Data Augmentation by Program Transformation——Appendix

Shiwen Yu, Ting Wang, Ji Wang*

State Key Laboratory for High Performance Computing
College of Computer Science and Technology

National University of Defense Technology

Changsha, China

1. Examples of Generalization Failures of Code2Vec

We have tested six examples listed on the website of Code2vec (`https://code2vec.org/`) at 2020.6.14 by applying SPAT rules. The results are listed in Figure 8. With slight modifications to code forms, Code2vec will fail to predict their method names. It indicates that Code2vec is very sensitive to code forms. Rather than analyzing the semantics of source code, it is more likely to remember certain method naming patterns or simply memorize the data.

A more detailed statistical experiment of generalization for slight form changes can be regarded as "Stability Testing" [1]. It measures the performance difference of deep learning models before and after code transformations to estimate their generalizability. Like the data in the training set and test set must not cross, the transformation rules used in augmenting the training set should not be used again in the test set. Otherwise, we can evaluate neither that the generalizability is improved by data augmentation nor that the model is stable in the stability testing.

*Corresponding author
Email addresses: `yushiwen14@nudt.edu.cn` (Shiwen Yu), `wj@nudt.edu.cn` (Ji Wang)

2. FLOW and Denotational Semantics

2.1. Formal Definitions

We list the context-free productions of FLOW as following:

$$S ::= \text{skip} \mid v := E \mid S;\, S \mid (\text{if } B \text{ then } S \text{ else } S) \mid (\text{while } B \text{ do } S)$$

$$B ::= (B \wedge B) \mid (\neg B) \mid (E = E) \mid (E < E) \mid t \mid f$$

$$E ::= (E + E) \mid (E - E) \mid (E * E) \mid z \mid v$$

Where S represents the statements of FLOW. All legal S group to a set named Sts; B represents boolean expressions. All legal B group to a set named $Bexp$; E represents arithmetic expressions. All legal E group to a set named $Eexp$; v belongs to the set of 26 lower case letters in English, representing variable identifiers; z belongs to the set of integers; t and f represent the boolean "true" and "false." S, B, and E are all non-terminal symbols; z and v are variables of terminal symbols; t and f are terminal symbols.

Conventionally, we use the symbols S, B, and E representing the fully extended source code, and use $X \in S$ to indicate that X can be a grammar sub-tree of S.

The domains and denotation sets we use are defined as follows:

1. $T_\perp$: the set of boolean values $\{1, 0, \perp\}$. The variable on it is marked b. $\perp$ is the symbol of the minimal element.
2. $Z_\perp$: the set of integers $\{-\infty, \ldots, 0, \ldots, +\infty\} \cup \{\perp\}$.
3. $V_\perp$: the set of letters $\{\text{'a'}, \text{'b'}, \ldots, \text{'z'}, \perp\}$.
4. $V_\perp \to_c Z_\perp$: the set of program states, which is the set of continuous functions (marked as $\to_c$) from letters to integers. We mark it as $State$, and the variable in it is named s.
5. $State \to_c T_\perp$: the set of the denotations of boolean expressions, which is the set of continuous functions from states to boolean values. We mark it as Bp.

6. $State \to_c Z_\perp$: the set of the denotations of arithmetic expressions, which is the set of continuous functions from states to integers. We mark it as Ep.
7. $State \to_c State$: the set of the denotations of program statements, which is the set of continuous functions from states to states. We mark it as Sp.

The semantic functions we use are defined as follows:

1. $\mathbb{B}$: $Bexp \to Bp$, which inputs a boolean expression and outputs a continuous function from a state to a boolean value.

$$\mathbb{B}(t)(s) =_{df} 1 \qquad \mathbb{B}(f)(s) =_{df} 0$$

$$\mathbb{B}((B_1 \wedge B_2))(s) =_{df} \begin{cases} 1 & \mathbb{B}(B_1)(s) = \mathbb{B}(B_2)(s) = 1 \\ 0 & else \end{cases}$$

$$\mathbb{B}((\neg B))(s) =_{df} \begin{cases} 0 & \mathbb{B}(B)(s) = 1 \\ 1 & \mathbb{B}(B)(s) = 0 \end{cases}$$

$$\mathbb{B}((E_1\ op_r\ E_2))(s) =_{df} \mathbb{E}(E_1)(s)\ op_r\ \mathbb{E}(E_2)(s)$$

where op_r is $<$ or $=$. When op_r is in the right hand, it means the functions "less than" or "equal": $Z_\perp \times Z_\perp \to_c T_\perp$.

2. $\mathbb{E}$: $Eexp \to Ep$, which inputs an arithmetic expression and outputs a continuous function from a state to an integer.

$$\mathbb{E}(z)(s) =_{df} z \qquad \mathbb{E}(v)(s) =_{df} s(v)$$

$$\mathbb{E}((E_1\ op_a\ E_2))(s) =_{df} \mathbb{E}(E_1)(s)\ op_a\ \mathbb{E}(E_2)(s)$$

where op_a is $+$, $-$, or $*$. op_a in the left hand is only a symbol, whereas op_a in the right hand represents the functions "plus", "minus", or "times": $Z_\perp \times Z_\perp \to_c Z_\perp$.

3. $\mathbb{S}$: $Sts \to Sp$, which inputs a program statement and outputs a continuous

function from one state to another.

$$\mathbb{S}(\text{skip}) =_{df} id \qquad \mathbb{S}(S_1;\ S_2) =_{df} \mathbb{S}(S_2) \circ \mathbb{S}(S_1)$$

$$\mathbb{S}(v := E)(s)(v') =_{df} \begin{cases} s(v') & v' \neq v \\ \mathbb{E}(E)(s) & v' = v \end{cases}$$

$$\mathbb{S}((\text{if } B \text{ then } S_1 \text{ else } S_2))$$

$$=_{df} cond(\mathbb{B}(B), \mathbb{S}(S_1), \mathbb{S}(S_2))$$

$$\mathbb{S}((\text{while } B \text{ do } S)) =_{df} fix(W)$$

$$W : Sp \to_c Sp$$

$$W(\theta) = cond(\mathbb{B}(B), \theta \circ \mathbb{S}(S), id)$$

where id is the identity function, $\circ$ is the composition operator, *cond* is the conditional operator, and fix is the minimum fixpoint operator. Put it simply, *cond* chooses to apply one of the last two functions by the boolean value of applying the first function. $fix(W)$ is solving the minimal fixpoint of W. Their formal definitions can be checked in Appendix 2.2.

2.2. Proof of Legitimacy

We prove the that all semantic functions we define are legal. That is, we need to prove that the output of all semantic functions must be continuous functions. Before doing so, we need to declare that all domains we use are CPO (Complete Partial Order): if $(\sqsubseteq, D)$ has a minimal element $(\bot, \forall d \in D \quad \bot \sqsubseteq d)$ and every ω chain in $(\sqsubseteq, D)$ has a supremum, then we say $\sqsubseteq$ is a CPO to D and $(\sqsubseteq, D)$ is a CPO set. ω chain is a sequence of elements:$d_1, d_2, d_3, \ldots$ where $d_i \sqsubseteq d_j$ if $i < j$. It is obvious: $T_\bot$ has only three element $\{1, 0, \bot\}$, and two ω chains $(\bot \sqsubseteq 1,\ \bot \sqsubseteq 0)$, forming into a flat order set (one kind of CPO set); likewise, $Z_\bot$, $V_\bot$, are also flat order sets.

Let $(\sqsubseteq_1, D)$ and $(\sqsubseteq_2, E)$ be CPO sets, and a function be $f : D \to E$. We

say f is continuous if and only if:

$$(d_1 \sqsubseteq_1 d_2) \Longrightarrow (f(d_1) \sqsubseteq_2 f(d_2)) \quad (d_1, d_2 \in D)$$

and for each ω chain $\{d_i\}$ in D:

$$f(\bigsqcup_i d_i) = \bigsqcup_i f(d_i)$$

We use $(D \rightarrow_c P)$ to represent the set that contains only and all continuous functions. We have a lemma:

Lemma 2.1. *if* $(\sqsubseteq_1, D)$ *and* $(\sqsubseteq_2, P)$ *are CPO sets, then* $(\sqsubseteq_3, (D \rightarrow_c P))$ *is also CPO sets, where* $f \in (D \rightarrow_c P)$, $f_1 \sqsubseteq_3 f_2$ *if and only if* $\forall d \in D \quad f_1(d) \sqsubseteq_2 f_2(d)$.

70 Therefore, *State*, *Bp*, *Ep*, and *Sp* are all CPOs sets.

Now, we prove the definitions of $\mathbb{B}$ and $\mathbb{E}$ are legal. Note that we have $(\exists B_1 \in B \quad \mathbb{B}(B_1) = \bot) \Longrightarrow \mathbb{B}(B) = \bot$ and $\mathbb{B}(\bot) = \bot$ conventional. E and S have the same property. First, it is obvious to say $\mathbb{B}(t)$ and $\mathbb{B}(f)$ are continuous.

Then, we prove the rest inductively. For $\mathbb{B}((B_1 \wedge B_2))$:

$(s_1 \sqsubseteq s_2)$

$\quad \Longrightarrow \mathbb{B}(B_1)(s_1) \sqsubseteq \mathbb{B}(B_1)(s_2) \wedge \mathbb{B}(B_2)(s_1) \sqsubseteq \mathbb{B}(B_2)(s_2)$

$\quad \Longrightarrow \mathbb{B}(B_1)(s_1) = \bot \wedge \mathbb{B}(B_2)(s_1) = \bot$

$\quad \Longrightarrow \mathbb{B}((B_1 \wedge B_2))(s_1) = \bot$

$\quad \Longrightarrow \mathbb{B}((B_1 \wedge B_2))(s_1) \sqsubseteq \mathbb{B}((B_1 \wedge B_2))(s_2)$

$\mathbb{B}(B_1)(\bigsqcup_i s_i) = \bigsqcup_i \mathbb{B}(B_1)(s_i) \wedge \mathbb{B}(B_2)(\bigsqcup_i s_i) = \bigsqcup_i \mathbb{B}(B_2)(s_i)$

When $\mathbb{B}(B_1)(\bigsqcup_i s_i) = \mathbb{B}(B_2)(\bigsqcup_i s_i) = 1$, we have:

$\mathbb{B}((B_1 \wedge B_2))(\bigsqcup_i s_i) = 1$

$\bigsqcup_i \mathbb{B}(B_1)(s_i) = \bigsqcup_i \mathbb{B}(B_2)(s_i) = 1$

$\bigsqcup_i \mathbb{B}((B_1 \wedge B_2))(s_i) = 1 = \mathbb{B}((B_1 \wedge B_2))(\bigsqcup_i s_i)$

Similarly, when $\mathbb{B}(B_1)(\bigsqcup_i s_i) \neq \mathbb{B}(B_2)(\bigsqcup_i s_i) \neq \bot$, we have:

$\mathbb{B}((B_1 \wedge B_2))(\bigsqcup_i s_i) = 0$

$\bigsqcup_i \mathbb{B}(B_1)(s_i) \neq \bigsqcup_i \mathbb{B}(B_2)(s_i) \neq \bot$

$\bigsqcup_i \mathbb{B}((B_1 \wedge B_2))(s_i) = 0 = \mathbb{B}((B_1 \wedge B_2))(\bigsqcup_i s_i)$

Also, when $\mathbb{B}(B_1)(\bigsqcup_i s_i) = \bot$ or $\mathbb{B}(B_2)(\bigsqcup_i s_i) = \bot$, we have:

$\mathbb{B}((B_1 \wedge B_2))(\bigsqcup_i s_i) = \bot$

$\bigsqcup_i \mathbb{B}(B_1)(s_i) = \bot \vee \bigsqcup_i \mathbb{B}(B_2)(s_i) = \bot$

$\bigsqcup_i \mathbb{B}((B_1 \wedge B_2))(s_i) = \bot = \mathbb{B}((B_1 \wedge B_2))(\bigsqcup_i s_i)$

Hence, we proved $\mathbb{B}((B_1 \wedge B_2))$ is a continuous function if $\mathbb{B}(B_1)$ and $\mathbb{B}(B_2)$ are both continuous functions. Similarly, we can prove that $\mathbb{B}((\neg B))$ is a continuous function if $\mathbb{B}(B)$ is. Also, $\mathbb{B}((E_1\ op_r\ E_2))$ is a continuous function if $\mathbb{E}(E_1)$ and $\mathbb{E}(E_2)$ are. Similarly, we can inductively prove the legitimacy of $\mathbb{E}$.

Next, we move to prove the legitimacy of $\mathbb{S}$. First, we have two lemmas:

Lemma 2.2. *If a function f can be constructed by other continuous functions with continuous operators (compose operator, curry operator, etc.), f is a continuous function.*

Lemma 2.3. *For all $f \in (D \to_c P)$, $fix(f)$ exists and is unique with respect to $\sqsubseteq_3$. Where $fix(f) =_{df} \bigsqcup_i f^i(\bot)$.*

Therefore, the inductive proof of the legitimacy of $\mathbb{S}$ is to construct $\mathbb{S}(S)$ by continuous operators:

$\mathbb{S}(S_1;\ S_2) = \mathbb{S}(S_2) \circ \mathbb{S}(S_1)$

$\mathbb{S}((\text{if } B \text{ then } S_1 \text{ else } S_2))$

$= cond(\mathbb{B}(B), \mathbb{S}(S_1), \mathbb{S}(S_2))$

$\mathbb{S}((\text{while } B \text{ do } S)) = fix(W)$

$W = curry(cond)(\mathbb{B}(B), id) \circ curry(\circ)(\mathbb{S}(S))$

Therefore:

$W : Sp \to_c Sp \quad \text{and} \quad fix(W) \in Sp$

Where $\circ$, *cond*, and *curry* are defined as following:

$f_1 \circ f_2(x) =_{df} f_1(f_2(x))$

$$cond(\mathbb{B}(B), f_1, f_2)(x) =_{df} \begin{cases} f_1(x) & \mathbb{B}(B) = 1 \\ f_2(x) & \mathbb{B}(B) = 0 \\ \bot & else \end{cases}$$

$curry(f_1)(x)(y) =_{df} f_1(x, y)$

For the beginning of induction, $\mathbb{S}(\text{skip}) = id$ is obviously continuous. We give

the proof of $\mathbb{S}(v := E)$ as following:

$(s_1 \sqsubseteq s_2)$

$\quad \Longrightarrow \forall v \quad s_1(v) \sqsubseteq s_2(v)$

$\quad \Longrightarrow \forall v \quad s_1(v) = s_2(v) \vee s_1(v) = \bot$

$\quad \Longrightarrow \forall v \quad \mathbb{S}(v_1 := E)(s_1)(v) \sqsubseteq \mathbb{S}(v_1 := E)(s_2)(v)$

$\quad \Longrightarrow \mathbb{S}(v_1 := E)(s_1) \sqsubseteq \mathbb{S}(v_1 := E)(s_2)$

when $v = v_1$:

$\mathbb{S}(v_1 := E)(\bigsqcup_i s_i)(v) = E = (\bigsqcup_i \mathbb{S}(v_1 := E)(s_i))(v)$

when $v \neq v_1$:

$\mathbb{S}(v_1 := E)(\bigsqcup_i s_i)(v) = (\bigsqcup_i s_i)(v) = (\bigsqcup_i \mathbb{S}(v_1 := E)(s_i))(v)$

Hence, we proved that $\mathbb{S}(v := E)$ is continuous. Inductively, we proved that $\mathbb{S}$ is legal. A more detailed formal proof can be checked in [2].

2.3. Proofs of Denotational Semantic Preserving

We list the 18 rules' C and R functions in Table 1.

2.3.1. LocalVarRenaming*$(a \to b)$*

We first need to extend the semantic function:

$\mathbb{S}((\text{dec } [L]\ S))(s) =_{df} restore(L, s) \circ \mathbb{S}(S) \circ declare(L)(s)$

$declare(L)(s) =_{df} cond(v \in L, Z_0, s)$

$restore(L, s_1)(s_2) =_{df} cond(v \in L, s_1, s_2)$

Where Z_0 is a constant value function that always outputs 0. It is easy to see that this definition is legal.

The target we are going to prove is:

$\mathbb{S}((\text{dec [a] } S_1)) = \mathbb{S}(R((\text{dec [a] } S_1)))$.

First, we define a condition: $Q_{\text{a}\to\text{b}}(s_1, s_2)$ is true if and only if for all $v \neq a \neq b$, $s_1(v) = s_2(v)$ and $s_1(a) = s_2(b)$. Next, we prove that $Q_{\text{a}\to\text{b}}(s_1, s_2) \Longrightarrow$

Table 1: Formal Definitions of 18 Rules

Rid	Name	ϵ ($p = 0.9$)	Applicability		#/sec
0	LocalVarRenaming(a $\rightarrow$ b)	ϵ = -3.2e-04	9133/9133	1	1,615
	$S ::= (\text{dec } [L]\ S)$ $L ::= v,\ L \mid v$ $C(S) =_{df} (\exists S_1 \quad (S = (\text{dec } [\text{a}]\ S_1)) \wedge (\text{b} \notin S_1))$ $R((\text{dec } [L]\ S)) =_{df} (\text{dec } [R(L)]\ R(S))$ $\quad R(v,\ L) =_{df} R(v),\ R(L)$ $R(v := E) =_{df} R(v) := R(E)$ $\quad R(\text{skip}) =_{df} \text{skip}$ $\quad R(S_1;\ S_2) =_{df} R(S_1);\ R(S_2)$ $R((\text{if } B \text{ then } S_1 \text{ else } S_2)) =_{df} (\text{if } R(B) \text{ then } R(S_1) \text{ else } R(S_2))$ $R((\text{while } B \text{ do } S)) =_{df} (\text{while } R(B) \text{ do } R(S))$ $R((B_1 \wedge B_2)) =_{df} (R(B_1) \wedge R(B_2))$ $\quad R((\neg B)) =_{df} (\neg R(B))$ $R((E_1\ op_r\ E_2)) =_{df} (R(E_1)\ op_r\ R(E_2))$ $\quad R(t) =_{df} t$ $\quad R(f) =_{df} f$ $R((E_1\ op_a\ E_2)) =_{df} (R(E_1)\ op_a\ R(E_2))$ $R(z) =_{df} z \quad R(v) = \begin{cases} \text{b} & v = \text{a} \\ v & else \end{cases}$				
1	For2While	ϵ = 1.6e-04	1684/9133	0.18	1,611
	$C(S) =_{df} (\exists B, S_1, S_2, S_3 \quad S = (\text{for } (S_1;\ B;\ S_2) \text{ do } S_3))$ $R((\text{for } (S_1;\ B;\ S_2) \text{ do } S_3)) =_{df} S_1;\ (\text{while } B \text{ do } S_3;\ S_2)$				
2	While2For	ϵ = -1.5e-06	3194/9133	0.35	1,607
	$C(S) =_{df} (\exists B, S_1 \quad S = (\text{while } B \text{ do } S_1))$ $R((\text{while } B \text{ do } S_1)) =_{df} (\text{for } (\text{skip};\ B;\ \text{skip}) \text{ do } S_1)$				
3	ReverseIfElse	ϵ = 4.3e-06	5998/9133	0.66	1,458
	$C(S) =_{df} (\exists B, S_1, S_2 \quad S = (\text{if } B \text{ then } S_1 \text{ else } S_2))$ $R((\text{if } B \text{ then } S_1 \text{ else } S_2)) =_{df} (\text{if } (\neg B) \text{ then } S_2 \text{ else } S_1)$				
4	SingleIF2ConditionalExp	ϵ = 4.4e-04	1544/9133	0.17	1,603
	$E ::= (B,\ E,\ E)$ $C(S) =_{df} (\exists B, v, E_1, E_2 \quad S = (\text{if } B \text{ then } v := E_1 \text{ else } v := E_2))$ $R((\text{if } B \text{ then } v := E_1 \text{ else } v := E_2)) =_{df} v := (B,\ E_1,\ E_2)$				
5	ConditionalExp2SingleIF	ϵ = 2.5e-04	340/9133	0.04	1,682
	$C(S) =_{df} (\exists B, v, E_1, E_2 \quad S = v := (B,\ E_1,\ E_2))$ $R(v := (B,\ E_1,\ E_2)) =_{df} (\text{if } B \text{ then } v := E_1 \text{ else } v := E_2)$				
6	PP2AddAssignment	ϵ = 2.7e-04	365/9133	0.04	1,695
	$S ::= v\ +\!= E \mid v{+}{+}$ $C(S) =_{df} (\exists v \quad S = v{+}{+}) \quad R(v{+}{+}) =_{df} v\ +\!= 1$				
7	AddAssignemnt2EqualAssignment	ϵ = -2.4e-05	870/9133	0.10	1,683
	$C(S) =_{df} (\exists v, E \quad S = v\ +\!= E) \quad R(v\ +\!= E) =_{df} v := (v + E)$				
8	InfixExpressionDividing	ϵ = 9.1e-06	1470/9133	0.16	1,611
	$C(S) =_{df} (\exists v_1, v_2, E_1, E_2, E_3, op_{a1}, op_{a2} \quad (S = v_1 := ((E_1\ op_{a1}\ E_2)\ op_{a2}\ E_3)) \wedge (v_2 \notin S))$ $R(v_1 := ((E_1\ op_{a1}\ E_2)\ op_{a2}\ E_3)) =_{df} (\text{dec } v_2\ v_2 := (E_1\ op_{a1}\ E_2);\ v_1 := (v_2\ op_{a2}\ E_3))$				
9	IfDividing	ϵ = 2.3e-07	8/9133	0.001	1,551
	$C(S) =_{df} (\exists B_1, B_2, S_1, S_2 \quad S = (\text{if } (B_1 \wedge B_2) \text{ then } S_1 \text{ else } S_2))$ $R((\text{if } (B_1 \wedge B_2) \text{ then } S_1 \text{ else } S_2)) =_{df} (\text{if } B_1 \text{ then } (\text{if } B_2 \text{ then } S_1 \text{ else } S_2) \text{ else } S_2)$				
10	StatementsOrderRearrangement	ϵ = 3.4e-04	6445/9133	0.71	1,326
	$C(S) =_{df} (\exists S_1, S_2 \quad ((S = S_1;\ S_2) \wedge \forall v \in S_1 \quad v \notin S_2))$ $R(S_1;\ S_2) =_{df} S_2;\ S_1$				
11	LoopIfContinue2Else	ϵ = -1.8e-06	201/9133	0.02	1,675
	$S ::= (\text{while } B \text{ do } S;\ (\text{if } B \text{ then continue else } S);\ S)$ $C(S) =_{df} (\exists B_1, B_2, S_1, S_2, S_3 \quad S = (\text{while } B_1 \text{ do } S_1;\ (\text{if } B_2 \text{ then continue else } S_2);\ S_3))$ $R((\text{while } B_1 \text{ do } S_1;\ (\text{if } B_2 \text{ then continue else } S_2);\ S_3)) =_{df} (\text{while } B_1 \text{ do } S_1;\ (\text{if } (\neg B_2) \text{ then } S_2;\ S_3 \text{ else skip}))$				
12	VarDeclarationMerging	ϵ = 1.5e-04	1471/9133	0.16	1,689
	$C(S) =_{df} (\exists S_1, v_1, v_2 \quad S = (\text{dec } [v_1]\ (\text{dec } [v_2]\ S_1)))$ $R((\text{dec } [v_1]\ (\text{dec } [v_2]\ S_1))) =_{df} (\text{dec } [v_1,\ v_2]\ S_1)$				
13	VarDeclarationDividing	ϵ = 3.2e-04	378/9133	0.04	1,688
	$C(S) =_{df} (\exists S_1, v_1, v_2 \quad S = (\text{dec } [v_1,\ v_2]\ S_1))$ $R((\text{dec } [v_1,\ v_2]\ S_1)) =_{df} (\text{dec } [v_1]\ (\text{dec } [v_2]\ S_1))$				
14	SwitchEqualSides	ϵ = 7.4e-05	2586/9133	0.28	1,691
	$C(B) =_{df} (\exists E_1, E_2 \quad B = (E_1 = E_2))$ $R((E_1 = E_2))) =_{df} (E_2 = E_1))$				
15	SwitchStringEqual	ϵ = 6.7e-05	596/9133	0.07	1,558
	$C(B) =_{df} (\exists E_1, E_2 \quad B = (E_1.euqals(E_2)))$ $R((E_1.euqals(E_2)))) =_{df} (E_2.euqals(E_1)))$				
16	PrePostFixExpressionDividing	ϵ = 2.1e-04	73/9133	0.007	1,602
	$C(S) =_{df} (\exists v_1, v_2, E_1, E_2, op \quad (S = v_1 := ((E_1 + +)\ op\ E_2)) \wedge (v_2 \notin S))$ $R(v_1 := ((E_1 + +)\ op\ E_2)) =_{df} (\text{dec } v_2\ v_2 := (E_1 + +);\ v_1 := (v_2\ op\ E_2))$				
17	Case2IfElse	ϵ = 1.4e-04	89/9133	0.01	1,581
	$S ::= (\text{switch } E\ (Q))$ $Q ::= (\text{case } E\ S) \mid (\text{case } E\ S)\ ;Q$ $C(S) =_{df} (\exists E, Q \quad S = (\text{switch } E\ (Q)))$ $R((\text{switch } E\ (Q))) =_{df} R(Q, E)$ $R((\text{case } E\ S), E_0) =_{df} (\text{if } (E_0 = E) \text{ then } S \text{ else skip}$ $R((\text{case } E\ S)\ ;Q, E_0) =_{df} (\text{if } (E_0 = E) \text{ then } S \text{ else } R(Q, E_0)$				

$\mathbb{E}(E)(s_1) = \mathbb{E}(R(E))(s_2)$ and $Q_{\mathrm{a}\to\mathrm{b}}(s_1, s_2) \Longrightarrow \mathbb{B}(B)(s_1) = \mathbb{B}(R(B))(s_2)$. Like before, we inductively prove them:

$Q_{\mathrm{a}\to\mathrm{b}}(s_1, s_2) \Longrightarrow \mathbb{E}(z)(s_1) = \mathbb{E}(R(z))(s_2)$

When $v = \mathrm{a}$

$Q_{\mathrm{a}\to\mathrm{b}}(s_1, s_2) \Longrightarrow \mathbb{E}(a)(s_1) = \mathbb{E}(b)(s_2) = \mathbb{E}(R(a))(s_2)$

$v = \mathrm{b}$ is not possible since $C(S) \Longrightarrow (\mathrm{b} \neq v))$

When $v \neq \mathrm{a} \neq \mathrm{b}$

$Q_{\mathrm{a}\to\mathrm{b}}(s_1, s_2) \Longrightarrow \mathbb{E}(v)(s_1) = \mathbb{E}(v)(s_2) = \mathbb{E}(R(v))(s_2)$

Also, when $\mathbb{E}(E_1)$ and $\mathbb{E}(E_2)$ have the above property, we will have:

$Q_{\mathrm{a}\to\mathrm{b}}(s_1, s_2)$

$\Longrightarrow \mathbb{E}(E_1)(s_1) = \mathbb{E}(R(E_1))(s_2)$

$\quad \wedge \mathbb{E}(E_2)(s_1) = \mathbb{E}(R(E_2))(s_2)$

$\Longrightarrow \mathbb{E}(E_1)(s_1)\ op_a\ \mathbb{E}(E_2)(s_1)$

$\quad = \mathbb{E}(R(E_1))(s_2)\ op_a\ \mathbb{E}(R(E_2))(s_2)$

$\Longrightarrow \mathbb{E}(E_1\ op_a\ E_2)(s_1) = \mathbb{E}(R(E_1)\ op_a\ R(E_2))(s_2)$

$\Longrightarrow \mathbb{E}(E_1\ op_a\ E_2)(s_1) = \mathbb{E}(R(E_1\ op_a\ E_2))(s_2)$

Similarly, we can prove B also have this property.

Then, we inductively prove that for all S:

$Q_{\mathrm{a}\to\mathrm{b}}(s_1, s_2) \Longrightarrow Q_{\mathrm{a}\to\mathrm{b}}(\mathbb{S}(S)(s_1), \mathbb{S}(R(S))(s_2))$

For the beginning statement $R(\mathrm{skip}) = \mathrm{skip}$:

$Q_{\mathrm{a}\to\mathrm{b}}(s_1, s_2)$

$\Longrightarrow Q_{\mathrm{a}\to\mathrm{b}}(id(s_1), id(s_2))$

$\Longrightarrow Q_{\mathrm{a}\to\mathrm{b}}(\mathbb{S}(R(\mathrm{skip}))(s_1), \mathbb{S}(\mathrm{skip})(s_2))$

For the beginning statement $R(v := E)) = R(v) := R(E)$:

when $v_1 \neq \mathrm{a} \neq \mathrm{b}$:

$R(v_1 := E) = v_1 := R(E)$

Also, we have:

$Q_{\mathrm{a}\to\mathrm{b}}(s_1, s_2) \Longrightarrow \mathbb{E}(E)(s_1) = \mathbb{E}(R(E))(s_2)$

Therefore:

$Q_{\mathrm{a}\to\mathrm{b}}(s_1, s_2)$

$\Longrightarrow Q_{\mathrm{a}\to\mathrm{b}}(\mathbb{S}(v_1 := E)(s_1), \mathbb{S}(v_1 := R(E))(s_2))$

$\Longrightarrow Q_{\mathrm{a}\to\mathrm{b}}(\mathbb{S}(v_1 := E)(s_1), \mathbb{S}(R(v_1 := E))(s_2))$

When $v_1 = \mathrm{a}$:

$Q_{\mathrm{a}\to\mathrm{b}}(s_1, s_2)$

$\Longrightarrow Q_{\mathrm{a}\to\mathrm{b}}(\mathbb{S}(a := E)(s_1), \mathbb{S}(b := R(E))(s_2))$

$\Longrightarrow Q_{\mathrm{a}\to\mathrm{b}}(\mathbb{S}(a := E)(s_1), \mathbb{S}(R(a := E))(s_2))$

$v = \mathrm{b}$ is not possible since $C(S) \Longrightarrow (\mathrm{b} \neq v))$

Similarly, when S_1, S_2, and B have the above properties, we have:

$Q_{\mathrm{a}\to\mathrm{b}}(s_1, s_2)$

$\Longrightarrow Q_{\mathrm{a}\to\mathrm{b}}(\mathbb{S}(S_1)(s_1), \mathbb{S}(R(S_1))(s_2))$

$\quad \wedge Q_{\mathrm{a}\to\mathrm{b}}(\mathbb{S}(S_2)(s_1), \mathbb{S}(R(S_2))(s_2))$

$\Longrightarrow Q_{\mathrm{a}\to\mathrm{b}}(\mathbb{S}(S_1)(s_1), \mathbb{S}(R(S_1))(s_2))$

$\Longrightarrow Q_{\mathrm{a}\to\mathrm{b}}(\mathbb{S}(S_2)(\mathbb{S}(S_1)(s_1)), \mathbb{S}(R(S_2))(\mathbb{S}(R(S_1))(s_2)))$

$\Longrightarrow Q_{\mathrm{a}\to\mathrm{b}}(\mathbb{S}(S_2) \circ \mathbb{S}(S_1)(s_1), \mathbb{S}(R(S_2)) \circ \mathbb{S}(R(S_1))(s_2))$

$\Longrightarrow Q_{\mathrm{a}\to\mathrm{b}}(\mathbb{S}(S_2;\ S_1)(s_1), \mathbb{S}(R(S_2);\ R(S_1))(s_2))$

$\Longrightarrow Q_{\mathrm{a}\to\mathrm{b}}(\mathbb{S}(S_2;\ S_1)(s_1), \mathbb{S}(R(S_2;\ S_1))(s_2))$

In the same way, we can prove:

$Q_{\mathrm{a}\to\mathrm{b}}(s_1, s_2)$

$\Longrightarrow Q_{\mathrm{a}\to\mathrm{b}}(\mathbb{S}(S_1)(s_1), \mathbb{S}(R(S_1))(s_2))$

$\quad \wedge Q_{\mathrm{a}\to\mathrm{b}}(\mathbb{S}(S_2)(s_1), \mathbb{S}(R(S_2))(s_2))$

$\quad \wedge \mathbb{B}(B)(s_1) = \mathbb{B}(R(B))(s_2)$

$\Longrightarrow Q_{\mathrm{a}\to\mathrm{b}}(\mathbb{S}((\text{if } B \text{ then } S_1 \text{ else } S_2))(s_1),$

$\qquad \mathbb{S}((\text{if } R(B) \text{ then } R(S_1) \text{ else } R(S_2)))(s_2))$

$\Longrightarrow Q_{\mathrm{a}\to\mathrm{b}}(\mathbb{S}((\text{if } B \text{ then } S_1 \text{ else } S_2))(s_1),$

$\qquad \mathbb{S}(R((\text{if } B \text{ then } S_1 \text{ else } S_2)))(s_2))$

Also, it is easy to see:

$Q_{\mathrm{a}\to\mathrm{b}}(s_1, s_2)$

$\Longrightarrow Q_{\mathrm{a}\to\mathrm{b}}(\mathbb{S}((\text{dec } [L]\ S_1))(s_1),$

$\qquad \mathbb{S}(R((\text{dec } [L]\ S_1)))(s_2))$

$\Longrightarrow Q_{\mathrm{a}\to\mathrm{b}}(\mathbb{S}((\text{while } B \text{ do } S_1))(s_1),$

$\qquad \mathbb{S}(R((\text{while } B \text{ do } S_1)))(s_2))$

Now, we proved that all S, E, and B have the $Q_{\mathrm{a}\to\mathrm{b}}$ properties. It is easy to

see $Q_{\mathrm{a}\to\mathrm{b}}(declare(\mathrm{a})(s), declare(R(\mathrm{a}))(s))$ is true. Also, we have $Q_{\mathrm{a}\to\mathrm{b}}(s_1, s_2) \wedge s_1(\mathrm{b}) = s(\mathrm{b}) \wedge s_2(\mathrm{a}) = s(\mathrm{a}) \Longrightarrow restore(\mathrm{a}, s)(s_1) = restore(\mathrm{b}, s)(s_2)$.

Therefore, we have:

$Q_{\mathrm{a}\to\mathrm{b}}(declare(\mathrm{a})(s), declare(R(\mathrm{a}))(s))$

$\Longrightarrow Q_{\mathrm{a}\to\mathrm{b}}(\mathbb{S}(S) \circ declare(\mathrm{a})(s), \mathbb{S}(R(S)) \circ declare(R(\mathrm{a}))(s))$

Additionally, we have: $\mathrm{b} \notin S \wedge \mathrm{a} \notin R(S)$. Therefore:

$\mathbb{S}(S)(s)(\mathrm{b}) = s(\mathrm{b}) \wedge \mathbb{S}(R(S))(s)(\mathrm{a}) = s(\mathrm{a})$

$\mathbb{S}(S) \circ declare(\mathrm{a})(s)(\mathrm{b}) = s(\mathrm{b})$

$\quad \wedge \mathbb{S}(R(S)) \circ declare(\mathrm{b})(s)(\mathrm{a}) = s(\mathrm{a})$

$Q_{\mathrm{a}\to\mathrm{b}}(\mathbb{S}(S) \circ declare(\mathrm{a})(s), \mathbb{S}(R(S)) \circ declare(R(\mathrm{a}))(s))$

$\Longrightarrow restore(\mathrm{a}, s)(\mathbb{S}(S) \circ declare(\mathrm{a})(s))$

$\quad = restore(\mathrm{b}, s)(\mathbb{S}(R(S)) \circ declare(R(\mathrm{a}))(s))$

$\Longrightarrow restore(\mathrm{a}, s) \circ \mathbb{S}(S) \circ declare(\mathrm{a})(s)$

$\quad = restore(R(\mathrm{a}), s) \circ \mathbb{S}(R(S)) \circ declare(R(\mathrm{a}))(s)$

$\Longrightarrow \mathbb{S}((\text{dec } [\mathrm{a}]\ S))(s) = \mathbb{S}((\text{dec } [R(\mathrm{a})]\ R(S)))(s)$

$\Longrightarrow \mathbb{S}((\text{dec } [\mathrm{a}]\ S)) = \mathbb{S}(R((\text{dec } [\mathrm{a}]\ S)))$

Hence, we proved that rule LocalVarRenaming is semantic-preserving.

2.3.2. For2While

Note that we need to extend FLOW with one extra production: $S :=$ (for $(S;\ B;\ S)$ do S) to support "for" loop statement. Also, we need to define the semantic function for this production:

$$\mathbb{S}((\text{for } (S_1;\ B;\ S_2) \text{ do } S_3)) =_{df} fix(W') \circ \mathbb{S}(S_1)$$

$$W'(\theta) = cond(\mathbb{B}(B), \theta \circ \mathbb{S}(S_2) \circ \mathbb{S}(S_3), id)$$

The target we are going to prove is:

$\mathbb{S}((\text{for } (S_1;\ B;\ S_2) \text{ do } S_3))$

$\quad = \mathbb{S}(S_1;\ (\text{while } B \text{ do } S_3;\ S_2))$

Notice that:

$\mathbb{S}((\text{for } (S_1;\ B;\ S_2) \text{ do } S_3)) = fix(W') \circ \mathbb{S}(S_1)$

$W'(\theta) = cond(\mathbb{B}(B), \theta \circ \mathbb{S}(S_2) \circ \mathbb{S}(S_3), id)$

We also have:

$\mathbb{S}(S_1;\ (\text{while } B \text{ do } S_3;\ S_2))$

$= \mathbb{S}((\text{while } B \text{ do } S_3;\ S_2)) \circ \mathbb{S}(S_1)$

$= fix(W) \circ \mathbb{S}(S_1)$

$W(\theta) = cond(\mathbb{B}(B), \theta \circ \mathbb{S}(S_3;\ S_2), id)$

$= cond(\mathbb{B}(B), \theta \circ \mathbb{S}(S_2) \circ \mathbb{S}(S_3), id)$

Therefore:

$W = W'$

$fix(W) = fix(W')$

$fix(W) \circ \mathbb{S}(S_1) = fix(W') \circ \mathbb{S}(S_1)$

$\mathbb{S}(S_1;\ (\text{while } B \text{ do } S_3;\ S_2))$

$= \mathbb{S}((\text{for } (S_1;\ B;\ S_2) \text{ do } S_3))$

Hence, we proved that the rule For2While is semantic-preserving.

2.3.3. While2For

The target we are going to prove is:

$\mathbb{S}((\text{while } B \text{ do } S_1)) = \mathbb{S}((\text{for } (\text{skip};\ B;\ \text{skip}) \text{ do } S_1))$

The proof is quite direct:

$\mathbb{S}((\text{for } (\text{skip};\ B;\ \text{skip}) \text{ do } S_1)) = fix(W') \circ \mathbb{S}(\text{skip})$

$= fix(W') \circ id$

$= fix(W')$

$W'(\theta) = cond(\mathbb{B}(B), \theta \circ \mathbb{S}(\text{skip}) \circ \mathbb{S}(S_1), id)$

$= cond(\mathbb{B}(B), \theta \circ id \circ \mathbb{S}(S_1), id)$

$= cond(\mathbb{B}(B), \theta \circ \mathbb{S}(S_1), id)$

$W(\theta) = cond(\mathbb{B}(B), \theta \circ \mathbb{S}(S_1), id)$

$= W'(\theta)$

$\mathbb{S}((\text{while } B \text{ do } S_1)) = fix(W)$

$= fix(W')$

$= \mathbb{S}((\text{for } (\text{skip};\ B;\ \text{skip}) \text{ do } S_1))$

Therefore, we proved that the rule While2For is semantic-preserving.

2.3.4. ReverseIfElse

The target we are going to prove is:

$\mathbb{S}((\text{if } B \text{ then } S_1 \text{ else } S_2)) = \mathbb{S}((\text{if } (\neg B) \text{ then } S_2 \text{ else } S_1))$

We have:

$\mathbb{S}((\text{if } B \text{ then } S_1 \text{ else } S_2)) = cond(\mathbb{B}(B), \mathbb{S}(S_1), \mathbb{S}(S_2))$

We also have:

$\mathbb{S}((\text{if } (\neg B) \text{ then } S_2 \text{ else } S_1)) = cond(\mathbb{B}((\neg B)), \mathbb{S}(S_2), \mathbb{S}(S_1))$

when $\mathbb{B}(B)(s) = 1$:

$\mathbb{S}((\text{if } B \text{ then } S_1 \text{ else } S_2))(s) = \mathbb{S}(S_1)(s)$

$\mathbb{S}((\text{if } (\neg B) \text{ then } S_2 \text{ else } S_1))(s) = \mathbb{S}(S_1)(s)$

$= \mathbb{S}((\text{if } B \text{ then } S_1 \text{ else } S_2))(s)$

Similarly, when $\mathbb{B}(B)(s) = 0$:

$\mathbb{S}((\text{if } (\neg B) \text{ then } S_2 \text{ else } S_1))(s) = \mathbb{S}(S_2)(s)$

$= \mathbb{S}((\text{if } B \text{ then } S_1 \text{ else } S_2))(s)$

Therefore:

$\mathbb{S}((\text{if } (\neg B) \text{ then } S_2 \text{ else } S_1)) = \mathbb{S}((\text{if } B \text{ then } S_1 \text{ else } S_2))$

Hence, we proved that the rule ReverseIfElse is semantic-preserving.

2.3.5. SingleIF2ConditionalExp

First we need to extend the semantic function:

$\mathbb{E}((B,\ E_1,\ E_2)) =_{df} cond(\mathbb{B}(B), \mathbb{E}(E_1), \mathbb{E}(E_2))$

It is easy to tell that this definition is legal. Then, the target we are going to prove is:

$\mathbb{S}((\text{if } B \text{ then } v := E_1 \text{ else } v := E_2)) = \mathbb{S}(v := (B,\ E_1,\ E_2))$

Note that:

$\mathbb{S}((\text{if } B \text{ then } v := E_1 \text{ else } v := E_2))$

$= cond(\mathbb{B}(B), \mathbb{S}(v := E_1), \mathbb{S}(v := E_2))$

When $\mathbb{B}(B)(s) = 1$:

$\mathbb{S}((\text{if } B \text{ then } v := E_1 \text{ else } v := E_2))(s) = \mathbb{S}(v := E_1)(s)$

And when $v' = v$:

$\mathbb{S}((\text{if } B \text{ then } v := E_1 \text{ else } v := E_2))(s)(v') = \mathbb{E}(E_1)(s)$

$$\mathbb{S}(v := (B,\, E_1,\, E_2))(s)(v') = \mathbb{E}((B,\, E_1,\, E_2))(s)$$
$$= \mathbb{E}(E_1)(s)$$
$$= \mathbb{S}((\text{if } B \text{ then } v := E_1 \text{ else } v := E_2))(s)(v')$$

Also when $v' \neq v$:

$$\mathbb{S}((\text{if } B \text{ then } v := E_1 \text{ else } v := E_2))(s)(v') = s(v')$$
$$\mathbb{S}(v := (B,\, E_1,\, E_2))(s)(v') = s(v')$$
$$= \mathbb{S}((\text{if } B \text{ then } v := E_1 \text{ else } v := E_2))(s)(v')$$

Therefore:

$$\mathbb{S}(v := (B,\, E_1,\, E_2))(s)$$
$$= \mathbb{S}((\text{if } B \text{ then } v := E_1 \text{ else } v := E_2))(s)$$

Similarly, when $\mathbb{B}(B)(s) = 0$, it is easy to see:

$$\mathbb{S}(v := (B,\, E_1,\, E_2))(s)$$
$$= \mathbb{S}((\text{if } B \text{ then } v := E_1 \text{ else } v := E_2))(s)$$

Therefore:

$$\mathbb{S}(v := (B,\, E_1,\, E_2)) = \mathbb{S}((\text{if } B \text{ then } v := E_1 \text{ else } v := E_2))$$

Hence, we proved that the rule SingleIF2ConditionalExp is semantic-preserving.

2.3.6. ConditionalExp2SingleIF

Its proof is the same as rule SingleIF2ConditionalExp.

2.3.7. PP2AddAssignment

First we need to extend the semantic function:

$$\mathbb{S}(v \mathrel{+}= E)(s)(v') =_{df} \begin{cases} s(v') & v' \neq v \\ \mathbb{E}(E)(s) + \mathbb{E}(v)(s) & v' = v \end{cases}$$

$$\mathbb{S}(v{+}{+})(s)(v') =_{df} \begin{cases} s(v') & v' \neq v \\ 1 + \mathbb{E}(v)(s) & v' = v \end{cases}$$

It is easy to tell that these definitions are legal. Then,obviously, we have:

$$\mathbb{S}(v \mathrel{+}= 1)(s)(v') = \mathbb{S}(v{+}{+})(s)(v')$$

Hence:

$$\mathbb{S}(v \mathrel{+}= 1) = \mathbb{S}(v{+}{+})$$

Therefore, we proved that the rule PP2AddAssignment is semantic-preserving.

2.3.8. AddAssignemnt2EqualAssignment

The target we are going to prove is:

$\mathbb{S}(v \mathrel{+}= E) = \mathbb{S}(v := (v + E))$

According to the definitions, it is easy to see:

$\mathbb{S}(v \mathrel{+}= E)(s)(v') = \mathbb{S}(v := (v + E))(s)(v')$

Hence, we proved that the rule AddAssignemnt2EqualAssignment is semantic-preserving.

2.3.9. InfixExpressionDividing

The target we are going to prove is:

$\mathbb{S}(v_1 := ((E_1\ op_{a1}\ E_2)\ op_{a2}\ E_3))$

$= \mathbb{S}(\text{dec } v_2\ v_2 := (E_1\ op_{a1}\ E_2);\ v_1 := (v_2\ op_{a2}\ E_3)))$

For any s and v, when $v = v_1$:

$v \neq v_2$ since $C(S) \Longrightarrow v_2 \neq v_1$

$\mathbb{S}(v_1 := ((E_1\ op_{a1}\ E_2)\ op_{a2}\ E_3))(s)(v)$

$= \mathbb{E}(((E_1\ op_{a1}\ E_2)\ op_{a2}\ E_3))(s)$

$= (\mathbb{E}(E_1)(s)\ op_{a1}\ \mathbb{E}(E_2)(s))\ op_{a2}\ \mathbb{E}(E_3)(s)$

also, we have:

$\mathbb{S}(\text{dec } v_2\ v_2 := (E_1\ op_{a1}\ E_2);\ v_1 := (v_2\ op_{a2}\ E_3)))(s)(v)$

$= restore(v_2, s)$

$\circ\ \mathbb{S}(v_2 := (E_1\ op_{a1}\ E_2);\ v_1 := (v_2\ op_{a2}\ E_3))$

$\circ\ declare(v_2)(s)(v)$

$= restore(v_2, s)$

$\circ\ \mathbb{S}(v_1 := (v_2\ op_{a2}\ E_3)) \circ \mathbb{S}(v_2 := (E_1\ op_{a1}\ E_2))$

$\circ\ declare(v_2)(s)(v) \qquad (v \neq v_2)$

$= \mathbb{E}((v_2\ op_{a2}\ E_3))(\mathbb{S}(v_2 := (E_1\ op_{a1}\ E_2))$

$\circ\ declare(v_2)(s))$

$= \mathbb{E}(v_2)(\mathbb{S}(v_2 := (E_1\ op_{a1}\ E_2)) \circ declare(v_2)(s))$

op_{a2}

$$
\begin{aligned}
&\mathbb{E}(E_3)(\mathbb{S}(v_2 := (E_1\ op_{a1}\ E_2)) \circ declare(v_2)(s)) \\
=\ &\mathbb{S}(v_2 := (E_1\ op_{a1}\ E_2)) \circ declare(v_2)(s)(v_2) \\
&op_{a2} \\
&\mathbb{E}(E_3)(s) \qquad\qquad (v_2 \notin E_3) \\
=\ &\mathbb{E}((E_1\ op_{a1}\ E_2))(declare(v_2)(s)) \\
&op_{a2} \\
&\mathbb{E}(E_3)(s) \\
=\ &\mathbb{E}((E_1\ op_{a1}\ E_2))(s) \quad (v_2 \notin E_2 \wedge v_2 \notin E_1) \\
&op_{a2} \\
&\mathbb{E}(E_3)(s) \\
=\ &(\mathbb{E}(E_1)(s)\ op_{a1}\ \mathbb{E}(E_2)(s))\ op_{a2}\ \mathbb{E}(E_3)(s) \\
=\ &\mathbb{S}(v_1 := ((E_1\ op_{a1}\ E_2)\ op_{a2}\ E_3))(s)(v)
\end{aligned}
$$

Similarly, for any s and v, when $v \neq v_1$, we can prove:

$$
\begin{aligned}
&\mathbb{S}(\text{dec } v_2\ v_2 := (E_1\ op_{a1}\ E_2);\ v_1 := (v_2\ op_{a2}\ E_3)))(s)(v) \\
=\ &\mathbb{S}(v_1 := ((E_1\ op_{a1}\ E_2)\ op_{a2}\ E_3))(s)(v)
\end{aligned}
$$

Hence, we proved that the rule InfixExpressionDividing is semantic-preserving.

2.3.10. IfDividing

The target we are going to prove is:

$$
\begin{aligned}
&\mathbb{S}((\text{if } (B_1 \wedge B_2) \text{ then } S_1 \text{ else } S_2)) \\
=\ &\mathbb{S}((\text{if } B_1 \text{ then } (\text{if } B_2 \text{ then } S_1 \text{ else } S_2) \text{ else } S_2))
\end{aligned}
$$

When $\mathbb{B}(B_1)(s) = \mathbb{B}(B_2)(s) = 1$:

$$
\begin{aligned}
&\mathbb{S}((\text{if } (B_1 \wedge B_2) \text{ then } S_1 \text{ else } S_2))(s) \\
=\ &cond(\mathbb{B}((B_1 \wedge B_2)), \mathbb{S}(S_1), \mathbb{S}(S_2))(s) \\
=\ &\mathbb{S}(S_1)(s)
\end{aligned}
$$

We also have:

$$
\begin{aligned}
&\mathbb{S}((\text{if } B_1 \text{ then } (\text{if } B_2 \text{ then } S_1 \text{ else } S_2) \text{ else } S_2))(s) \\
=\ &cond(\mathbb{B}(B_1), cond(\mathbb{B}(B_2), \mathbb{S}(S_1), \mathbb{S}(S_2)), \mathbb{S}(S_2))(s) \\
=\ &cond(\mathbb{B}(B_2), \mathbb{S}(S_1), \mathbb{S}(S_2))(s) \\
=\ &\mathbb{S}(S_1)(s) \\
=\ &\mathbb{S}((\text{if } (B_1 \wedge B_2) \text{ then } S_1 \text{ else } S_2))(s)
\end{aligned}
$$

When $\mathbb{B}(B_1)(s) \neq 1 \vee \mathbb{B}(B_2)(s) \neq 1$:

$\mathbb{S}((\text{if } (B_1 \wedge B_2) \text{ then } S_1 \text{ else } S_2))(s)$

$= cond(\mathbb{B}((B_1 \wedge B_2)), \mathbb{S}(S_1), \mathbb{S}(S_2))(s)$

$= \mathbb{S}(S_2)(s)$

Also if $\mathbb{B}(B_1)(s) \neq 1$:

$\mathbb{S}((\text{if } B_1 \text{ then } (\text{if } B_2 \text{ then } S_1 \text{ else } S_2) \text{ else } S_2))(s)$

$= cond(\mathbb{B}(B_1), cond(\mathbb{B}(B_2), \mathbb{S}(S_1), \mathbb{S}(S_2)), \mathbb{S}(S_2))(s)$

$= \mathbb{S}(S_2)(s)$

$= \mathbb{S}((\text{if } (B_1 \wedge B_2) \text{ then } S_1 \text{ else } S_2))(s)$

If $\mathbb{B}(B_1)(s) = 1 \wedge \mathbb{B}(B_2)(s) \neq 1$:

$\mathbb{S}((\text{if } B_1 \text{ then } (\text{if } B_2 \text{ then } S_1 \text{ else } S_2) \text{ else } S_2))(s)$

$= cond(\mathbb{B}(B_1), cond(\mathbb{B}(B_2), \mathbb{S}(S_1), \mathbb{S}(S_2)), \mathbb{S}(S_2))(s)$

$= cond(\mathbb{B}(B_2), \mathbb{S}(S_1), \mathbb{S}(S_2))(s)$

$= \mathbb{S}(S_2)(s)$

$= \mathbb{S}((\text{if } (B_1 \wedge B_2) \text{ then } S_1 \text{ else } S_2))(s)$

Therefore, we proved that the rule IfDividing is semantic-preserving.

2.3.11. StatementsOrderRearrangement

The target we are going to prove is:

$\mathbb{S}(S_1; S_2) = \mathbb{S}(S_2; S_1)$

Note that we have:

$C(S) \Longrightarrow \forall v \in S_1 \quad v \notin S_2$

We inductively prove it:

It is easy to see that if $S_1 = \text{skip}$ or $S_2 = \text{skip}$, it holds.

If $S_1 = v_1 := E_1$ and $S_2 = v_2 := E_2$:

$v_1 \neq v_2 \wedge v_1 \notin E_2 \wedge v_2 \notin E_1$

It is easy to see:

$\mathbb{S}(v_1 := E_1; v_2 := E_2)(s)(v)$

$= \mathbb{S}(v_2 := E_2; v_1 := E_1)(s)(v)$

Inductively, we can prove that all S_1 and S_2 have this property. That is:

$\forall v \in S_1 \quad v \notin S_2 \Longrightarrow \mathbb{S}(S_1; S_2)(s)(v) = \mathbb{S}(S_2; S_1)(s)(v)$

Therefore, we proved that the rule StatementsOrderRearrangement is semantic-preserving.

2.3.12. LoopIfContinue2Else

First we need to extend the semantic function:

$\mathbb{S}$((while B_1 do S_1; (if B_2 then continue else S_2); S_3))

$=_{df}$ $fix(W)$

$W(\theta) =_{df}$

$cond(\mathbb{B}(B_1), cond(\mathbb{B}(B_2), \theta, \theta \circ \mathbb{S}(S_3) \circ \mathbb{S}(S_2)) \circ \mathbb{S}(S_1), id)$

This definition is obviously legal. The target we are going to prove is:

$\mathbb{S}$((while B_1 do S_1; (if B_2 then continue else S_2); S_3))

$= \mathbb{S}$((while B_1 do S_1; (if $(\neg B_2)$ then S_2; S_3 else skip)))

Note that we have:

$\mathbb{S}$((while B_1 do S_1; (if $(\neg B_2)$ then S_2; S_3 else skip)))

$= fix(W')$

$W'(\theta) = cond(\mathbb{B}(B_1), \theta \circ cond(\mathbb{B}((\neg B_2)), \mathbb{S}(S_3) \circ \mathbb{S}(S_2), id) \circ \mathbb{S}(S_1), id)$

$= cond(\mathbb{B}(B_1), cond(\mathbb{B}((\neg B_2)), \theta \circ \mathbb{S}(S_3) \circ \mathbb{S}(S_2), \theta) \circ \mathbb{S}(S_1), id)$

$= cond(\mathbb{B}(B_1), cond(\mathbb{B}(B_2), \theta, \theta \circ \mathbb{S}(S_3) \circ \mathbb{S}(S_2)) \circ \mathbb{S}(S_1), id)$

$= W(\theta)$

Therefore, we have:

$fix(W') = fix(W)$

$\mathbb{S}$((while B_1 do S_1; (if B_2 then continue else S_2); S_3))

$= \mathbb{S}$((while B_1 do S_1; (if $(\neg B_2)$ then S_2; S_3 else skip)))

Hence, we proved that the rule LoopIfContinue2Else is semantic-preserving.

2.3.13. VarDeclarationMerging

The target we are going to prove is:

$\mathbb{S}$((dec [v_1] (dec [v_2] S_1))) = $\mathbb{S}$((dec [v_1, v_2] S_1))

First, we prove:

$declare(v_2) \circ declare(v_1) = declare(v_1, v_2)$

For any s and v, if $v \neq v_1 \wedge v \neq v_2$:

$declare(v_2) \circ declare(v_1)(s)(v) = s(v)$

$declare(v_1,\ v_2)(s)(v) = s(v)$

$declare(v_2) \circ declare(v_1)(s)(v) = declare(v_1,\ v_2)(s)(v)$

else if $v = v_1 \vee v = v_2$:

$declare(v_2) \circ declare(v_1)(s)(v) = 0$

$declare(v_1,\ v_2)(s)(v) = 0$

$declare(v_2) \circ declare(v_1)(s)(v) = declare(v_1,\ v_2)(s)(v)$

Therefore, we proved $declare(v_2) \circ declare(v_1) = declare(v_1, v_2)$.

Similarly, we can prove $restore(v_2, s) \circ restore(v_1, s) = restore(v_1,\ v_2, s)$.

Note that:

$$\begin{aligned}
&\mathbb{S}((\text{dec } [v_1] \ (\text{dec } [v_2] \ S_1)))(s) \\
&\quad = restore(v_1, s) \circ \mathbb{S}((\text{dec } [v_2] \ S_1)) \circ declare(v_1)(s) \\
&\quad = restore(v_1, s) \circ restore(v_2, s) \circ \mathbb{S}(S_1) \circ declare(v_2) \circ declare(v_1)(s) \\
&\quad = restore(v_1,\ v_2, s) \circ \mathbb{S}(S_1) \circ declare(v_2,\ v_1)(s) \\
&\quad = \mathbb{S}((\text{dec } [v_1,\ v_2] \ S_1))(s)
\end{aligned}$$

Hence, we proved that the rule VarDeclarationMerging is semantic-preserving.

2.3.14. VarDeclarationDividing

This proof is the same as rule VarDeclarationMerging.

2.3.15. SwitchEqualSides

The target we are going to prove is:

$\mathbb{B}((E_1 = E_2)) = \mathbb{B}((E_2 = E_1))$

We have:

$$\begin{aligned}
\mathbb{B}((E_1 = E_2))(s) = &\quad \mathbb{E}(E_1)(s) = \mathbb{E}(E_2)(s) \\
= &\quad \mathbb{E}(E_2)(s) = \mathbb{E}(E_1)(s) \\
= &\quad \mathbb{B}((E_2 = E_1))(s)
\end{aligned}$$

Therefore, we proved the rule SwitchEqualSides is semantic-preserving.

2.3.16. SwitchStringEqual

This proof is similar to the rule SwitchEqualSides.

2.3.17. PrePostFixExpressionDividing

This proof is similar to the rule InfixExpressionDividing.

2.3.18. Case2IfElse

The target we are going to prove is:

$\mathbb{S}((\text{switch } E\ (Q))) = \mathbb{S}(R(Q, E))$

We first define the semantic function of the "switch-statement" as following:

$\mathbb{S}((\text{switch } E\ (Q))) =_{df} \mathbb{S}(Q, E)$

$\mathbb{S}((\text{case } E\ S), E_0) =_{df} cond(\mathbb{B}(E_0 = E), \mathbb{S}(S), id)$

$\mathbb{S}((\text{case } E\ S)\ ; Q, E_0) =_{df} cond(\mathbb{B}(E_0 = E), \mathbb{S}(S), \mathbb{S}(Q, E_0))$

In the same time, we can have the following by the definition of the transformation function.

$$\begin{aligned}\mathbb{S}(R((\text{case } E\ S), E_0)) &= \mathbb{S}((\text{if } (E_0 = E) \text{ then } S \text{ else skip}))\\ &= cond(\mathbb{B}(E_0 = E), \mathbb{S}(S), id)\\ &= \mathbb{S}((\text{case } E\ S), E_0)\end{aligned}$$

Suppose we have:

$\mathbb{S}(R(Q, E_0)) = \mathbb{S}(Q, E_0)$

Then, we can have:

$$\begin{aligned}\mathbb{S}(R((\text{case } E\ S)\ ; Q, E_0)) &= \mathbb{S}((\text{if } (E_0 = E) \text{ then } S \text{ else } R(Q, E_0)))\\ &= cond(\mathbb{B}(E_0 = E), \mathbb{S}(S), \mathbb{S}(R(Q, E_0)))\\ &= cond(\mathbb{B}(E_0 = E), \mathbb{S}(S), \mathbb{S}(Q, E_0))\\ &= \mathbb{S}((\text{case } E\ S)\ ; Q, E_0)\end{aligned}$$

Therefore, inductively, we proved the target.

3. Implementation of SPAT

In order to verify our hypothesis as well as to provide a convenient tool for the following research, we developed the big code data augmentation tool SPAT. SPAT is designed with two layers: the first layer deals with all kinds of IO, parallelization, and syntax parsing/editing/printing with the help of JDT; the second layer contains the transformation rules. A rule consists of two parts: a group of conditions (*cons*, corresponding to C) to locate the transformable

sub-trees in an Abstract Syntax Tree (AST) and a sequence of edits (node-level) (*edits*, corresponding to R) to rewrite the sub-trees.

The procedure of SPAT can be seen as a pipeline. First, we parse the original source code snippet c into an AST. Then, the sub-trees in AST will be iterated, and those who satisfy the *cons* of the transformation rule will be recorded (we only record the first sub-tree if multiple transformable sub-trees are overlapped). Next, new sub-trees are generated from the recorded sub-trees according to the *edits*. Finally, the original sub-trees are replaced with the new ones, and the transformed source code c' is printed from the new AST (also called "pretty print").

We present all implemented rules in Figure 9 and Figure 10. Each rule is described with three elements: 1) the conditions of transferable sub-trees in AST, 2) the edits to rewrite the corresponding sub-trees, and 3) an illustrating example in pseudo-code. We use Conditions-Edits cards to ease communication.

4. Experiments Setting Details

4.1. Method Name Prediction

4.1.1. Architecture of Code2vec

The input of Code2vec is the Abstract Syntax Tree (AST) of programs. An AST is a tree-like representation of a code snippet. It does not contain every detail in the original source code (e.g., parentheses and comments) but rather represents the important syntactic structures that make up the code's functionality. The main idea of Code2vec is to uses a bag of "contexts" to represent a code snippet. Each "context" is a sequence of AST nodes connected by the "father and son" relationship extracted from the source code's AST. Attention mechanism [3] is adopted to aggregate the bag of "contexts" into a single continuous vector (called code vector). The word with an embedding vector nearest to the code vector is chosen as the prediction. After the training is finished, the Code2vec model can also be used as a pre-trained code representation model

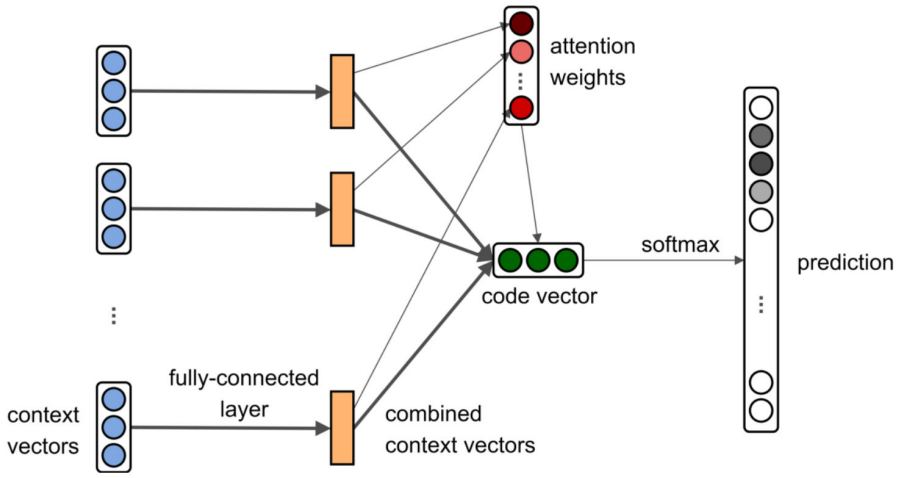

Figure 1: The architecture of Code2vec

for other big code tasks such as code search and code completion. The overall architecture of Code2vec can be seen in Figure 1. Its code is available at `https://github.com/tech-srl/code2vec`.

4.1.2. Datasets

We used three datasets from the original work of Code2vec: the dataset *small*, *med*, and *large*. The datasets can be downloaded from *https://s3.amazonaws.com/code2vec/data/java-small_data.tar.gz*, *https://s3.amazonaws.com/code2vec/data/java-med_data.tar.gz*, and *https://s3.amazonaws.com/code2vec/data/java-large_data.tar.gz*.

4.1.3. Experiment Settings

The hyperparameters used in this paper are the same as the original, which can be checked in Table 2. Adam optimizer is adopted.

4.1.4. Results of Rule Selection

Results of the test results on validation sets of all 18 rules are listed in Table 3 and Table 4. The selected rules for dataset *small* are 0,2,3,5,6,8,10,11,14,15, for

Table 2: Hyperameters used in the training of Code2vec.

MAX_CONTEXTS	200
WORD_VOCAB_SIZE	1301136
PATH_VOCAB_SIZE	911417
TARGET_VOCAB_SIZE	261245
BATCH_SIZE	1024
EMBEDDINGS_SIZE	128
MAX_TO_KEEP	10

Table 3: The rule selection results of rules 0-8 on Code2vec.

	F1 scores on the validation sets									
Datasets	ori	0	1	2	3	4	5	6	7	8
small	0.233	**0.261**	0.232	**0.243**	**0.241**	0.230	**0.235**	**0.235**	0.227	**0.239**
med	0.388	**0.411**	0.372	0.371	**0.397**	**0.393**	**0.389**	**0.390**	0.373	**0.407**
large	0.594	**0.607**	**0.598**	**0.601**	**0.604**	**0.597**	**0.595**	**0.595**	0.583	**0.599**

med are 0,3,4,5,6,8,10,11,12,14,17, and for large are 0,1,2,3,4,5,6,8,10,11,14,15,17.

4.2. Code Commenting

4.2.1. Architecture of DeepCom

DeepCom [4] treats code commenting as a variant of Natural Language Translation. It is based on Neural Machine Translation (NMT) [5], and focused on method-level commenting for Java. DeepCom consists of three parts: 1) encoder, 2) attention mechanism, and 3) decoder. The encoder uses LSTM to

Table 4: The rule selection results of rules 9-17 on Code2vec.

	F1 scores on the validation sets									
Datasets	ori	9	10	11	12	13	14	15	16	17
small	0.233	0.233	**0.244**	**0.234**	0.231	0.230	**0.234**	**0.235**	0.229	0.231
med	0.388	0.388	**0.394**	**0.390**	**0.397**	0.375	**0.394**	0.381	0.377	**0.391**
large	0.594	0.594	**0.601**	**0.601**	0.592	0.591	**0.596**	**0.595**	0.590	**0.598**

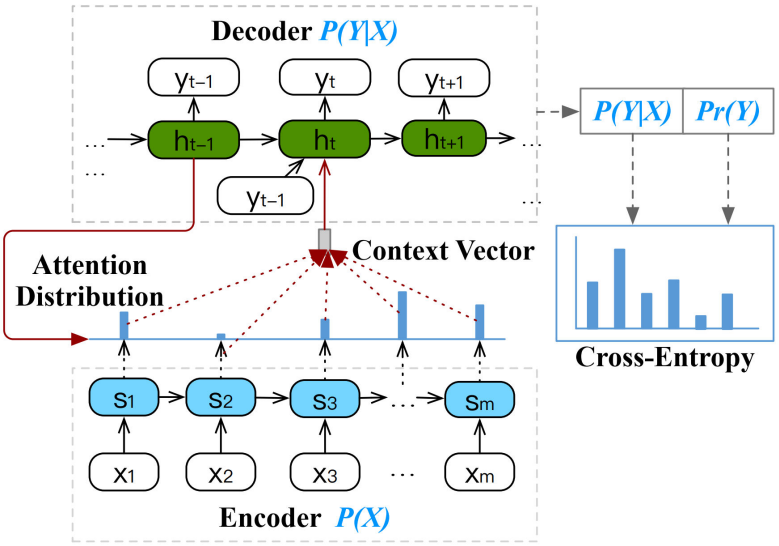

Figure 2: The architecture of DeepCom

encode the sequence of tokens converted from AST. Then, the attention mechanism compresses hidden states outputted by the LSTM into context vectors. In the end, context vectors are used to calculate the comments in decoding. The architecture of DeepCom is depicted in Figure 2. The input of DeepCom is not the plain text of the source code. Instead, it proposed a new structure-based traversal (SBT) method to traverse the ASTs of source codes (see Figure 3). Its code is available at `https://github.com/xing-hu/EMSE-DeepCom`.

4.2.2. Datasets

DeepCom used Eclipse's JDT compiler to parse the Java methods into ASTs and extract corresponding Javadoc comments, which are standard comments for Java methods. They get 69,708 <Java method, comment> pairs in the end. The dataset is available at `https://github.com/xing-hu/DeepCom`.

4.2.3. Experiment Settings

The SGD (with minibatch size 100 randomly chosen from training instances) is used to train the parameters. DeepCom uses two-layered LSTMs with 512

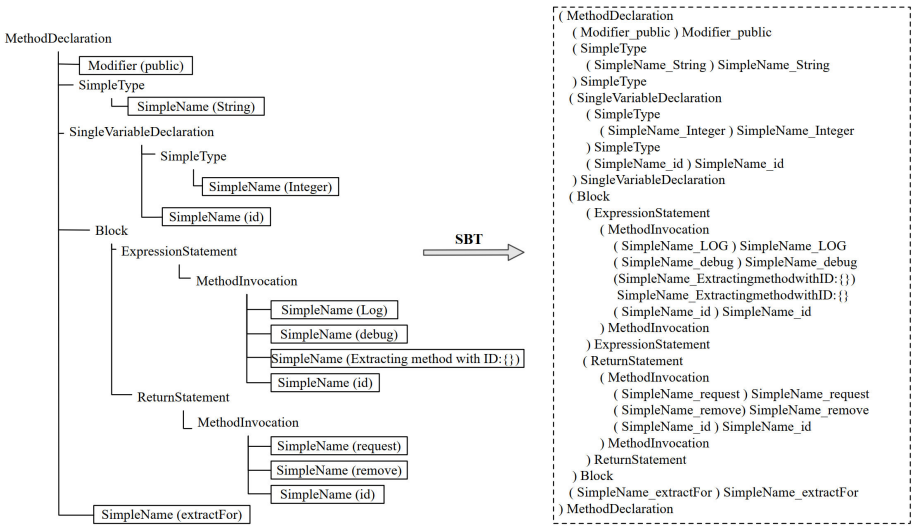

Figure 3: An example of how SBT works

dimensions of the hidden states and 512-dimensional word embeddings. The learning rate is set to 0.5 and is decayed using the rate of 0.99. Dropout is adopted with 0.5.

4.2.4. Results of Rule Selection

The test results on validation sets of all 18 rules for DeepCom are listed in Figure 4. The selected rules are: 0,1,2,3,5,8,10,11,14,16,17.

The test results on validation sets of all 18 rules for Hybrid-DeepCom are listed in Figure 5. The selected rules are: 0,1,2,3,7,8,9,10,11,13,14,15,16,17.

4.3. Code Clone Detection

4.3.1. Architecture of ASTNN

ASTNN [6] first parse a source code fragment into an AST and design a preorder traversal algorithm to split each AST into a sequence of statement trees (ST-trees, which are trees consisting of statement nodes as roots and corresponding AST nodes of the statements). All ST-trees are encoded by the Statement Encoder (see Figure 6) to vectors. It then uses the Bidirectional

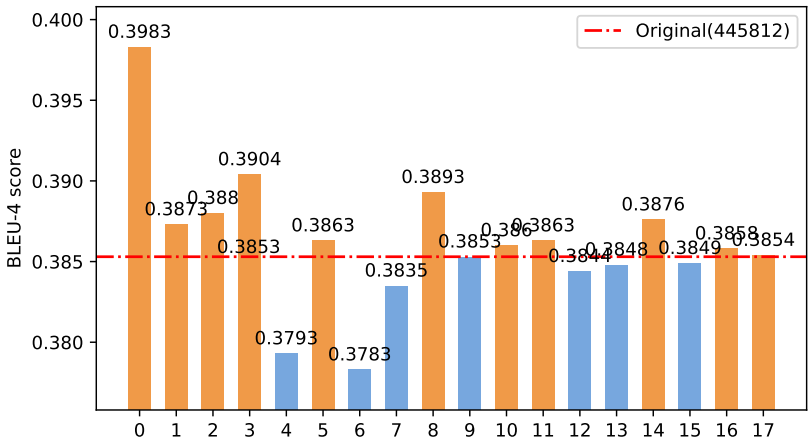

Figure 4: The rule selection results of rules 0-17 on DeepCom. The orange bars represent the selected rules.

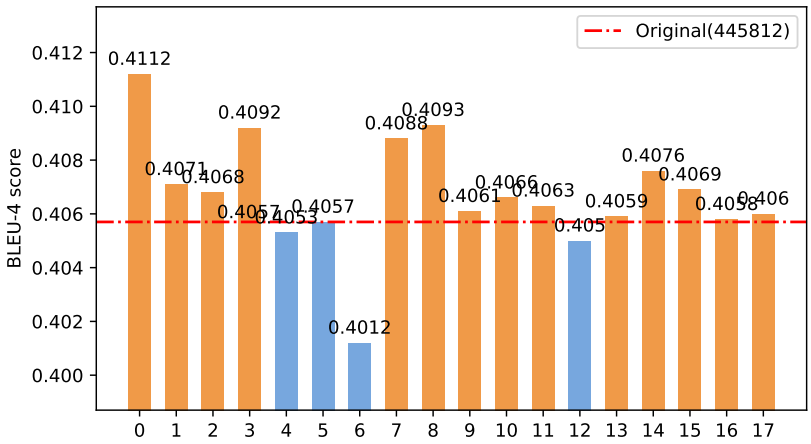

Figure 5: The rule selection results of rules 0-17 on Hybrid-DeepCom. The orange bars represent the selected rules.

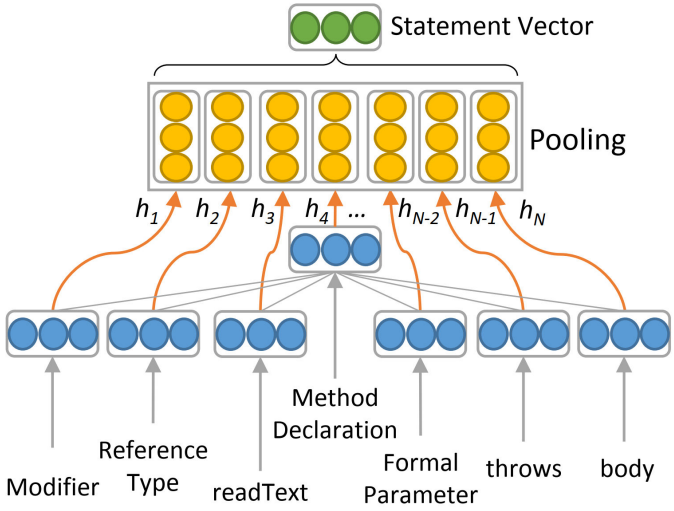

Figure 6: The statement encoder, where blue, orange and green circles represent the initial embeddings, hidden states and statement vector, respectively.

Gated Recurrent Unit (Bi-GRU) to model the naturalness of the statements. The hidden states of Bi-GRU are sampled into a single vector by pooling, which is the representation of the code fragment. The architecture of ASTNN is depicted in Figure 7.

4.3.2. Architecture of TBCCD

TBCCD [7] chose Abstract Syntax Tree as the input representation of code snippets. It utilizes tree-based convolution [8] as its basic module: a "continues binary tree" convolution unit that can handle children nodes of different numbers. Cosine similarity is used to measure the semantic distance between two code vectors in the end.

4.3.3. Datasets

The datasets of ASTNN and TBCCD are both chosen from BCB [9], but they adopted different strategies. Their datasets and codes are available at `https://github.com/yh1105/datasetforTBCCD` and `https://github.com/zhangj111/astnn`. The datasets built by us (*Kurts* and *Educoder*) will be made public soon.

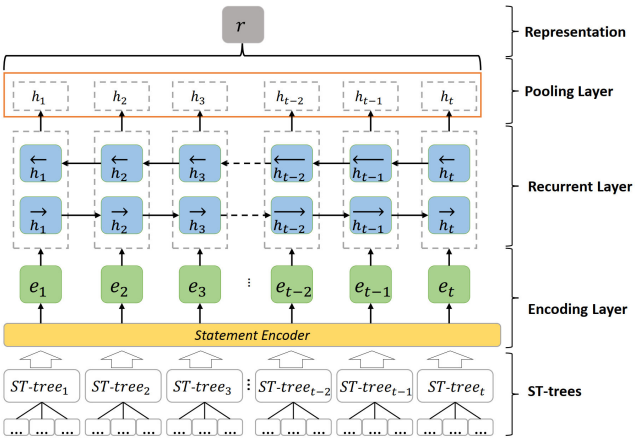

Figure 7: The architecture of AST-based Neural Network.

4.3.4. Experiment Settings

For TBCCD, the number of convolution kernels is 600, and the depth of the sliding window is 2; the dimension of the fully connected layer is 50; the number of epochs is ten, and the SGD optimizer is adopted; The threshold is decided on the validation set. For ASTNN, symbol embedding is trained using word2vec and Skip-gram with an embedding size 128; the hidden dimension of ST-tree encoder and bidirectional GRU is 100; the mini-batch size is set to 64, and the epoch number is 5; the threshold is 0.5, and AdaMax optimizer is adopted.

Different from the former two tasks where each data record only contains one code snippet, data records in Code Clone Detection have two code snippets. When augmentation, we only transform one side of a clone pair and keep the other side unchanged. That is, if the original data record is $< c_1, c_2, 1 >$, then the transformed one is $< c_1', c_2, 1 >$. We want to control the syntax variance to the minimum by doing so.

In addition, since we trained ASTNN and TBCCD on BCB and tested them on *Educoder*, we directly adopt all rules instead of rule selection because the result of rule selection on the validation set from BCB is only suitable for the test set form BCB.

5. Appendix G. Usage of SPAT

The command *"java -jar SPAT.jar [RuleId] [RootDir] [OutputDir] [PathofJre] & [PathofotherDependentJar]"* will transform all ".java" files in the root directory with the desired transformation rule in parallel. *[RootDir]* is a root directory path, in which each ".java" file is regarded as a code snippet. Each file should contain one Java class. For method-level code snippets, users need to warp each method with a "foo" class. All transformed files will have the same file names and relative paths as their source files. For example, a java file in *"rootDir/dir0/dir1/foo.java"* will have a transformed one in *"outputDir/dir0/dir1/foo.java"*. *[PathofJre]* is the path of *rt.jar* (usually placed in *".../jre1.x.x_xxx/lib/"*). This jar file is used to parse the basic dependencies of a java file, such as *String* type. Users can also include extra jar file paths for better parsing of code snippets.

References

[1] K. Wang and Z. Su, "Learning blended, precise semantic program embeddings," *CoRR*, vol. abs/1907.02136, 2019. [Online]. Available: http://arxiv.org/abs/1907.02136

[2] A. Jung, M. Fiore, E. Moggi, P. W. O'Hearn, J. G. Riecke, G. Rosolini, and I. Stark, "Domains and denotational semantics: History, accomplishments and open problems," *SCHOOL OF COMPUTER SCIENCE RESEARCH REPORTS-UNIVERSITY OF BIRMINGHAM CSR*, 1996.

[3] V. Mnih, N. Heess, A. Graves, and K. Kavukcuoglu, "Recurrent models of visual attention," in *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds., 2014, pp. 2204–2212. [Online]. Available: http://papers.nips.cc/paper/5542-recurrent-models-of-visual-attention

[4] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*, F. Khomh, C. K. Roy, and J. Siegmund, Eds. ACM, 2018, pp. 200–210. [Online]. Available: https://doi.org/10.1145/3196321.3196334

[5] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1409.0473

[6] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. IEEE Press, 2019, p. 783–794. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00086

[7] H. Yu, W. Lam, L. Chen, G. Li, T. Xie, and Q. Wang, "Neural detection of semantic code clones via tree-based convolution," in *Proceedings of the 27th International Conference on Program Comprehension*, ser. ICPC '19. IEEE Press, 2019, p. 70–80. [Online]. Available: https://doi.org/10.1109/ICPC.2019.00021

[8] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, ser. AAAI'16. AAAI Press, 2016, p. 1287–1293.

[9] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *2014 IEEE International Conference on Software Maintenance and Evolution*, Sep. 2014, pp. 476–480.

Original Prediction: **contain**

```
boolean f(Object target) {
    for (Object elem : this.elements) {
        if (elem.equals(target)) {
            return true;
        }
    }
    return false;
}
```

Transformed by ***rule 3:*** **isApplicable**

```
boolean f(Object target) {
    for (Object elem : this.elements) {
        if (!(elem.equals(target)))
            ;
        else {
            return true;
        }
    }
    return false;
}
```

Original: **sort**

```
void f(int[] array) {
    boolean swapped = true;
    for (int i = 0; i < array.length && swapped; i++) {
        swapped = false;
        for (int j = 0; j < array.length - 1 - i; j++) {
            if (array[j] > array[j+1]) {
                int temp = array[j];
                array[j] = array[j+1];
                array[j+1]= temp;
                swapped = true;
            }
        }
    }
}
```

Transformed by ***rule 0, rule 1:*** **merge**

```
void f(int[] O4Zxt74v) {
    boolean wPm6DDIi = true;
    int h5vzKqDn = 0;
    while (h5vzKqDn < O4Zxt74v.length && wPm6DDIi) {
        wPm6DDIi = false;
        for (int oy35WL7G = 0;
             oy35WL7G < O4Zxt74v.length - 1 - h5vzKqDn;
             oy35WL7G++) {
            if (O4Zxt74v[oy35WL7G] > O4Zxt74v[oy35WL7G + 1]) {
                int WuL3aj5H = O4Zxt74v[oy35WL7G];
                O4Zxt74v[oy35WL7G] = O4Zxt74v[oy35WL7G + 1];
                O4Zxt74v[oy35WL7G + 1] = WuL3aj5H;
                wPm6DDIi = true;
            }
        }
        h5vzKqDn++;
    }
}
```

Original: **count**

```
int f(String target, ArrayList<String> array) {
    int count = 0;
    for (String str : array) {
        if (target.equals(str)) {
            count++;
        }
    }
    return count;
}
```

Transformed by ***rule 0:*** **nativeFindAll**

```
int f(String vPbjselX, ArrayList<String> MbEdUkXg) {
    int OcNt2Xtt = 0;
    for (String xGud7LeM : MbEdUkXg) {
        if (vPbjselX.equals(xGud7LeM)) {
            OcNt2Xtt++;
        }
    }
    return OcNt2Xtt;
}
```

Original: **get**

```
Object f(int target) {
    for (Object elem : this.elements) {
        if (elem.hashCode().equals(target)) {
            return elem;
        }
    }
    return this.defaultValue;
}
```

Transformed by ***rule 0, rule 3:*** **getDefaultValue**

```
Object f(int OAZtgHmF) {
    for (Object Yp8yl5v2 : this.elements) {
        if (!(Yp8yl5v2.hashCode().equals(OAZtgHmF)))
            ;
        else {
            return Yp8yl5v2;
        }
    }
    return this.defaultValue;
}
```

Original: **reverseArray**

```
String[] f(final String[] array) {
    final String[] newArray =
                        new String[array.length];
    for (int index = 0; index < array.length; index++){
        newArray[array.length - index - 1] =
                                    array[index];
    }
    return newArray;
}
```

Transformed by ***rule 0, rule 1:*** **parse**

```
String[] f(final String[] zKeN64Hq) {
    final String[] o9PWyOHR =
                        new String[zKeN64Hq.length];
    int NnK1Xzki = 0;
    while (NnK1Xzki < zKeN64Hq.length) {
        o9PWyOHR[zKeN64Hq.length - NnK1Xzki - 1] =
                                    zKeN64Hq[NnK1Xzki];
        NnK1Xzki++;
    }
    return o9PWyOHR;
}
```

Figure 8: Prediction failures of Code2vec when code forms are slightly changed. Code snippets on the left are the original accurate predictions made by Code2vec, while the ones on the right sides are transformed by SPAT with different rules. *Rule 0* is *LocalVarRenaming*, *Rule 1* is *For2While*, and *Rule 3* is *ReverseIfElse*.

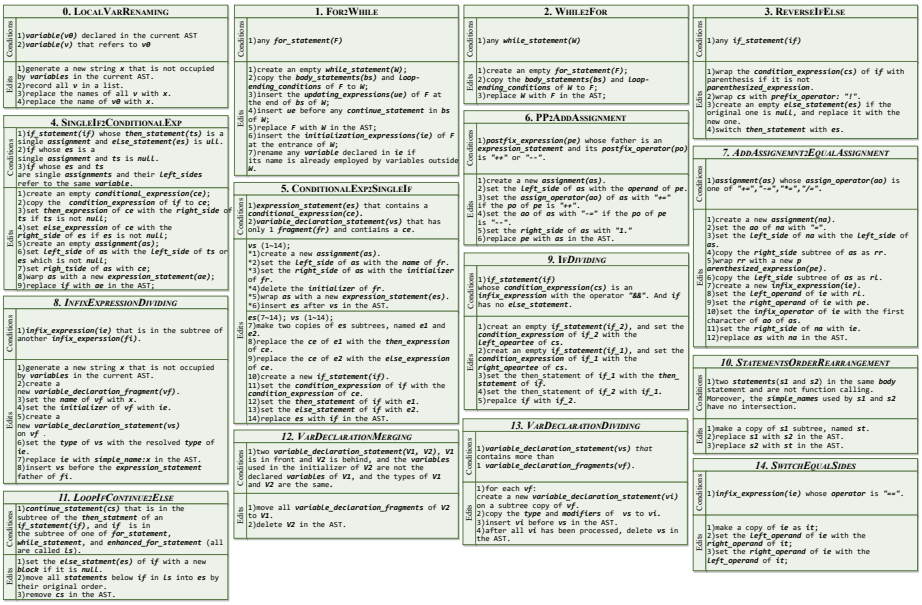

Figure 9: CE tables of all rules. All bold-italic phrases represent AST sub-trees. We abbreviate the names of AST sub-trees to 1 or 2 letters after their first occurrences. The conditions are grouped with "or" logical relation, and the editing operations are executed by the sequence order.

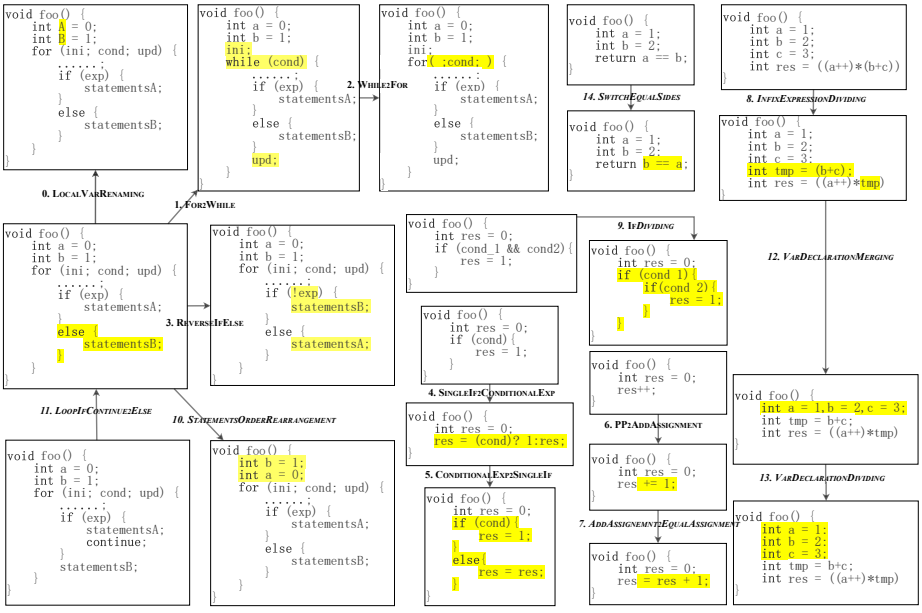

Figure 10: Pseudo-code examples of all rules. Arrows indicate program transformations. Code lines colored yellow highlight the difference made by the transformation.