Test Coverage Metric for Two-staged Language with Abstract Interpretation

Taeksu Kim, Chunwoo Lee, Kiljoo Lee, Soohyun Baik, Chisu Wu
Software Engineering Laboratory
Seoul National University
Seoul, Korea
{*dolicoli,oniguni,laazycat,shbaik82,wuchisu*}*@selab.snu.ac.kr*

Kwangkeun Yi
Programming Research Laboratory
Seoul National University
Seoul, Korea
kwang@ropas.snu.ac.kr

***Abstract*—As a program written in multi-staged language can generate and execute code fragments in excution time, it is hard to predict how many code fragments will be generated in execution time. Therefore, current test coverages are not likely to give right answers when they are apply to a program written in multi-staged language because the program size could not be estimated easily.**

In this paper, we present static analysis which detects code fragments generated in execution time using abstract interpretation and prove the correctness of analyzer. Moreover we propose new test coverage for multi-staged language using the result of analysis.

***Keywords*-software testing; test coverage; multi-staged language; abstract interpretation;**

I. Introduction

Multi-staged language is a programming language which can generate and execute new program codes in execution time[1]. Recently, multi-staged language is used widely because it can make software development faster and easier. Many Web programming languages and script languages(e.g. Ruby, Python, PHP, Lisp, JavaScript, etc.) adopt multi-staged features in their specification so that they support rapid development.

Test coverage is a measure to evaluate the quality of a test suite. It is defined as a ratio of *size of executed programs with test suite* to *size of program that should be tested*. For example, statement coverage uses SLOC as the criteria for measuring program size and decision coverage uses the number of branches[2].

It seems unsuitable for applying current test coverages to multi-staged language. Because it is hard to estimate what code fragments would be generated and executed in multi-staged language. Figure 1(a) shows an example source code written in JavaScript. A test suite $\{(0,0),(0,1)\}$ can acquire 100% decision coverage. However the correct decision coverage value should be 67% because the example code in figure 1(a) will run actually like a code in figure 1(b).

It is needed to analyze what code fragments would be generated in execution time to guarantee the correctness of a test coverage. In this paper, we designed static analysis which detects code fragments generated in execution time using abstract interpretation[3] and proved the correctness of analyzer. We also proposed new test coverage for multi-staged language using the result.

```
function foo(a, b) {
  var x = "if ("+ a +" > 0) return 3;" +
          "else return 4;";
  if (b > 0) return eval(x);
  else return 0;
}
```

(a) Original code

```
function foo(a, b) {
  if (b > 0 && a > 0) return 3;
  else if (b > 0) return 4;
  else return 0;
}
```

(b) Real runnig code

Figure 1. Example JavaScript code

The rest of this paper is organized as follows. The next chapter defines syntax and semantics of simple two-staged language. Chapter III presents an analyzer using abstract interpretation and proof of the correctness of the analyzer. In chapter IV, we propose new test coverage using the analyzer. Related work are introduced in chapter V. Finally, coclusions and the idea for the future work are presented.

II. Two-staged Language

A. Syntax and Assumptions

Figure 2 shows a syntax definition of a simplified two-staged language. In the language, a program consists of expressions. Functions and code fragments are the first-class objects. This language uses quasi-quotations[4] to generate and execute code fragments in the execution time. Boxing(‘) and unboxing(,) operators are labeled uniquely by *Label* and *Alphabet*.

To decrease the complexity of the analysis we make some assumptions on the language as follows.

- The language only supports two-stage processing.
- Expression $\mathbf{run}\ e$ never generate code fragments.
- All *Labels* and *Alphabets* are unique values in the program text.

From now on, we will only concern this language. Multi-staged languages over two-stage can be treated similarly.

$$
\begin{array}{rcl}
\mathcal{E} & \in & Stage \to Exp \to 2^{Env} \to 2^{Val \times Env} \\
 & & \text{is the least fixed point of} \\
\mathcal{F} & \in & (Stage \to Exp \to 2^{Env} \to 2^{Val \times Env}) \to \\
 & & (Stage \to Exp \to 2^{Env} \to 2^{Val \times Env}) \\
Stage & = & \{0, 1\}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{F}\ \mathcal{E}\ 0\ [\![c]\!]\ \Sigma & := & \{\langle c, \sigma\rangle : \sigma \in \Sigma\} \\
\mathcal{F}\ \mathcal{E}\ 0\ [\![x]\!]\ \Sigma & := & \{\langle \sigma(x), \sigma\rangle : \sigma \in \Sigma\} \\
\mathcal{F}\ \mathcal{E}\ 0\ [\![\mathbf{let}\ x\ e_1\ e_2]\!]\ \Sigma & := & \mathbf{let}\ T = \mathcal{E}\ 0\ [\![e_1]\!]\ \Sigma\ \mathbf{in} \\
 & & \bigcup_{\langle v,\sigma\rangle \in T} \mathcal{E}\ 0\ [\![e_2]\!]\ \{\sigma[x \mapsto v]\} \\
\mathcal{F}\ \mathcal{E}\ 0\ [\![\mathbf{if}\ e_1\ e_2\ e_3]\!]\ \Sigma & := & \mathcal{E}\ 0\ [\![e_2]\!]\ (\neg\mathcal{B}\ [\![e_1]\!]\ \Sigma) \\
 & & \cup\ \mathcal{E}\ 0\ [\![e_3]\!]\ (\mathcal{B}\ [\![e_1]\!]\ \Sigma)
\end{array}
$$

where for $\Sigma \in 2^{Env}$ and $e \in Exp$,

$$
\begin{array}{rcl}
\mathcal{B}\ [\![e]\!]\ \Sigma & \in & 2^{Env} = \{\sigma \in \Sigma : e \textit{ becomes } 0 \textit{ under } \sigma\} \\
\neg\mathcal{B}\ [\![e]\!]\ \Sigma & \in & 2^{Env} = \Sigma - \mathcal{B}\ [\![e]\!]\ \Sigma \\
\mathcal{F}\ \mathcal{E}\ 0\ [\![f\lambda x.e]\!]\ \Sigma & := & \{\langle\langle f\lambda x.e, \sigma\rangle, \sigma\rangle : \sigma \in \Sigma\} \\
\mathcal{F}\ \mathcal{E}\ 0\ [\![e_1\ \ e_2]\!]\ \Sigma & = & \\
 & & \mathbf{let}\ T_1 = \mathcal{E}\ 0\ [\![e_1]\!]\ \Sigma\ \mathbf{in} \\
 & & \mathbf{let}\ T_2 = \bigcup_{\langle f\lambda x.b,\sigma\rangle\sigma_1 \in T_1} \mathcal{E}\ 0\ [\![e_2]\!]\ \{\sigma_2\}\ \mathbf{in} \\
 & & \bigcup \mathcal{E}\ 0\ [\![b]\!]\ \{\sigma_2[x \mapsto v][f \mapsto \langle f\lambda x.b, \sigma\rangle]\} \\
 & & \text{where } \langle\langle f\lambda x.b, \sigma\rangle, \sigma_1\rangle \in T_1, \langle v, \sigma_2\rangle \in T_2 \\
\mathcal{F}\ \mathcal{E}\ 0\ [\![\text{`}_l e]\!]\ \Sigma & := & \mathbf{let}\ T = \mathcal{E}\ 1\ [\![e]\!]\ \Sigma\ \mathbf{in} \\
 & & \{\langle [e'], \sigma\rangle : \langle [e'], \sigma\rangle \in T\} \\
\mathcal{F}\ \mathcal{E}\ 0\ [\![\mathbf{run}\ e]\!]\ \Sigma & := & \mathbf{let}\ T = \mathcal{E}\ 0\ [\![e]\!]\ \Sigma\ \mathbf{in} \\
 & & \bigcup_{\langle [e'],\sigma\rangle \in T} \mathcal{E}\ 0\ [\![e']\!]\ \{\sigma\}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{F}\ \mathcal{E}\ 1\ [\![c]\!]\ \Sigma & := & \{\langle [c], \sigma\rangle : \sigma \in \Sigma\} \\
\mathcal{F}\ \mathcal{E}\ 1\ [\![x]\!]\ \Sigma & := & \{\langle [x], \sigma\rangle : \sigma \in \Sigma\} \\
\mathcal{F}\ \mathcal{E}\ 1\ [\![\mathbf{let}\ x\ e_1\ e_2]\!]\ \Sigma & := & \mathbf{let}\ T_1 = \mathcal{E}\ 1\ [\![e_1]\!]\ \Sigma\ \mathbf{in} \\
 & & \mathbf{let}\ T_2 = \bigcup_{\langle [e'_1],\sigma_1\rangle \in T_1} \mathcal{E}\ 1\ [\![e_2]\!]\ \{\sigma_1\}\ \mathbf{in} \\
 & & \{\langle [\mathbf{let}\ x\ e'_1\ e'_2], \sigma_2\rangle : \\
 & & \quad \langle [e'_1], \sigma_1\rangle \in T_1, \langle [e'_2], \sigma_2\rangle \in T_2\} \\
\mathcal{F}\ \mathcal{E}\ 1\ [\![\mathbf{if}\ e_1\ e_2\ e_3]\!]\ \Sigma & := & \mathbf{let}\ T_1 = \mathcal{E}\ 1\ [\![e_1]\!]\ \Sigma\ \mathbf{in} \\
 & & \mathbf{let}\ T_2 = \bigcup_{\langle [e'_1],\sigma_1\rangle \in T_1} \mathcal{E}\ 1\ [\![e_2]\!]\ \{\sigma_1\}\ \mathbf{in} \\
 & & \mathbf{let}\ T_3 = \bigcup_{\langle [e'_2],\sigma_2\rangle \in T_2} \mathcal{E}\ 1\ [\![e_3]\!]\ \{\sigma_2\}\ \mathbf{in} \\
 & & \{\langle [\mathbf{if}\ e'_1\ e'_2\ e'_3], \sigma_3\rangle : \\
 & & \quad \langle [e'_1], \sigma_1\rangle \in T_1, \langle [e'_2], \sigma_2\rangle \in T_2, \\
 & & \quad \langle [e'_3], \sigma_3\rangle \in T_3\} \\
\mathcal{F}\ \mathcal{E}\ 1\ [\![f\lambda x.e]\!]\ \Sigma & := & \mathbf{let}\ T = \mathcal{E}\ 1\ [\![e]\!]\ \Sigma\ \mathbf{in} \\
 & & \{\langle [f\lambda x.e'], \sigma\rangle : \langle [e'], \sigma\rangle \in T\} \\
\mathcal{F}\ \mathcal{E}\ 1\ [\![e_1\ \ e_2]\!]\ \Sigma & := & \mathbf{let}\ T_1 = \mathcal{E}\ 1\ [\![e_1]\!]\ \Sigma\ \mathbf{in} \\
 & & \mathbf{let}\ T_2 = \bigcup_{\langle [e'_1],\sigma_1\rangle \in T_1} \mathcal{E}\ 1\ [\![e_2]\!]\ \{\sigma_1\}\ \mathbf{in} \\
 & & \{\langle [e'_1\ \ e'_2], \sigma_2\rangle : \\
 & & \quad \langle [e'_1], \sigma_1\rangle \in T_1, \langle [e'_2], \sigma_2\rangle \in T_2\} \\
\mathcal{F}\ \mathcal{E}\ 1\ [\![_\alpha e]\!]\ \Sigma & := & \mathcal{E}\ 0\ [\![e]\!]\ \Sigma \\
\mathcal{F}\ \mathcal{E}\ 1\ [\![\mathbf{run}\ e]\!]\ \Sigma & = & \mathbf{let}\ T := \mathcal{E}\ 1\ [\![e]\!]\ \Sigma\ \mathbf{in} \\
 & & \{\langle [\mathbf{run}\ e'], \sigma\rangle : \langle [e'], \sigma\rangle \in T\}
\end{array}
$$

Figure 3. A collecting semantics of the target language

$$
\begin{array}{rcl}
e & \to & c \\
 & | & x \\
 & | & \mathbf{if}\ e\ e\ e \\
 & | & \mathbf{let}\ x\ e\ e \\
 & | & f\lambda x.e \\
 & | & e\ \ e \\
 & | & \text{`}_l e \\
 & | & {}_\alpha e \\
 & | & \mathbf{run}\ e \\
Val & = & Code + Closure + Constant \\
Env & = & Var \xrightarrow{fin} Val \\
x & \in & Var \\
l & \in & Label \\
\alpha & \in & Alphabet
\end{array}
$$

Figure 2. Syntax of simplified two-staged language

B. Collecting Semantics

We should make concrete domains to be CPOs(Complete Partial Ordered Sets) for an abstract interpretation. Let us make each concrete domain to be a power set of them to make concrete domains to be CPOs.

Figure 3 shows a collecting semantics of the target language. A semantic function $\mathcal{E}$ is defined as the least fixed point of a function $\mathcal{F}$. Boxing operator makes a subexpression as a code fragment and unboxing operator evaluates the value of subexpression in one-stage processing. Run operator evaluates the value of a code fragment in zero-stage processing. The semantics of other expressions are same as typical single-staged languages' semantics.

Lemma 1: $2^{Env}, 2^{Val}, 2^{Val \times Env}$ are CPOs.

Proof: As 2^{Env} is a power set, it is naturally partially ordered set with relation $\subseteq$. Besides, $\emptyset$ is a bottom and set of all environments is a least upper bound of all chains in 2^{Env}. Therefore 2^{Env} is a CPO.

Similarly, 2^{Val} and $2^{Val \times Env}$ are CPOs. ■

III. Static Analysis of the Target Language

A. Abstract Interpretation

Abstract interpretation[3] is a theory for static analysis of software systems that gains information about its semantics without performing all the calculations. We can derive the result which includes all possible cases with the abstract interpretation and prove its correctness.

Therefore, an abstract interpretation is a proper framework for designing an analyzer which can detect code fragments of multi-staged language generated in execution time. In this paper, we adopt the abstract interpretation for the analysis and proof of its correctness.

B. Grammar

Our analysis aims to know what code fragments will be generated and executed in execution time. So, we should abstract values which are required for generating code fragments. It is needed to abstract closures and code fragments of a program because the target language treats a function as a first-class object.

As it is enough to know the function declaration of a closure, we can abstract the closure easily. On the other hand, we should know all possible code values to abstract a code fragment because the code fragments can be substitued by those code values in runtime. Therefore we need a new data type to abstract a code fragment which can indicates following informations.

- Subexpressions that it contains.
- Possible code fragments that the subexpressions can be substituted in one-stage processing.

Let us define a new data type, *Grammar* as a production rule of formal grammar[5] to abstract the code fragment. Rules are as follows.

- (Terminal)

$$Z \rightarrow \triangle \mid label \mid Z[Nonterminal^*] \mid Z|Z$$

  - Label l: a code fragment whose label is l.
  - $\triangle$: a code fragment whose label is unknown.
  - $Z[Nonterminal^*]$: a code fragment which has unboxing subexpressions represented with nonterminal alphabets. The label of the code fragment is detemined by Z.
  - $Z|Z$: a sequence of the terminals.
- (Nonterminal)

$$N \rightarrow \mathfrak{s} \mid alphabet$$

  - $\mathfrak{s}$: a start symbol
  - Alphabet a: an unboxing operation whose alphabet is a.
- (Start Symbol)

$$\Sigma \rightarrow \mathfrak{s}$$

Terminal and non-terminal represent labels of boxing expressions and alphabets of unboxing expressions respectively. For example, the code fragment in figure 4 can be represented as a grammar as follows.

$$\mathfrak{s} \rightarrow 5[a, b], a \rightarrow 1|2, b \rightarrow 3|4$$

It means that an expression labeled by label 5-$\text{`}^5(_{,a}x\ \ _{,b}y)$ contains two unbox subexpressions labeled by $a$ and $b$. Moreover, the unbox subexpression labeled by $a$ can be substituted by an expression labeled by 1-$\text{`}^1 f\lambda x.x$ or 2-$\text{`}^2 g\lambda y.y$ and the unbox subexpression labeled by $b$ can be substituted by an expression labeled by 3-$\text{`}^3 100$ or 4-$\text{`}^4 101$. Therefore the code fragment can be one of the following code fragment in execution time.

- $\text{`}^5(f\lambda x.x\ \ 100)$
- $\text{`}^5(f\lambda x.x\ \ 101)$
- $\text{`}^5(g\lambda y.y\ \ 100)$
- $\text{`}^5(g\lambda y.y\ \ 101)$

```
let x = (if 0 then `1fλx.x else `2gλy.y) in
let y = (if 1 then `3100 else `4101) in
`5(,a x  ,b y)
```

Figure 4. Example code

One code fragment corresponds with one grammar. On the other side, one grammar can stand for a number of code fragments. So we can define following functions naturally.

Definition 1:

$$\begin{aligned} \psi &\in Grammar \rightarrow 2^{Exp} \\ \xi &\in Code \rightarrow Grammar \end{aligned}$$

- $\psi(g)$: For an arbitrary grammar g, a set of expressions that can be represented by g.
- $\xi(c)$: For an arbitrary code fragment c, a grammar that represents c.

C. Abstract Semantics and Operations

We should abstract concrete domains to abstract domains for abstract interpretation. Figure 5 shows abstract domains. $\hat{Val}$ and $\hat{Env}$ are abstract domains of 2^{Val} and 2^{Env} respectively. When a value set V is a set of codes, it can be abstracted as a grammar and if V is a set of closure, it can be abstracted as a set of expressions. A detailed abstraction functions are shown in figure 6.

$$\begin{aligned} \hat{Val} &= Grammar + 2^{Exp} + \{\cdot\} + \hat{\perp}_v + \hat{\top}_v + \hat{\top}_g \\ \hat{Env} &= Var \xrightarrow{fin} \hat{Val} \end{aligned}$$

Figure 5. Abstract domains

Lemma 2: $\hat{Env}, \hat{Val}, \hat{Val} \times \hat{Env}$ are CPOs.

Proof: Let us define an order between abstract values as follows.

- (Grammar Order)

$$g \sqsubseteq_g g_2 \text{ iff } \psi(g_1) \subseteq \psi(g_2)$$

$$2^{Val} \underset{\gamma_v}{\overset{\alpha_v}{\rightleftharpoons}} \hat{Val}$$

$$\alpha_v V = \begin{cases} \hat{\bot}_v & \text{if } V = \emptyset \\ \{\cdot\} & \text{if } V \in 2^{Constant} \\ \{f \mid \langle f, \sigma \rangle \in V\} & \text{if } V \in 2^{Closure} \\ \bigsqcup_{e \in V} \xi(e) & \text{if } V \in 2^{Code} \text{ and } V = \{e_1, e_2, \cdots\} \\ \hat{\top}_v & \text{otherwise} \end{cases}$$

$$2^{Env} \underset{\gamma_e}{\overset{\alpha_e}{\rightleftharpoons}} \hat{Env}$$

$$\alpha_e\ \Sigma \quad = \quad \lambda x.\alpha_v \bigcup_{\sigma \in \Sigma} \{\sigma x\}$$

$$2^{Val \times Env} \underset{\gamma_{v \times e}}{\overset{\alpha_{v \times e}}{\rightleftharpoons}} \hat{Val} \times \hat{Env}$$

$$\alpha_{v \times e}\ T = \begin{cases} \langle \hat{\bot}_v, \hat{\bot}_e \rangle & \text{if } T = \emptyset \\ \langle \bigsqcup_{\langle v, \sigma \rangle \in T} \alpha_v\{v\}, \bigsqcup_{\langle v, \sigma \rangle \in T} \alpha_e\{\sigma\} \rangle & \text{otherwise} \end{cases}$$

Figure 6. Abstract functions

- (Order)

$$\hat{\bot}_v \quad \sqsubseteq_v \quad \hat{v} \quad \text{for arbitrary } \hat{v} \in \hat{Val}$$
$$\hat{v} \quad \sqsubseteq_v \quad \hat{\top}_v \quad \text{for arbitrary } \hat{v} \in \hat{Val}$$

$$\hat{v_1} \sqsubseteq_v \hat{v_2} \iff \begin{cases} \hat{v_1} \subseteq \hat{v_2} & \text{if } \hat{v_1}, \hat{v_2} \in 2^{Exp} \\ \hat{v_1} \sqsubseteq_g \hat{v_2} & \text{if } \hat{v_1}, \hat{v_2} \in Grammar \\ \hat{v_1} = \hat{v_2} & \text{if } \hat{v_1} = \{\cdot\}, \hat{v_2} = \{\cdot\} \end{cases}$$

Then $\hat{Val}$ is a partially ordered set for $\sqsubseteq_v$ and $\hat{Val}$ has a bottom $\hat{\bot}_v$. Besides, all chains in $\hat{Val}$ has a least upper bound $\hat{\top}_v$. Therefore $\hat{Val}$ is a CPO.

Next, let us define an order of $\hat{Env}$ as follows.

$$\hat{\sigma_1} \sqsubseteq_e \hat{\sigma_2} \text{ iff } \text{dom}(\hat{\sigma_1}) \subseteq \text{dom}(\hat{\sigma_1}) \wedge$$
$$\forall x \in \text{dom}(\hat{\sigma_1}).\hat{\sigma_1}(x) \sqsubseteq_v \hat{\sigma_2}(x)$$

Then $\hat{Env}$ is a partially ordered set for $\sqsubseteq_e$ and $\hat{Val}$ has a bottom $[\ ]$. Besides, all chains in $\hat{Env}$ has a least upper bound (let a set of all variables as $X = \{x_1, x_2, \cdots\}$. Then $[x_1 \mapsto \hat{\top}_v, x_2 \mapsto \hat{\top}_v, \cdots]$ is a least upper bound). Therefore $\hat{Env}$ is a CPO.

As $\hat{Val} \times \hat{Env}$ is a production set of CPOs, it is a CPO. ■

We should define some operations for abstract domains.

- (Slot Append) For a grammar g and an alphabet α,

$$\alpha \oplus g$$

is a grammar that represents a code fragment whose label is unknown and which has an unboxing expression labeled with α. The unboxing expression can be substitued with code framgments represented by g.

- (Code Append) For a grammar g and a label l,

$$l \otimes g$$

is a grammar that represents a code fragment whose label is l and which has unboxing expressions represented by g.

- (Grammar Join) Grammar g_1, g_2,

$$g_1 \odot g_2$$

is a grammar such that $\psi(g_1 \odot g_2) = \psi(g_1) \cup \psi(g_2)$.

- (Value Join)

$$\hat{v_1} \sqcup \hat{v_2} = \begin{cases} \hat{v_1} & \text{if } \hat{v_2} = \hat{\bot}_v \\ \hat{v_2} & \text{if } \hat{v_1} = \hat{\bot}_v \\ \{\cdot\} & \text{if } \hat{v_1} = \{\cdot\}, \hat{v_2} = \{\cdot\} \\ \hat{v_1} \cup \hat{v_2} & \text{if } \hat{v_1}, \hat{v_2} \in 2^{Exp} \\ \hat{v_1} \odot \hat{v_2} & \text{if } \hat{v_1}, \hat{v_2} \in Grammar \\ \hat{\top}_v & \text{otherwise} \end{cases}$$

- (Environment Join)

$$\hat{\sigma_1} \hat{+} \hat{\sigma_2} \in \hat{Env}$$

$$(\hat{\sigma_1} \hat{+} \hat{\sigma_2})x = \begin{cases} \hat{\sigma_1} x & \\ \quad \text{if } x \in \text{dom } (\hat{\sigma_1}) \text{ and } x \notin \text{dom } (\hat{\sigma_2}) & \\ \hat{\sigma_2} x & \\ \quad \text{if } x \notin \text{dom } (\hat{\sigma_1}) \text{ and } x \in \text{dom } (\hat{\sigma_2}) & \\ \hat{\sigma_1} x \ \sqcup \ \hat{\sigma_2} x \quad \text{otherwise} & \end{cases}$$

For the target language, we can define abstract semantics as figure 7. A semantic function $\hat{\mathcal{E}}$ is defined as the least fixed point of a function $\hat{\mathcal{F}}$.

D. Abstractions

Next step of the abstract interpretation is defining abstract functions between concrete domains and abstract domains. We define abstract functions as figure 6. Then we can easily show that concrete domain is Galois connected[6] with abstract domain.

Lemma 3: 2^{Val} and $\hat{Val}$ are Galois connected with α_v and γ_v.

Proof: To prove the lemma, we should show that

$$V \in 2^{Val}, \hat{v} \in \hat{Val}, \alpha_v V \sqsubseteq_v \hat{v} \iff V \subseteq \gamma_v \hat{v}$$

We present the proof when $\hat{v} \in Grammar$ and $V \in 2^{Code}$. Other cases can be easily shown similarly.

1) Assume that $V \in 2^{Val}, \hat{v} \in \hat{Val}$ and $\alpha_2 V \sqsubseteq_v \hat{v}$.
If $V \in 2^{Code}$, $\hat{v} \sqsupseteq_v \alpha_v V = \xi(e_1) \sqcup \xi(e_2) \sqcup \cdots$ where $V = \{e_1, e_2, \cdots\}$. Therefore $\hat{v} = \hat{\top}_v$ or $\hat{v} = g$ such that $g \sqsupseteq_v \xi(e_1) \sqcup \xi(e_2) \sqcup \cdots$.
When $\hat{v} = \hat{\top}_v$, $\gamma_v \hat{v}$ = Set of all values. Therefore $V \subseteq \gamma_v \hat{v}$.
When $\hat{v} = g$, $\psi(g) \supseteq \{e_1, e_2, \cdots\}$ by definition. Therefore $\gamma_v \hat{v} = \psi(g) \supseteq \{e_1, e_2, \cdots\} = V$. Therefore, $V \subseteq \gamma_v \hat{v}$. Thus,

$$\alpha_v V \sqsubseteq_v \hat{v} \implies V \subseteq \gamma_v \hat{v}$$

2) Assume that $V \in 2^{Val}, \hat{v} \in \hat{Val}$ such that $V \subseteq \gamma_2 \hat{v}$.
If $\hat{v} \in Grammar$, $V \subseteq \gamma_v \hat{v} = \psi(\hat{v})$. Therefore $V = \emptyset$ or $V \in 2^{Code}$ and $\forall e \in V, \exists e' \in \psi(\hat{v})$ such that $e = e'$.

$$
\begin{array}{rcl}
\hat{\mathcal{E}} & \in & Stage \to Exp \to \hat{Env} \to \hat{Val} \times \hat{Env} \\
 & & \text{is the least fixed point of} \\
\hat{\mathcal{F}} & \in & (Stage \to Exp \to \hat{Env} \to \hat{Val} \times \hat{Env}) \to \\
 & & (Stage \to Exp \to \hat{Env} \to \hat{Val} \times \hat{Env})
\end{array}
$$

$$
\begin{array}{rcl}
\hat{\mathcal{F}}\ \hat{\mathcal{E}}\ 0\ [\![c]\!]\ \hat{\sigma} & = & \langle \{\cdot\}, \hat{\sigma} \rangle \\
\hat{\mathcal{F}}\ \hat{\mathcal{E}}\ 0\ [\![x]\!]\ \hat{\sigma} & = & \langle \hat{\sigma} x, \hat{\sigma} \rangle \\
\hat{\mathcal{F}}\ \hat{\mathcal{E}}\ 0\ [\![\textbf{let } x\ e_1\ e_2]\!]\ \hat{\sigma} & = & \textbf{let } \langle \hat{v}', \hat{\sigma}' \rangle = \hat{\mathcal{E}}\ 0\ [\![e_1]\!]\ \hat{\sigma} \textbf{ in} \\
 & & \hat{\mathcal{E}}\ 0\ [\![e_2]\!]\ \hat{\sigma}'[x \mapsto \hat{v}'] \\
\hat{\mathcal{F}}\ \hat{\mathcal{E}}\ 0\ [\![\textbf{if } e_1\ e_2\ e_3]\!]\ \hat{\sigma} & = & \textbf{let } \langle \hat{v}_2, \hat{\sigma}_2 \rangle = \hat{\mathcal{E}}\ 0\ [\![e_2]\!]\ \hat{\sigma} \textbf{ in} \\
 & & \textbf{let } \langle \hat{v}_3, \hat{\sigma}_3 \rangle = \hat{\mathcal{E}}\ 0\ [\![e_3]\!]\ \hat{\sigma} \textbf{ in} \\
 & & \langle \hat{v}_2 \sqcup \hat{v}_3, \hat{\sigma}_2 \sqcup \hat{\sigma}_3 \rangle \\
\hat{\mathcal{F}}\ \hat{\mathcal{E}}\ 0\ [\![f \lambda x.e]\!]\ \hat{\sigma} & = & \langle \{f \lambda x.e\}, \hat{\sigma} \rangle \\
\hat{\mathcal{F}}\ \hat{\mathcal{E}}\ 0\ [\![e_1\ \ e_2]\!]\ \hat{\sigma} & = & \textbf{let } \langle \hat{v}, \hat{\sigma}' \rangle = \hat{\mathcal{E}}\ 0\ [\![e_2]\!]\ \hat{\sigma} \textbf{ in} \\
 & & \textbf{let } \langle \hat{f}, \hat{\sigma}_0 \rangle = \hat{\mathcal{E}}\ 0\ [\![e_1]\!]\ \hat{\sigma}' \textbf{ in} \\
 & & \textbf{let } \langle \hat{v_1}, \hat{\sigma}_1 \rangle = \hat{\mathcal{E}}\ 0\ [\![b_1]\!]\ \hat{\sigma}'_1 \textbf{ in} \\
 & & \text{where } \hat{\sigma}'_1 = \hat{\sigma}_0[x_1 \mapsto \hat{v}][f_1 \mapsto \{f_1 \lambda x_1.b_1\}] \\
 & & \cdots \\
 & & \textbf{let } \langle \hat{v_n}, \hat{\sigma}_n \rangle = \hat{\mathcal{E}}\ 0\ [\![b_n]\!]\ \hat{\sigma}'_n \textbf{ in} \\
 & & \text{where } \hat{\sigma}'_n = \hat{\sigma}_0[x_n \mapsto \hat{v}][f_n \mapsto \{f_n \lambda x_n.b_n\}] \\
 & & \langle \hat{v_1} \sqcup \hat{v_2} \sqcup \cdots \sqcup \hat{v_n}, \hat{\sigma}_1 \sqcup \hat{\sigma}_2 \sqcup \cdots \sqcup \hat{\sigma}_n \rangle \\
 & & \text{where } \hat{f} = \{f_1 \lambda x_1.b_1, \cdots, f_n \lambda x_n.b_n\} \\
\hat{\mathcal{F}}\ \hat{\mathcal{E}}\ 0\ [\![\text{`}^l e]\!]\ \hat{\sigma} & = & \textbf{let } \langle g, \hat{\sigma}' \rangle = \hat{\mathcal{E}}\ 1\ [\![e]\!]\ \hat{\sigma} \textbf{ in} \\
 & & \langle l \otimes g, \hat{\sigma}' \rangle \\
\hat{\mathcal{F}}\ \hat{\mathcal{E}}\ 0\ [\![\textbf{run } e]\!]\ \hat{\sigma} & = & \textbf{let } \langle g, \hat{\sigma}' \rangle = \hat{\mathcal{E}}\ 0\ [\![e]\!]\ \hat{\sigma} \textbf{ in} \\
 & & \textbf{let } \langle \hat{v}_1, \hat{\sigma}_1 \rangle = \hat{\mathcal{E}}\ 0\ [\![e_1]\!]\ \hat{\sigma}' \textbf{ in} \\
 & & \cdots \\
 & & \textbf{let } \langle \hat{v}_n, \hat{\sigma}_n \rangle = \hat{\mathcal{E}}\ 0\ [\![e_n]\!]\ \hat{\sigma}' \textbf{ in} \\
 & & \langle \hat{v_1} \sqcup \hat{v_2} \sqcup \cdots \sqcup \hat{v_n}, \hat{\sigma}_1 \sqcup \hat{\sigma}_2 \sqcup \cdots \sqcup \hat{\sigma}_n \rangle \\
 & & \text{where } \psi(g) = \{\text{`}^{l_1} e_1, \cdots, \text{`}^{l_n} e_n\} \\
 & & \text{if } \psi(g) \text{ is not infinite} \\
\hat{\mathcal{F}}\ \hat{\mathcal{E}}\ 0\ [\![\textbf{run } e]\!]\ \hat{\sigma} & = & \textbf{let } \langle g, \hat{\sigma}' \rangle = \hat{\mathcal{E}}\ 0\ [\![e]\!]\ \hat{\sigma} \textbf{ in} \\
 & & \langle \top_g, \hat{\sigma}' \rangle \\
 & & \text{if } \psi(g) \text{ is infinite} \\
\hat{\mathcal{F}}\ \hat{\mathcal{E}}\ 1\ [\![c]\!]\ \hat{\sigma} & = & \langle \emptyset, \hat{\sigma} \rangle \\
\hat{\mathcal{F}}\ \hat{\mathcal{E}}\ 1\ [\![x]\!]\ \hat{\sigma} & = & \langle \emptyset, \hat{\sigma} \rangle \\
\hat{\mathcal{F}}\ \hat{\mathcal{E}}\ 1\ [\![\textbf{let } x\ e_1\ e_2]\!]\ \hat{\sigma} & = & \textbf{let } (\hat{v_1}, \hat{\sigma_1}) = \hat{\mathcal{E}}\ 1\ [\![e_1]\!]\ \hat{\sigma} \textbf{ in} \\
 & & \textbf{let } (\hat{v_2}, \hat{\sigma_2}) = \hat{\mathcal{E}}\ 1\ [\![e_2]\!]\ \hat{\sigma}_1 \textbf{ in} \\
 & & \langle \hat{v_1} \sqcup \hat{v_2}, \hat{\sigma}_2 \rangle \\
\hat{\mathcal{F}}\ \hat{\mathcal{E}}\ 1\ [\![\textbf{if } e_1\ e_2\ e_3]\!]\ \hat{\sigma} & = & \textbf{let } \langle \hat{v}_1, \hat{\sigma}_1 \rangle = \hat{\mathcal{E}}\ 1\ [\![e_1]\!]\ \hat{\sigma} \textbf{ in} \\
 & & \textbf{let } \langle \hat{v}_2, \hat{\sigma}_2 \rangle = \hat{\mathcal{E}}\ 1\ [\![e_2]\!]\ \hat{\sigma}_1 \textbf{ in} \\
 & & \textbf{let } \langle \hat{v}_3, \hat{\sigma}_3 \rangle = \hat{\mathcal{E}}\ 1\ [\![e_3]\!]\ \hat{\sigma}_2 \textbf{ in} \\
 & & \langle \hat{v_1} \sqcup \hat{v_2} \sqcup \hat{v_3}, \hat{\sigma}_3 \rangle \\
\hat{\mathcal{F}}\ \hat{\mathcal{E}}\ 1\ [\![f \lambda x.e]\!]\ \hat{\sigma} & = & \hat{\mathcal{E}}\ 1\ [\![e]\!]\ \hat{\sigma} \\
\hat{\mathcal{F}}\ \hat{\mathcal{E}}\ 1\ [\![e_1\ \ e_2]\!]\ \hat{\sigma} & = & \textbf{let } (\hat{v_1}, \hat{\sigma_1}) = \hat{\mathcal{E}}\ 1\ [\![e_1]\!]\ \hat{\sigma} \textbf{ in} \\
 & & \textbf{let } (\hat{v_2}, \hat{\sigma_2}) = \hat{\mathcal{E}}\ 1\ [\![e_2]\!]\ \hat{\sigma}_1 \textbf{ in} \\
 & & \langle \hat{v_1} \sqcup \hat{v_2}, \hat{\sigma}_2 \rangle \\
\hat{\mathcal{F}}\ \hat{\mathcal{E}}\ 1\ [\![,_a e]\!]\ \hat{\sigma} & = & \textbf{let } \langle g, \hat{\sigma}' \rangle = \hat{\mathcal{E}}\ 0\ [\![e]\!]\ \hat{\sigma} \textbf{ in} \\
 & & \langle a \oplus g, \hat{\sigma}' \rangle \\
\hat{\mathcal{F}}\ \hat{\mathcal{E}}\ 1\ [\![\textbf{run } e]\!]\ \hat{\sigma} & = & \hat{\mathcal{E}}\ 1\ [\![e]\!]\ \hat{\sigma}
\end{array}
$$

Figure 7. Abstract semantics

When $V = \emptyset$, $\alpha_v V = \hat{\bot}_v \sqsubseteq_v \hat{v}$.
When $V \in 2^{Code}$, $\alpha_v V = \xi(e_1) \sqcup \xi(e_2) \sqcup \cdots$ where $V = \{e_1, e_2, \cdots\}$. Thus, $\alpha_v V \sqsubseteq_v \hat{v}$ by definition of ψ. Thus,

$$V \subseteq \gamma_v \hat{v} \implies V \subseteq \alpha_v V$$

■

Lemma 4: 2^{Env} and $\hat{Env}$ are Galois connected with α_e and γ_e.

Proof:

1) Assume that $\Sigma \in 2^{Env}, \hat{\sigma} \in \hat{Env}$ such that $\alpha_e \Sigma \sqsubseteq_e \hat{\sigma}$. $\forall x \in Var$,

$$
\begin{array}{rcll}
\hat{\sigma} x & \sqsupseteq_v & (\alpha_e \Sigma) x & \text{(by assumption)} \\
 & = & (\lambda x.\alpha_v \bigcup_{\sigma \in \Sigma} \{\sigma x\}) x & \text{(by definition)} \\
 & = & \alpha_v \bigcup_{\sigma \in \Sigma} \{\sigma x\} & \\
 & = & \bigsqcup_{\sigma \in \Sigma} \alpha_v \{\sigma x\} & \text{(by lemma 3)}
\end{array}
$$

Therefore, $\forall \sigma \in \Sigma$, $\alpha_v \{\sigma x\} \sqsubseteq_v \hat{\sigma} x$.

$$
\begin{array}{rcll}
\alpha_v \{\sigma x\} & \sqsubseteq_v & \hat{\sigma} x & \\
\gamma_v \circ \alpha_v \{\sigma x\} & \subseteq & \gamma_v \hat{\sigma} x & \text{(by lemma 3)} \\
\therefore \{\sigma x\} & \subseteq & \gamma_v \hat{\sigma} x & \text{(by lemma 3)} \\
 & = & (\gamma_e \hat{\sigma}) x & \text{(by definition)}
\end{array}
$$

Thus,

$$\alpha_e \Sigma \sqsubseteq_e \hat{\sigma} \implies \Sigma \subseteq \gamma_e \hat{\sigma}$$

2) Assume that $\Sigma \in 2^{Env}, \hat{\sigma} \in \hat{Env}$ such that $\Sigma \subseteq \gamma_e \hat{\sigma}$.

$$
\begin{array}{rcll}
(\alpha_e \Sigma) x & = & \alpha_v \bigcup_{\sigma \in \Sigma} \{\sigma x\} & \text{(by definition)} \\
 & \sqsubseteq_v & \alpha_v \bigcup_{\sigma' \in \gamma_e \hat{\sigma}} \{\sigma' x\} & \text{(by assumption)}
\end{array}
$$

By definition of γ_e, $\sigma' x = v$ such that $v \in \gamma_v(\hat{\sigma} x)$.

$$\begin{aligned} \therefore (\alpha_e \Sigma) x &\sqsubseteq_v \alpha_v \bigcup_{\sigma' \in \gamma_e \hat{\sigma}} \{\sigma' x\} \\ &\sqsubseteq_v \alpha_v \bigcup_{v \in \gamma_v(\hat{\sigma} x)} \{v\} \\ &\sqsubseteq_v (\alpha_v \circ \gamma_v)(\hat{\sigma} x) \\ &\sqsubseteq_v \hat{\sigma} x \quad \text{(by lemma 3)} \end{aligned}$$

Thus,

$$\Sigma \subseteq \gamma_e \hat{\sigma} \implies \alpha_e \Sigma \sqsubseteq_e \hat{\sigma}$$

■

Let us define an abstraction between between $2^{Val \times Env}$ and $\hat{Val} \times \hat{Env}$.

Definition 2 (Abstraction):

$$\alpha_{v \times e} : 2^{Val \times Env} \to \hat{Val} \times \hat{Env}$$

$$\alpha_{v \times e}\ T = \begin{cases} \langle \hat{\perp}_v, \hat{\perp}_e \rangle & \text{if } T = \emptyset \\ \langle \bigsqcup_{\langle v, \sigma \rangle \in T} \alpha_v\{v\}, \bigsqcup_{\langle v, \sigma \rangle \in T} \alpha_e\{\sigma\} \rangle & \text{otherwise} \end{cases}$$

Lemma 5: $2^{Val \times Env}$ and $\hat{Val} \times \hat{Env}$ are Galois connected with $\alpha_{v \times e}$ and $\gamma_{v \times e}$.

Proof: Trivially, $2^{Val \times Env}$ and $\hat{Val} \times \hat{Env}$ are CPOs.

1) Assume that $T \in 2^{Val \times Env}, \langle \hat{v}, \hat{\sigma} \rangle \in \hat{Val} \times \hat{Env}$ such that $\alpha_{v \times e} T \sqsubseteq \langle \hat{v}, \hat{\sigma} \rangle$.
Let $T = \bigcup_{i \in N} \{\langle v_i, \sigma_i \rangle\}$. Then, by assumption

$$\begin{aligned} \alpha_{v \times e} T &= \langle \bigsqcup_i \alpha_v\{v_i\}, \bigsqcup_i \alpha_e\{\sigma_i\} \rangle \\ &\sqsubseteq \langle \hat{v}, \hat{\sigma} \rangle \end{aligned}$$

Therefore $\bigsqcup_i \alpha_v\{v_i\} \sqsubseteq_v \hat{v}$ and $\alpha_v\{v_i\} \sqsubseteq_v \hat{v}$. Thus, $\{v_i\} \subseteq \gamma_v \hat{v}$ by lemma 3 and $v_i \in \gamma_v \hat{v}$.
Similarly, $\sigma_i \in \gamma_e \hat{\sigma}$.
Let

$$\begin{aligned} A &= \gamma_{v \times e} \langle \hat{v}, \hat{\sigma} \rangle \\ &= \bigcup_{v \in \gamma_v \hat{v}, \sigma \in \gamma_e \hat{\sigma}} \{\langle v, \sigma \rangle\} \end{aligned}$$

Trivially, $\langle v_i, \sigma_i \rangle \in A$. Therefore

$$\begin{aligned} T &= \bigcup_{i \in N} \{\langle v_i, \sigma_i \rangle\} \\ &\subseteq A \\ &= \gamma_{v \times e} \langle \hat{v}, \hat{\sigma} \rangle \end{aligned}$$

Thus,

$$\alpha_{v \times e} T \sqsubseteq \langle \hat{v}, \hat{\sigma} \rangle \implies T \subseteq \gamma_{v \times e} \langle \hat{v}, \hat{\sigma} \rangle$$

2) Assume that $T \in 2^{Val \times Env}, \langle \hat{v}, \hat{\sigma} \rangle \in \hat{Val} \times \hat{Env}$ such that $T \subseteq \gamma_{v \times e} \langle \hat{v}, \hat{\sigma} \rangle$.
Let $T = \bigcup_{i \in N} \{\langle v_i, \sigma_i \rangle\}$. Then, by assumption

$$\begin{aligned} \bigcup_{i \in N} \{\langle v_i, \sigma_i \rangle\} &= T \\ &\subseteq \gamma_{v \times e} \langle \hat{v}, \hat{\sigma} \rangle \\ &= \bigcup_{v \in \gamma_v \hat{v}, \sigma \in \gamma_e \hat{\sigma}} \{\langle v, \sigma \rangle\} \end{aligned}$$

So, $\forall \langle v_i, \sigma_i \rangle \in T, \exists \langle v', \sigma' \rangle \in \gamma_v \langle \hat{v}, \hat{\sigma} \rangle$ such that $v_i = v'$and $\sigma_i = \sigma'$. Therefore

$$\begin{aligned} v' \in \gamma_v \hat{v} &\iff \{v'\} \subseteq \gamma_v \hat{v} \\ &\iff \{v_i\} \subseteq \gamma_v \hat{v} \\ &\iff \alpha_v\{v_i\} \sqsubseteq_v \hat{v} \end{aligned}$$

Similarly, $\alpha_e\{\sigma_i\} \sqsubseteq_e \hat{\sigma}$.

$$\begin{aligned} \therefore \alpha_{v \times e} T &= \alpha_{v \times e} (\bigcup_{i \in N} \{\langle v_i, \sigma_i \rangle\}) \\ &= \langle \bigsqcup_{\cdots} \alpha_v\{v_i\}, \bigsqcup_{\cdots} \alpha_e\{\sigma_i\} \rangle \\ &\sqsubseteq \langle \hat{v}, \hat{\sigma} \rangle \end{aligned}$$

Thus,

$$T \subseteq \gamma_{v \times e} \langle \hat{v}, \hat{\sigma} \rangle \implies \alpha_{v \times e} T \sqsubseteq \langle \hat{v}, \hat{\sigma} \rangle$$

■

E. Correctness of Abstractions

We should prove that the abstraction between collecting domains and abstract domains is sound. To show the correctness of abstraction, we should prove the following theorem.

Theorem 1: For $e \in Exp$, For arbitrary expression e,

$$\alpha_{v \times e} \circ fix\mathcal{F}[\![e]\!] \sqsubseteq fix\hat{\mathcal{F}}[\![e]\!] \circ \alpha_e$$

Proof: We will prove the theorem by fixpoint induction[7].
Let $P(f, g)$ be an assertion

$$P(f, g) = \forall s \in Stage, \forall e \in Exp : \alpha_{v \times e} \circ f\ s\ [\![e]\!] \sqsubseteq g\ s\ [\![e]\!] \circ \alpha_e$$

Base case: From lemma 4 and lemma 5, α_e and $\alpha_{v \times e}$ are strict. Therefore, $P(\perp, \perp)$ holds trivially.

Inductive case: Assume that for continuous functions $\mathcal{E}$ and $\hat{\mathcal{E}}$, $P(\mathcal{E}, \hat{\mathcal{E}})$ holds. Then we should show that

$$P(\mathcal{F}(\mathcal{E}), \hat{\mathcal{F}}(\hat{\mathcal{E}}))$$

The entire proof of the inductive case can be found in [8]. In this paper, we present the proof of the induction step for **run** expression. Other cases can be easily shown as similar.

- (**run** e: 0-stage)

Let

$$\begin{aligned} T &= \mathcal{E}\ 0\ [\![e]\!]\ \Sigma \\ \langle g, \hat{\sigma}' \rangle &= \hat{\mathcal{E}}\ 0\ [\![e]\!]\ \alpha_e \Sigma \end{aligned}$$

As $\alpha_{v \times e} T \sqsubseteq \langle g, \hat{\sigma}' \rangle$ by induction hypothesis, $\forall \langle e', \sigma \rangle \in T$, $\alpha_v\{e'\} \sqsubseteq_v g$ and $\alpha_e\{\sigma\} \sqsubseteq_e \hat{\sigma}'$ by definition. Thus, $\{e'\} \subseteq \gamma_v g$ and $e' \in \psi(g)$.
Let $L = \alpha_{v \times e}(\mathcal{F}\ \mathcal{E}\ 0\ [\![\mathbf{run}\ e]\!]\ \Sigma)$.

$$\begin{aligned} L &= \alpha_{v\times e}(\bigcup_{\langle e',\sigma\rangle \in VS} \mathcal{E}\ 0\ [\![e']\!]\ \{\sigma\}) \\ &\sqsubseteq \bigsqcup_{\langle e',\sigma\rangle \in VS} \alpha_{v\times e}(\mathcal{E}\ 0\ [\![e']\!]\ \{\sigma\}) \quad \text{(by lemma 5)} \\ &\sqsubseteq \bigsqcup_{\langle e',\sigma\rangle \in VS} \hat{\mathcal{E}}\ 0\ [\![e']\!]\ \alpha_e\{\sigma\} \quad \text{(by induction hypothesis)} \\ &\sqsubseteq \bigsqcup_{\langle e',\sigma\rangle \in VS} \hat{\mathcal{E}}\ 0\ [\![e']\!]\ \hat{\sigma} \quad \text{(by continuity of } \hat{\mathcal{E}}) \\ &\sqsubseteq \bigsqcup_{e' \in \psi(g)} \hat{\mathcal{E}}\ 0\ [\![e']\!]\ \hat{\sigma} \\ &= \hat{\mathcal{F}}\ \hat{\mathcal{E}}\ 0\ [\![\mathbf{run}\ e]\!]\ \alpha_e \Sigma \end{aligned}$$

■

IV. Test Coverage

In this chapter, we propose new test coverage metric for two-staged language using the analyzer described in chapter III.

Let $\hat{V}_{e,\hat{\sigma}} \in 2^{\hat{Val}}$ be a set of abstract values returned by expressions e on an abstract environment $\hat{\sigma}$ and for some abstract value set $\hat{V}$, let $\hat{V}^*$ be a set of all grammar elements in $\hat{V}$. Then $\bigcup_{g \in \hat{V}^*_{e,\hat{\sigma}}} \psi(g)$ means all expressions generated by expression e on abstract environment $\hat{\sigma}$. Now, among these expression set, let us define $\mathfrak{B}\ e\ \hat{\sigma}$ as a set of expression having branch.

Let $\#B_e$ be a number of branches generated in execution time, $\#B_c$ be a number of branches which already exist in the program text and $\#B_t$ be a number of branches tested by test case t. Then we can define new decision coverage (NDC) of test suite T on two-staged language program as follows.

Definition 3 (New Decision Coverage):

$$\begin{aligned} NDC(T) &= \frac{\sum_{t \in T} \#B_t}{\#B_c + \#B_e} \\ &= \frac{\sum_{t \in T} \#B_t}{\#B_c + 2 \times \text{cardinality of}(\mathfrak{B}\ e\ \emptyset)} \end{aligned}$$

Figure 8 is an example code to compare decision coverage with our new decision coverage. Labels of boxing expressions are $1, 2, \cdots, 10$ and alphabets of unboxing expressions are a, b, c and d.

```
f0λi.λj.
   let x = if i then ‘1(f1λv.v) else ‘2(f2λv.v) in
   let y = if i then ‘3(f3λv.v) else ‘4(f4λv.v) in
   let z = if i then ‘5(f5λv.v) else ‘6(f6λv.v) in
   let w = if i then ‘7(f7λv.v) else ‘8(f8λv.v) in
   if j then
     run ‘9(if i then ,a x else ,b y)
   else
     run ‘10(if i then ,c z else ,d w)
```

Figure 8. Example code for test coverages comparison

Axiomatically, $\#B_c = 2 \times 5 = 10$.
By analyzing this program, we've got result

$$\begin{aligned} \hat{V}^*_{e,\emptyset} &= \mathfrak{s} \to 9[a,b] \mid 10[c,d], \\ &\quad a \to 1|2, \quad b \to 3|4, \\ &\quad c \to 5|6, \quad d \to 7|8 \end{aligned}$$

We have 8 candidate code fragments with this grammar. Therefore,

$$\#B_e = 2 \times \textbf{cardinality of } \mathfrak{B}\ e\ \emptyset = 16$$

Table I
Test coverage for each test suite

Test Suite	Test Values	DC	NDC
A	$\{(0,0),(0,1)\}$	0.6	0.23
B	$\{(0,0),(0,1),(1,0)\}$	1.0	0.75
C	$\{(0,0),(0,1),(1,0),(1,1)\}$	1.0	0.88

We can achieve 60% decision coverage with a test suite (A) in table I because we can test 6 of 10 branches with the test suite. With test suite (B), we can test 10 branches and get 100% coverage but we do not have enough test cases yet because there exists more branches that would be generated in execution time. The decision coverage do not changed with test suite (C) but we can raise new decision coverage from 75% to 88% because we can test totally 16 branches.

What was interesting is that it is not possible to gain 100% new decision coverage sometimes. Consider the simple example code in figure 9. No matter how many test cases we have, we can never achieve 100% decision coverage because there is an unreachable code block. Similarly, our analyzer makes a result including all possible code fragments, but some code fragments in the result may not be generated actually. In this case, it is not possible to achieve 100% new decision coverage.

```
function foo(a) {
  if (a | MAX_INT) return 3;
  else return 4;
}
```

Figure 9. Example code including unreachable code block

V. Related Work

A basis of an abstract interpretation was founded by Cousot et al[3]. He showed that the abstract interpretation of programs consists in using that denotation to describe computations in another universe of abstract objects, so that the result of abstract execution give some informations on the actual computations. Thus, abstract interpretation can be used for static analysis.

Multi-staged language has a long history[9]. Keppl developed a portable system for modifying instruction spaces and presented that dynamic code generation can be effective for different applications. Poletto described the design and implementation of 'C, a high-level language for dynamic code generation[10]. With the study, it has been possible to compile and analyze multi-staged language easily. Despite of many studies on multi-staged language and test coverage, there were few tries to define or modify test coverage for multi-staged langauage.

Rajan claimed that test coverage should be used in carefully for each program style[11]. For example, test suites that provide MC/DC on the non-inlined implementation did poorly on the inlined implementations. They show that a coverage metric that takes masking into consideration irrespective of implementation structure, or a canonical way of structuring code so that condition masking is revealed when measuring coverage using existing coverage criteria.

VI. Conclusion

In this paper we have shown that applying existing test coverage metric to multi-staged language could be distorted because some code fragments can be generated dynamically in execution time. We proposed new test coverage metric for multi-staged language to correct the distortion. We designed an abstract analysis to estimate what code fragments would be generated dynamically during execution time and proved the correctness of the analysis. We claimed that our new test coverage can give more precise result than current test coverages and we hope that this technique would help developers to increase responsibility of software.

The accuracy of analysis should be improved. Advanced techniques in static analysis such as widening and narrowing operations could be used to enhance the analysis. The language specifiction discussed in this paper should be extended to cover many common language features such as long jump, global variables, exceptions and so on.

Improving the accuracy of the analysis should be followed by the implemention of an automated tool to measure our test coverage. The measurement tool could be used to show the efficiency of new test coverage. Our approach can also be used to extend other existing test coverages(statement, MC/DC and so on). We plan to study on conjugating static analysis technique to extend other test coverages.

Acknowledgment

This work was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology(MEST) / Korea Science and Engineering Foundation(KOSEF), grant number R11-2008-007-01002-0.

References

[1] I. Kim, K. Yi, and C. Calcagno, "A polymorphic modal type system for lisp-like multi-staged languages," in *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2006, pp. 257–269.

[2] S. Chilenski, J.J. Miller, "Applicability of modified condition/decision coverage to software testing," *Software Engineering Journal*, vol. 9, pp. 193–200, 1994.

[3] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. New York, NY, USA: ACM, 1977, pp. 238–252.

[4] W. V. Quine, *Mathematical Logic, Revised Edition*. Harvard University Press, 2003.

[5] N. Chomsky, "Three models for the description of language," *Information Theory, IEEE Transactions on*, vol. 2, no. 3, pp. 113–124, 1956. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1056813

[6] M. Erne, J. Koslowski, A. Melton, and G. E. Strecker, "A primer on galois connections," in *York Academy of Science*, 1992.

[7] J. D. U. Hopcroft, John E.; Rajeev Motwani, *Introduction to Automata Theory, Languages, and Computation (2nd ed.)*. Addison-Wesley, 2001.

[8] T. Kim, C. Lee, K. Lee, S. Baik, and K. Yi, "A control flow analysis for 2-staged programming languages," Research On Software Analysis for Error-free Computing Center, Seoul National University, Technical Memorandum ROSAEC-2009-005, September 2009.

[9] D. Keppel, "A portable interface for on-the-fly instruction space modification," *SIGARCH Comput. Archit. News*, vol. 19, no. 2, pp. 86–95, 1991.

[10] M. Poletto, W. C. Hsieh, D. R. Engler, and M. F. Kaashoek, "C and tcc: a language and compiler for dynamic code generation," *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 2, pp. 324–369, 1999.

[11] A. Rajan, M. W. Whalen, and M. P. Heimdahl, "The effect of program and model structure on mc/dc test adequacy coverage," in *ICSE '08: Proceedings of the 30th international conference on Software engineering*. New York, NY, USA: ACM, 2008, pp. 161–170.