{\ell_0}$ that the analysis (eventually) concludes the expression labeled ℓ_1 may evaluate to, additional constraints are generated to capture value flow from the actual argument expression ℓ_2 to formal function argument x, and from the function result to the calling expression ℓ. To enforce termination, we use a standard widening operator that transforms each collected expression whose size is greater than a threshold into $\top$.

Note that the analysis for collecting calling contexts does not affect the correctness of the overall algorithm but just determines the effectiveness of matching.

We derive calling contexts from a result of the path-sensitive 0-CFA as follows: given programs $P_1^{\ell_1}, \cdots, P_m^{\ell_m}$, we collect set constraints C_i for each program such that $\text{true} \vdash [\![P_i]\!]^{\ell_i} \hookrightarrow C_i$, and obtain the least solution σ_i. We collect a set of calling contexts $\Delta = \Delta_1 \cup \cdots \cup \Delta_m$ where each Δ_i is

$$\bigcup_{\substack{f \in \mathsf{functions}(P_i) \\ \delta \in Ctx}} \{\langle f, \delta, g\rangle \mid (e_1^{\ell_1}\ e_2) \in \mathsf{Sub}(\mathsf{body}(f)), g \in \sigma_i(\ell_1, \delta)\}.$$

We also conjoin the analysis results for each program and obtain $\sigma = \bigsqcup_{1 \le i \le m} \sigma_i$, which will also be used in Section **??**.

Now we are ready to measure similarity between arbitrary two functions using the calling contexts. Given two functions f and g, we compute a distance between the two functions as follows:

$$\mathsf{dist}(f, g) = w_1 \times |CC^{f,g}_{\text{in}}|^{-1} + w_2 \times |CC^{f,g}_{\text{out}}|^{-1}$$

where $w_{\{1,2\}}$ are coefficients that can be adjusted via statistical learning. $CC^{f,g}_{\text{in}}$ (resp. $CC^{f,g}_{\text{out}}$) is called incoming (resp. outgoing) compatible calling context and defined as follows:

$$CC^{f,g}_{\text{in}} = \{(\delta, \delta') \mid \delta, \delta' \in Ctx, \langle \_, \delta, f\rangle, \langle \_, \delta', g\rangle \in \Delta, SAT(\delta \wedge \delta')\}$$

$$CC^{f,g}_{\text{out}} = \{(\delta, \delta') \mid \delta, \delta' \in Ctx, \langle f, \delta, \_\rangle, \langle g, \delta', \_\rangle \in \Delta, SAT(\delta \wedge \delta')\}$$

If both f and g do not have any callers (resp. callees), we do not consider the term involving $CC^{f,g}_{\text{in}}$ (resp. $CC^{f,g}_{\text{out}}$). Note that the more compatible pairs of calling contexts two functions have, the shorter distance they have between. Based on this notion of distance, we are equipped with the following matching function that takes a function in a buggy program and returns a function in a correct program to be referred for correction:

$$\mathcal{M} = \lambda P_b.\, \{f \mapsto \underset{\substack{P_c \in \mathcal{P}_c \\ g \in \mathsf{functions}(P_c) \\ \mathsf{type}(f) = \mathsf{type}(g)}}{\mathsf{argmin}} \mathsf{dist}(f, g) \mid f \in \mathsf{functions}(P_b)\}.$$

Note that we only consider functions of the same type.

In case of tie, we pick the most syntactically similar function. To measure the syntactic similarity, we use the method of embedding ASTs into numerical vectors. We compute Euclidean distances between vectors to obtain the syntactic distances.

2.2 Context-Aware Repair

In this subsection, we explain how to extract repair templates from correct programs and instantiate them to generate patches. In particular, our goal is to obtain a sequence of *edit actions* that transform a given buggy program into a new correct one. This sequence is called *edit script*. We consider the following edit actions:

- $\mathsf{Modify}(\ell, e)$ replaces the old subexpression at label ℓ by the new expression e.
- $\mathsf{Insert}(\ell, p \to e)$ adds a new pattern matching case $p \to e$ into a match expression associated with label ℓ.
- $\mathsf{Delete}(\ell, p \to e)$ removes an existing pattern matching case $p \to e$ from a match expression associated with label ℓ.
- $\mathsf{Define}(f)$ adds a new definition of function f into the expression. If we apply this action into an expression e, the resulting expression would be $\mathsf{let\ rec}\ f(\mathsf{param}(f)) = \mathsf{body}(f)\ \mathsf{in}\ e$.

2.2.1 Learning Repair Templates

We generate edit scripts by *instantiating* templates (which we call *repair templates*) collected from correct programs. A repair template is a variant of an edit action where each expression in Modify or Insert action does not have any variables but just *holes*. Each hole is annotated with a type and plays a role as a placeholder that can be replaced with a

$$\frac{}{\delta \vdash [\![n]\!]^{\ell} \hookrightarrow n \in \sigma(\ell,\delta)} \quad \frac{[\![e]\!]^{\ell'} \hookrightarrow C}{\delta \vdash [\![\lambda x.e^{\ell'}]\!]^{\ell} \hookrightarrow \lambda x.e \in \sigma(\ell,\delta) \cup C} \quad \frac{\delta \vdash [\![e_1]\!]^{\ell_1} \hookrightarrow C_1 \quad \delta \vdash [\![e_2]\!]^{\ell_2} \hookrightarrow C_2}{\delta \vdash [\![e_1\ e_2]\!]^{\ell} \hookrightarrow C_1 \cup C_2 \cup \mathbf{fn}\ \ell_1 : \ell_2 \implies \ell}$$

$$\frac{x \in \{x_\iota\} \cup FId}{\delta \vdash [\![x]\!]^{\ell} \hookrightarrow x \in \sigma(x,\delta) \cap \sigma(\ell,\sigma)} \quad \frac{x \notin \{x_\iota\} \cup FId}{\delta \vdash [\![x]\!]^{\ell} \hookrightarrow \sigma(x,\delta) \subseteq \sigma(\ell,\sigma)} \quad \frac{\delta \vdash [\![e]\!]^{\ell'} \hookrightarrow C}{\delta \vdash [\![\kappa^{-i}(e^{\ell'})]\!]^{\ell} \hookrightarrow C \cup \mathbf{cn}_{\delta}^{\kappa^{-i}}\ \ell' \implies \ell}$$

$$\frac{\delta \vdash [\![e_1]\!]^{\ell_1} \hookrightarrow C_1 \quad \delta \vdash [\![e_2]\!]^{\ell_2} \hookrightarrow C_2}{\delta \vdash [\![e_1^{\ell_1} \oplus e_2^{\ell_2}]\!]^{\ell} \hookrightarrow \sigma(\ell_1,\delta) \oplus_{\uparrow} \sigma(\ell_2,\delta) \subseteq \sigma(\ell,\delta) \cup C_1 \cup C_2} \quad \frac{\delta \vdash [\![e_1]\!]^{\ell_1} \hookrightarrow C_1 \quad \cdots \quad \delta \vdash [\![e_n]\!]^{\ell_n} \hookrightarrow C_n}{\delta \vdash [\![\kappa(e_1^{\ell_1},\cdots,e_n^{\ell_n})]\!]^{\ell} \hookrightarrow \bigcup_{1 \le i \le n} C_i \cup \mathbf{cn}_{\delta}^{\kappa}\ \ell_1,\cdots,\ell_n \implies \ell}$$

$$\frac{\delta \vdash [\![e]\!]^{\ell_0} \hookrightarrow C_0}{\delta \vdash [\![\mathsf{match}\ e^{\ell_0}\ \mathsf{with}\ \overline{p_i \to e_i}^{k}]\!]^{\ell} \hookrightarrow C_0 \cup \mathbf{pat}_{\delta}\ \ell_0 : \overline{p_i \to e_i}^{k} \Longrightarrow \ell} \quad \frac{e_1 \in \sigma(\ell_1,\delta)\ \cdots\ e_n \in \sigma(\ell_n,\delta)}{\mathbf{cn}_{\delta}^{\kappa}\ \ell_{1..n} \Longrightarrow \ell \hookrightarrow \kappa(e_1,..,e_n) \in \sigma(\ell,\delta)} \quad \frac{e \in \sigma(\ell',\delta)}{\mathbf{cn}_{\delta}^{\kappa^{-i}}\ \ell' \Longrightarrow \ell \hookrightarrow \kappa^{-i}(e) \in \sigma(\ell,\delta)}$$

$$\frac{\lambda x.e_0^{\ell_0} \in \sigma(\ell_1,\delta)}{\mathbf{fn}_{\delta}\ \ell_1 : \ell_2 \implies \ell \hookrightarrow \{\sigma(\ell_2,\delta) \subseteq \sigma(x,\delta), \sigma(\ell_0,\delta) \subseteq \sigma(\ell,\delta)\}} \quad \frac{f \in \sigma(\ell_1,\delta) \quad x = \mathsf{param}(f) \quad e_0^{\ell_0} = \mathsf{body}(f)}{\mathbf{fn}_{\delta}\ \ell_1 : \ell_2 \implies \ell \hookrightarrow \{\sigma(\ell_2,\delta) \subseteq \sigma(x,\delta), \sigma(\ell_0,\delta) \subseteq \sigma(\ell,\delta)\}}$$

$$\frac{e_0 \in \sigma(\ell_0,\delta) \quad \begin{array}{c}\delta \wedge r_1 \vdash e_1 \hookrightarrow C_1\\ \delta \wedge r_2 \vdash e_2 \hookrightarrow C_2\\ \cdots\\ \delta \wedge r_k \hookrightarrow C_k\end{array} \quad r_i = \begin{cases} e_0 = p_i & (i=1)\\ e_0 = p_i \wedge \bigwedge_{1 \le j < i} e_0 \ne p_j & (\text{o.w.})\end{cases} \quad \begin{array}{l} C_{FV} = \bigcup_{1 \le i \le k} \{\sigma(x,\delta) \subseteq \sigma(x,\delta \wedge r_i) \mid x \in FV(e_i)\}\\ C_{\sqcup} = \bigcup_{1 \le i \le k} \{\sigma(\ell_i, \delta \wedge r_i) \subseteq \sigma(\ell,\delta)\}\end{array}}{\mathbf{pat}_{\delta}\ \ell_0 : \overline{p_i \to e_i^{\ell_i}}^{k} \implies \ell \hookrightarrow \bigcup_{1 \le i \le k} C_k \cup C_{FV} \cup C_{\sqcup} \cup \bigcup_{1 \le i \le k, p_i = \kappa(x_1,\cdots,x_n)} \{\kappa^{-j}(e_0) \in \sigma(x_j,\delta) \mid 1 \le j \le n\}}$$

Figure 1. Constraint generation rules for our path-sensitive 0-CFA (some simple cases are omitted for brevity). The special constraints of the kinds **fn**, **cn**, and **pat** are interpreted by the constraint solver to generate additional concrete constraints. $E_1 \oplus_{\uparrow} E_2$ denotes $\{e_1 \oplus e_2 \mid e_1 \in E_1, e_2 \in E_2\}$.

variable of the type. The set $Exp_{\Box}$ of expressions with holes is similarly defined as Exp in the followings.

$$\begin{array}{rcl} e_{\Box} & \in & Exp_{\Box} \\ e_{\Box} & ::= & \Box_{\tau} \mid n \mid \lambda x.e_{\Box} \mid e_{\Box,1} \oplus e_{\Box,2} \mid e_{\Box,1}\ e_{\Box,2} \\ & \mid & \kappa(e_{\Box,1},\cdots,e_{\Box,a(\kappa)}) \mid \kappa^{-i}(e_{\Box}) \\ & \mid & \mathsf{let}\ x = e_{\Box,1}\ \mathsf{in}\ e_{\Box,2} \\ & \mid & \mathsf{let\ rec}\ f(x) = e_{\Box,1}\ \mathsf{in}\ e_{\Box,2} \\ & \mid & \mathsf{match}\ e_{\Box}\ \mathsf{with}\ \overline{p_{\Box,i} \to e_{\Box,i}}^{k} \\ p_{\Box} & ::= & \kappa(\Box_{\tau_1},\cdots,\Box_{\tau_k}) \mid \_ \end{array}$$

An expression with holes can be considered an abstraction of multiple expressions. The abstraction function $\alpha_e : Exp \to Exp_{\Box}$, which we apply to expressions in correct programs to extract templates, is defined as follows (to avoid unnecessary clutter, we omit simple inductive cases):

$$\alpha_e(n) = n \qquad \alpha_e(x^{\ell}) = \Box^{\ell}_{\mathsf{type}(x)} \qquad \alpha_e(\lambda x.e) = \lambda x.\alpha_e(e)$$
$$\cdots$$
$$\alpha_e(\mathsf{match}\ e\ \mathsf{with}\ \overline{p_i \to e_i}^{k}) = \mathsf{match}\ \alpha_e(e)\ \mathsf{with}\ \overline{p_i \to \alpha_e(e_i)}^{k}$$
$$\alpha_p(\kappa(x_1,\cdots,x_{a(\kappa)})) = \kappa(\alpha_e(x_1),\cdots,\alpha_e(x_{a(\kappa)})) \quad \alpha_p(\_) = \_$$

When abstracting a variable into a hole, we preserve its label. The label is used in various ways, which will be described later in the next subsection.

Now we describe how to generate repair templates. Given a buggy program P and the matching function $\mathcal{M}$, we obtain a set of templates $\mathcal{T} = \mathcal{T}_D \cup \mathcal{T}_M$ where $\mathcal{T}_D$ is a set of templates of kind Define defined as follows:

$$\mathcal{T}_D = \{\mathsf{Define}(g) \mid f \in \mathsf{functions}(P), g \in \mathsf{callees}(\mathsf{body}(\mathcal{M}(f)))\}.$$

In other words, we collect all the auxiliary functions used in reference programs. The function callees : $Exp \to \mathcal{P}(FId)$ returns all functions that may be invoked in a given expression. The set $\mathcal{T}_M$ includes templates of kinds Modify, Insert, and Delete defined as follows:

$$\mathcal{T}_M = \bigcup \{T \mid f \in \mathsf{functions}(P), [\![\mathsf{body}(f), \mathsf{body}(\mathcal{M}(f))]\!] \rightsquigarrow T\}.$$

The judgement $[\![e, e']\!] \rightsquigarrow T$ can be read as "by differencing a buggy expression e and a reference expression e', we extract a set T of edit action templates that can be potentially used to correct e". Figure **??** depicts inference rules for extracting templates for a given pairs of expressions.

2.2.2 Generating Edit Scripts

We instantiate the collected templates into edit actions using the following concretization function $\gamma_e^P : Exp_{\Box} \to \mathcal{P}(Exp)$ parametrized by a buggy program P.

$$\gamma_e^P(n) = \{n\} \qquad \gamma_e^P(\lambda x.e_{\Box}) = \{\lambda x.e \mid e \in \gamma_e^P(e_{\Box})\}$$
$$\cdots$$
$$\gamma_e^P(\mathsf{match}\ e_{\Box,0}\ \mathsf{with}\ \overline{p_i \to e_{\Box,i}}^{k}) = \{\mathsf{match}\ e_0\ \mathsf{with}\ \overline{p_i \to e_i}^{k} \mid e_i \in \gamma_e^P(e_{\Box,i})\}$$

Most importantly,

$$\gamma_e^P(\Box_{\tau}^{\ell}) = \begin{cases} \{x \in V \mid \mathsf{type}(x) = \tau\} & (\exists x \in V.\mathsf{type}(x) = \tau) \\ V & (\text{otherwise}) \end{cases}$$

where $V = \mathsf{vars}(P) \cup \{x \in FId \mid \delta \in Ctx.x \in \sigma(\ell,\delta)\}$ is the set of candidate variables for the given hole $\Box_{\tau}^{\ell}$. Note that the label ℓ always originates from a correct program. We consider not only variables in P but also function identifiers

$$\frac{}{[\![n_1, n_2]\!] \rightsquigarrow \emptyset}\ n_1 = n_2 \qquad \frac{}{[\![x_1^{\ell_1}, x_2^{\ell_2}]\!] \rightsquigarrow \{\mathsf{Modify}(\ell_1, \square^{\ell_2}_{\mathsf{type}(\ell_2)})\}} \qquad \frac{[\![e_1, e_2]\!] \rightsquigarrow T}{([\![\lambda x_1.e_1), (\lambda x_2.e_2)]\!] \rightsquigarrow T} \qquad \frac{[\![e_1, e_2]\!] \rightsquigarrow T}{[\![\kappa^{-i}(e_1), \kappa^{-i}(e_2)]\!] \rightsquigarrow T}$$

$$\frac{[\![e_1, e_1']\!] \rightsquigarrow T_1 \quad [\![e_2, e_2']\!] \rightsquigarrow T_2}{[\![(e_1 \oplus e_2), (e_1' \oplus' e_2')]\!] \rightsquigarrow T_1 \cup T_2}\ \oplus = \oplus' \qquad \frac{[\![e_1, e_1']\!] \rightsquigarrow T_1 \quad [\![e_2, e_2']\!] \rightsquigarrow T_2}{[\![(e_1\ e_2), (e_1'\ e_2')]\!] \rightsquigarrow T_1 \cup T_2} \qquad \frac{[\![e_1, e_1']\!] \rightsquigarrow T_1 \quad \cdots \quad [\![e_k, e_n']\!] \rightsquigarrow T_n}{[\![\kappa(e_1, \cdots, e_n), \kappa(e_1', \cdots, e_n')]\!] \rightsquigarrow \bigcup_{1 \le i \le n} T_i}$$

$$\frac{[\![e, e_2]\!] \rightsquigarrow T}{[\![e, (\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2)]\!] \rightsquigarrow T} \qquad \frac{[\![e_2, e]\!] \rightsquigarrow T}{[\![(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2), e]\!] \rightsquigarrow T} \qquad \frac{[\![e_1, e_1']\!] \rightsquigarrow T_1 \quad [\![e_2, e_2']\!] \rightsquigarrow T_2}{[\![(\mathsf{let\ rec}\ f_1(x_1) = e_1\ \mathsf{in}\ e_2), (\mathsf{let\ rec}\ f_2(x_2) = e_1'\ \mathsf{in}\ e_2')]\!] \rightsquigarrow T_1 \cup T_2}$$

$$\frac{[\![e, e_2]\!] \rightsquigarrow T}{[\![e, (\mathsf{let\ rec}\ f(x) = e_1\ \mathsf{in}\ e_2)]\!] \rightsquigarrow T} \qquad \frac{\begin{array}{c} T_M = \bigcup\{T_j \mid 1 \le j \le k, [\![e_j, e_j']\!] \rightsquigarrow T_j, \alpha_p(p_j) = \alpha_p(p_j')\} \\ [\![e_0, e_0']\!] \rightsquigarrow T_0 \quad T_I = \{\mathsf{Insert}(\ell, p_j \to \alpha_e(e_j)) \mid 1 \le j \le m, \alpha_p(p_j) \in \{\alpha_p(p_i')\}_{i=1}^n \backslash \{\alpha_p(p_i)\}_{i=1}^m\} \\ T_D = \{\mathsf{Delete}(\ell, p_j \to e_j) \mid 1 \le j \le n, \alpha_p(p_j) \in \{\alpha_p(p_i)\}_{i=1}^n \backslash \{\alpha_p(p_i')\}_{i=1}^m\} \end{array}}{[\![(\mathsf{match}\ e_0\ \mathsf{with}\ \overline{p_i \to e_i}^n), (\mathsf{match}\ e_0'\ \mathsf{with}\ \overline{p_i' \to e_i'}^m)]\!] \rightsquigarrow T_0 \cup T_M \cup T_I \cup T_D}$$

$$\frac{[\![e_2, e]\!] \rightsquigarrow T}{[\![(\mathsf{let\ rec}\ f(x) = e_1\ \mathsf{in}\ e_2), e]\!] \rightsquigarrow T} \qquad \frac{}{[\![e_1^{\ell_1}, e_2^{\ell_2}]\!] \rightsquigarrow \{\mathsf{Modify}(\ell_1, \alpha_e(e_2))\}} \quad \text{(the other remaining cases)}$$

Figure 2. Inference rules for extracting edit action templates for given two expressions.

reachable at ℓ as candidates for the hole. This is because we may copy function definitions in correct programs (via Define actions) into P and let P invoke the newly added functions. In case of no candidate variable of type τ, we just enumerate all the variables in V.

Over a buggy program P using the concretization function, we may obtain the following set $\mathcal{A}$ of edit actions.

$$\begin{aligned}\mathcal{A} = \{&\mathsf{Modify}(\ell, e) \mid \mathsf{Modify}(\ell, e_\square) \in \mathcal{T}_M, e \in \gamma_e^P(e_\square)\} \\ &\cup \{\mathsf{Insert}(\ell, p \to e) \mid \mathsf{Insert}(\ell, p \to e_\square) \in \mathcal{T}_M, e \in \gamma_e^P(e_\square)\} \\ &\cup \{\mathsf{Delete}(\ell, p \to e) \mid \mathsf{Delete}(\ell, p \to e) \in \mathcal{T}_M\} \\ &\cup \mathcal{T}_D.\end{aligned}$$

However, this method is not scalable in practice as the number of concretized edit actions is potentially exponential to the number of holes in the template (despite the type-based pruning). To reduce the number of candidates for the holes, we use the following improved concretization function $\tilde{\gamma}_e^P$, which is similarly defined as γ_e^P except for the following case:

$$\tilde{\gamma}_e^P(\square_\tau^\ell) = \begin{cases} V\mid_\ell & (V\mid_\ell \neq \emptyset) \\ \gamma_e^P(\square_\tau^\ell) & \text{otherwise} \end{cases}$$

where $V\mid_\ell$ is the set of variables in V that may take the same value reachable at label ℓ. Formally,

$$V\mid_\ell = \{x \in V \mid \mathsf{type}(x) = \tau, \exists \delta, \delta'.\ \sigma(x, \delta) \cap \sigma(\ell, \delta') \neq \emptyset\}.$$

Changing Annotated Types during Instantiation For ease of presentation, we have presented our instantiation method as if types associated with holes could never change after they were determined. In the actual implementation, we change the types during the course of instantiation. Whenever a hole is filled with a variable, we perform type inference to change types of the other holes accordingly.

Algorithm 1 The Cafe Algorithm

Input: A buggy program P_b
Input: A set of correct programs $\mathcal{P}_c$,
Input: A set of test cases $\mathcal{T}$
Output: A program P_c satisfying all the test cases in $\mathcal{T}$

1: $\mathcal{A} \leftarrow \emptyset$ ▷ Set of edit actions
2: $\mathcal{P} \leftarrow \mathcal{P}_c \cup \{P_b\}$
3: $\sigma \leftarrow$ result of the path-sensitive 0CFA on $\mathcal{P}$
4: $\Delta \leftarrow$ all calling contexts derivable from σ
5: Derive $\mathcal{M}$ from Δ ▷ $\mathcal{M} : FId \to FId$
6: $\mathcal{T} \leftarrow \mathsf{ExtractTemplates}(\mathcal{M}, \Delta, P_b, \mathcal{P}_c)$
7: **for** $T \in \mathcal{T}$ **do**
8: $\quad \mathcal{A} \leftarrow \mathcal{A} \cup \mathsf{InstantiateTemplate}(T, P_b, \sigma)$
9: $n \leftarrow 1$
10: **repeat** ▷ E: edit script comprising n edit actions
11: $\quad$ **for** each permutation E of n elements of $\mathcal{A}$ **do**
12: $\quad\quad P \leftarrow$ apply E into P_b
13: $\quad\quad$ **if** correct$(P, \mathcal{T})$ **then**
14: $\quad\quad\quad$ **return** P
15: $\quad n \leftarrow n + 1$
16: **until** $n \le |\mathcal{A}|$

2.2.3 Overall Algorithm

Putting all together, Algo. **??** depicts the Cafe algorithm. We first perform the path-sensitive 0CFA on all the programs and obtain the result σ (line **??**). Then, we derive calling contexts from the analysis result (line **??**). From the calling contexts, we obtain the matching function $\mathcal{M}$ that maps each function in the buggy program P_b to a function in a correct program that is most likely to be useful for repair (line **??**). Using the matching function, we collect repair templates (line **??**). By instantiating the templates, we obtain a set of edit actions that can be applicable to P_b (lines **??** – **??**). The main loop (lines **??** – **??**) applies each possible sequence of

the edit actions into P_b in turn. The variable n denotes the number of edit actions that can be used, which is initialized to be 1 (line **??**). We apply each edit script into P_b (line **??**) and check if the program has been fixed based on the given test cases (line **??**). If we have fixed the program, we return it as a final result (line **??**). Otherwise, we increase n by 1 (line **??**) and repeat the main loop.

The algorithm finds a *minimal* edit script that corrects the given buggy program.

Definition 2.2 (Minimality)**.** Given an incorrect program P_b and a set of possible edit actions $\mathcal{A}$, an edit script E comprising the edit actions in $\mathcal{A}$ to correct P_b is minimal if there does not exist an edit script E' such that $|E'| < |E|$ and E' fixes P_b.

2.2.4 Optimizations

When generating edit actions of kind Define that add new function definitions into a target buggy program, we avoid functions that incur a long subsequent call chain to prevent CAFE from generating huge patches (currently, we only consider immediate callees of a reference function). Additionally, when generating edit scripts by permuting edit actions, we avoid generating duplicated edit scripts that lead to the same effect by not respecting orders between edit actions targetting different labels.