PATL Specification

May 9, 2015

1 Normalized MJ

$$
\begin{array}{rcll}
p & ::= & \overline{cd} & \text{(Program)} \\
cd & ::= & \texttt{class C extends C } \{\overline{fd}\ cnd\ \overline{md}\} & \text{(Class Definition)} \\
fd & ::= & \texttt{C}\ f & \text{(Field Definition)} \\
cnd & ::= & \texttt{C}(\bar{\texttt{C}}\ \bar{x})\{\texttt{super}(\bar{e}); \bar{s}\} & \text{(Constructor)} \\
md & ::= & \tau\ m(\bar{\texttt{C}}\ \bar{x})\{\bar{s}\ \texttt{return}\ x;\} & \text{(Method Definition)} \\
\tau & ::= & \texttt{void} \mid \texttt{C} & \text{(Return Types)} \\
s & ::= & ps & \text{(Statement)} \\
 & \mid & \texttt{if}(x)\ \{\bar{s}\}\ \texttt{else}\ \{\bar{s}\} & \\
 & \mid & \texttt{while}(x)\{\bar{s}\} & \\
ps & ::= & x = e; \mid pe; & \text{(Primitive Statement)} \\
e & ::= & \texttt{null} \mid x \mid x.f \mid (\texttt{C})\ x \mid pe & \text{(Expression)} \\
pe & ::= & x.m(\bar{x}) \mid \texttt{new C}(\bar{x}) & \text{(Promotable Expression)}
\end{array}
$$

2 PATL Syntax

$$
\begin{array}{rcll}
\Pi & ::= & \emptyset \mid \Pi, \pi & \text{(Rule Sequence)} \\
\pi & ::= & (\bar{x} : \overline{\texttt{C} \hookrightarrow \texttt{C}})\ \{I\} & \text{(Transformation Rule)} \\
I & ::= & I^m; I^-; I^+ & \text{(Rule Body)} \\
I^m & ::= & \emptyset \mid \mathbf{m}\ p; I^m & \text{(Context Pattern)} \\
I^- & ::= & -\ p \mid -\ p; I^- & \text{(Source Pattern)} \\
I^+ & ::= & \emptyset \mid +\ p; I^+ & \text{(Target Pattern)} \\
p & ::= & x = r \mid r & \text{(Statement Pattern)} \\
r & ::= & x.m(\bar{x}) \mid \texttt{new C}(\bar{x}) \mid x.f & \text{(Expression Pattern)}
\end{array}
$$

3 Abstract Semantics

Definition 1 (Statement Match). *Given a transformation rule π, and a statement sequence $\bar{s}$, a statement match is a triple (p, σ, ι), where p is a statement pattern in π, σ is a map, mapping metavariables in π to variables in $\bar{s}$, and ι is the location of a statement in $\bar{s}$, s.t.*

1. Syntax match. $p \backslash \sigma = s$.
2. Type match. *For any metavariable x in p, $\mathsf{type}(\sigma(x)) <: \mathsf{type}(x)$ if x is used as a right value in p, and $\mathsf{type}(x) <: \mathsf{type}(\sigma(x))$ if x is used as a left value in p.*

Definition 2 (Statement Sequence Match). *A triple $(\overline{p}, \sigma, \bar{\iota})$ is a statement sequence match if (1) $|\overline{p}| = |\bar{\iota}|$, and (2) (p_i, σ, ι_i) is a statement match for any $i < |\overline{p}|$.*

Definition 3 (Rule Match). *A rule match between a statement sequence $\bar{s}$ and a rule π is a triple $(\pi, \sigma, \bar{\iota})$, where π is the rule of the form $(\bar{x} : \overline{\texttt{C}_1 \hookrightarrow \texttt{C}_2})\{I^m; I^-; I^+;\}$, σ is metavariable map, and $\bar{\iota} = (\iota_i, \ldots, \iota_n)$ is the locations of statements in $\bar{s}$, such that the following four conditions are satisfied:*

- Source match. *$(I^-, \sigma, \bar{\iota})$ is a statement sequence match.*

- Context match. *For any execution path $\bar{\iota}_p \in Path(p)$ passing $\bar{\iota}$, there exists a subsequence $\bar{\iota}_q$ before $\bar{\iota}$, where $(I^m, \sigma, \bar{\iota})$ is a statement sequence match and for any metavariable x in I^m, $\sigma(x)$ is not written between the matched statement and the first statement of $\bar{\iota}$.*
- Independence. *For any two metavariables, x_1 and x_2, $x_1 \neq x_2 \Rightarrow \sigma(x_1) \neq \sigma(x_2)$.*
- New variable. *For any metavariable x that appears only in I^+ but not in I^m or I^-, $\sigma(x)$ is a fresh variable that is different from any other variable in the program.*

Definition 4 (Match Set). *A match set S over a statement sequence $\bar{s}$ and a rule set Π is a set of matches over $\bar{s}$ where (1) $\pi \in \Pi$ for any match $(\pi, \sigma, \bar{\iota}) \in S$ and (2) for any statement location ι in $\bar{s}$, there is at most one match between a statement pattern p in I^- and the location ι.*

Definition 5 (Basic Transformation). *The sequence of the source statements $\bar{s}$ and the target statements $\bar{s}'$ conforms to a set of transformation rules Π under the basic semantics, denoted as $\Pi^0(\bar{s}, \bar{s}')$, if there exists a match set S over the source program p and the transformation rule set Π such that $\bar{s}' = \bar{s}\backslash\tau$ where τ is the smallest relation constructed by the following two rules.*

$$\frac{(\pi, \sigma, \bar{\iota}) \in S \qquad \pi = (\ldots)\{I^m; I^-; I^+;\}}{[\bar{\iota} \mapsto I^+\backslash\sigma] \in \tau}$$

$$\frac{(x_1 : \mathtt{C}_1 \hookrightarrow \mathtt{C}'_1, \ldots, x_n : \mathtt{C}_n \hookrightarrow \mathtt{C}'_n)\{I\} \in \Pi}{[\mathtt{C}_1 \mapsto \mathtt{C}'_1] \in \tau, \ldots, [\mathtt{C}_n \mapsto \mathtt{C}'_n] \in \tau}$$

4 Algorithmic Semantics

4.1 Contexts

$$\begin{array}{rcll}
\Gamma & ::= & \emptyset \mid \Gamma, v : \mathtt{C} & \text{(Typing context)} \\
\Psi & ::= & \emptyset \mid \Psi, x : \mathtt{C} \hookrightarrow \mathtt{C} & \text{(Metavariable Types)} \\
\Omega & ::= & \emptyset & \text{(Alias Set)} \\
 & \mid & \Omega, [v_1^{\iota_1}, v_2^{\iota_2}] & \\
 & \mid & \Omega, \langle v_1^{\iota_1}, v_2^{\iota_2} \rangle & \\
\Sigma & ::= & \emptyset & \text{(Dependency Relation)} \\
 & \mid & \Sigma, (l_1, l_2) & \\
 & \mid & \Sigma, (v_1^{\iota_1}, v_2^{\iota_2}) & \\
\Theta & ::= & \emptyset & \text{(Shifting Rule)} \\
 & \mid & \Theta, \langle \hat{\pi}; l_1, ..., l_n \rangle &
\end{array}$$

4.2 Matcher

$$\begin{array}{rcll}
\hat{\Pi} & ::= & \{\hat{\pi}_1, ..., \hat{\pi}_n\} & \text{(Matcher set)} \\
\hat{\pi} & ::= & [\pi](\bar{x} : \bar{S})\{\bar{\beta}\} & \text{(Matcher)} \\
S & ::= & \{\delta_1, ..., \delta_n\} & \text{(Binded Value Set)} \\
\delta & ::= & v^\iota & \text{(Binded Value)} \\
 & \mid & *v^\iota & \text{(Conditional Binded Value)} \\
v & ::= & x \mid x.f & \text{(Program Value)} \\
\beta & ::= & p : l & \text{(Statement Binding)} \\
 & \mid & *\, p : l\ [U] & \text{(Conditional Statement Binding)}
\end{array}$$

4.3 Transformation Process

$\boxed{\Pi(M) = M'}$

$$\frac{\begin{array}{c}\mathsf{depAnalysis}(\bar{s}) = \Sigma_d \quad \mathsf{aliasAnalysis}(\bar{s}) = \Omega \quad \{\bar{x} : \bar{\mathtt{C}}\} \vdash_\Omega \mathsf{Match}(\Pi, s) = \hat{\Pi} \\ \mathsf{MatchedSeqs}(\hat{\Pi}) = \Theta \quad \vdash^{\Theta}_{\Sigma_d \cup \mathsf{matchDep}(\Theta)} \bar{s} \rightsquigarrow_{shift} \bar{s}' \quad \mathsf{adapt}(\hat{\Pi}, \bar{s}') = \bar{s}''\end{array}}{\Pi(\mathtt{C}_1\ m(\bar{\mathtt{C}}\ \bar{x})\{\bar{s}; \mathtt{return}\ z;\}) = \Pi(\mathtt{C}_1)\ m(\overline{\Pi(\mathtt{C})}\ \bar{x})\{\bar{s}''; \mathtt{return}\ z;\}}$$

4.4 Type Transformation

$$\mathsf{TypeMapping}(\Pi) = \bigcup_{\pi \in \Pi} \mathsf{TypeMapping}(\pi)$$
$$\mathsf{TypeMapping}((\bar{x} : \overline{\mathtt{C} \hookrightarrow \mathtt{C}})\{I\}) = \{\overline{\mathtt{C} \hookrightarrow \mathtt{C}}\}$$

$\boxed{\Pi(\mathtt{C}) = \mathtt{C}'}$

$$\frac{\neg\exists \mathtt{C}.(\mathtt{C}_0 \hookrightarrow \mathtt{C} \in \mathsf{TypeMapping}(\Pi))}{\Pi(\mathtt{C}_0) = \mathtt{C}_0} \qquad \frac{\mathtt{C}_0 \hookrightarrow \mathtt{C} \in \mathsf{TypeMapping}(\Pi)}{\Pi(\mathtt{C}_0) = \mathtt{C}}$$

4.5 Matching

$$\begin{aligned}
&\mathit{init} = \{[\pi_1]()\{\}, ..., [\pi_n]()\{\}\} \\
&\mathit{out}[s] = \bigcup_{\hat{\pi} \in \mathit{in}[s]} \mathsf{match}(\hat{\pi}, s) \\
&\mathit{in}[s] = \bigcup_{s' \in \mathit{pred}(s)} \mathit{out}[s'] \\
&\mathit{exit} = \mathsf{clear}(\mathit{in}[s])
\end{aligned}$$

$\boxed{\Gamma \vdash_\Omega \mathsf{match}(\hat{\pi}, s) = \hat{\Pi}}$

$$\frac{\mathsf{matchpoint}(\hat{\pi}) = p_1 \quad \exists \hat{\pi}' = [\pi](\bar{x}' : \bar{S}')(\bar{p}' : \bar{l}') \in \mathit{in}[s].(p_1 : \mathsf{label}(s) \in \bar{p}' : \bar{l}')}{\Gamma \vdash_\Omega \mathsf{match}(\hat{\pi} :: [\pi](\bar{x} : \bar{S})\{\bar{\beta}\}, s) = \Big\{[\pi](\bar{x} : \bar{S})\{\bar{\beta}\}\Big\}}$$

$$\frac{\begin{array}{c}\mathsf{matchpoint}(\hat{\pi}) = p_1 \quad \mathsf{decl}(\pi); \Gamma \vdash \mathsf{varmatch}(p_1, s) = \bar{x}' : \overline{v'^{\iota'}} \\ \forall\ x_1 : v_1^{\iota 1} \in \bar{x}' : \overline{v'^{\iota'}}.(\exists x_1 : S_1 \in \bar{x} : \bar{S}.(S_1 = \emptyset \vee \forall v_2^{\iota 2} \in S.([v_1^{\iota 1}, v_2^{\iota 2}] \in \Omega))) \\ \mathsf{insert}(\bar{x}' : \overline{v'^{\iota'}}, \bar{x} : \bar{S}) = \bar{x} : \bar{S}'\end{array}}{\Gamma \vdash_\Omega \mathsf{match}(\hat{\pi} :: [\pi](\bar{x} : \bar{S})\{\bar{\beta}\}, s) = \Big\{[\pi](\bar{x} : \bar{S}')\{\bar{\beta},\ p_1 : \mathsf{label}(s)\}, [\pi](\bar{x} : \bar{S})\{\bar{\beta}\}\Big\}}$$

$$\frac{\begin{array}{c}\mathsf{matchpoint}(\hat{\pi}) = p_1 \quad \mathsf{decl}(\pi); \Gamma \vdash \mathsf{varmatch}(p_1, s) = \bar{x}' : \overline{v'^{\iota'}} \\ \forall\ x_1 : v_1^{\iota 1} \in \bar{x}' : \overline{v'^{\iota'}}.(\exists x_1 : S_1 \in \bar{x} : \bar{S}.(S_1 = \emptyset \vee \forall v_2^{\iota 2} \in S.([v_1^{\iota 1}, v_2^{\iota 2}] \vee \big\langle v_1^{\iota 1}, v_2^{\iota 2} \big\rangle \in \Omega))) \\ U = \big\{(x_1, v_1^{\iota 1}) \mid x_1 : v_1^{\iota 1} \in \bar{x}' : \overline{v'^{\iota'}} \wedge \exists\ x_1 : S_1 \in \bar{x} : \bar{S}.(\exists v_2^{\iota 2} \in S_1.(\big\langle v_1^{\iota 1}, v_2^{\iota 2} \big\rangle \in \Omega))\big\} \neq \emptyset \\ \mathsf{insert}(\bar{x}' : \overline{v'^{\iota'}}, \bar{x} : \bar{S}) = \bar{x} : \bar{S}'\end{array}}{\Gamma \vdash_\Omega \mathsf{match}(\hat{\pi} :: [\pi](\bar{x} : \bar{S})\{\bar{\beta}\}, s) = \Big\{[\pi](\bar{x} : \bar{S}')\{\bar{\beta},\ \ast p_1 : \mathsf{label}(s)\ [U]\}, [\pi](\bar{x} : \bar{S})\{\bar{\beta}\}\Big\}}$$

$$\frac{\text{otherwise}}{\Gamma \vdash_\Omega \mathsf{match}(\hat{\pi} :: [\pi](\bar{x} : \bar{S})\{\bar{\beta}\}, s) = \Big\{[\pi](\bar{x} : \bar{S})\{\bar{\beta}\}\Big\}}$$

$\boxed{\Psi; \Gamma \vdash \mathsf{varmatch}(p, s) = \big\{\bar{x} : \bar{v}\big\}}$

$$\frac{\Psi; \Gamma \vdash \mathsf{lhsvarmatch}(x; z) = \big\{x : z\big\} \quad \Psi; \Gamma \vdash \mathsf{varmatch}(r; e) = \big\{\bar{y} : \bar{v}\big\}}{\Psi; \Gamma \vdash \mathsf{varmatch}(x = r; z = e) = \big\{x : z, \bar{y} : \bar{v}\big\}}$$

$$\frac{\Psi \vdash x : \mathtt{C}_1 \hookrightarrow \mathtt{C}_1' \quad \Gamma \vdash x_1 : \mathtt{C}_2 \quad \mathtt{C}_2 <: \mathtt{C}_1}{\Psi; \Gamma \vdash \mathsf{varmatch}(x, x_1) = \big\{x : x_1\big\}}$$

$$\frac{\Psi \vdash \bar{x} : \overline{\mathtt{C}_1 \hookrightarrow \mathtt{C}_1'} \quad \Gamma \vdash \bar{x}_1 : \bar{\mathtt{C}}_2 \quad \bar{\mathtt{C}}_2 <: \bar{\mathtt{C}}_1}{\Psi; \Gamma \vdash \mathsf{varmatch}(\mathtt{new}\ \mathtt{C}(\bar{x}), \mathtt{new}\ \mathtt{C}(\bar{x}_1)) = \big\{\bar{x} : \bar{x}_1\big\}}$$

$$\frac{\begin{array}{c}\Psi \vdash x : \mathtt{C}_1 \hookrightarrow \mathtt{C}'_1,\ \bar{y} : \overline{\mathtt{C}_2 \hookrightarrow \mathtt{C}'_2} \\ \Gamma \vdash x_1 : \mathtt{C}_3, \bar{y}_1 : \bar{\mathtt{C}}_4 \qquad \mathtt{C}_3 <: \mathtt{C}_1, \bar{\mathtt{C}}_4 <: \bar{\mathtt{C}}_2\end{array}}{\Psi;\Gamma \vdash \mathsf{varmatch}(x.m(\bar{y}), x_1.m(\bar{y}_1)) = \left\{x : x_1, \bar{y} : \bar{y}_1\right\}}$$

$$\frac{\Psi \vdash x : \mathtt{C}_1 \hookrightarrow \mathtt{C}'_1 \quad \Gamma \vdash x_1 : \mathtt{C}_2 \quad \mathtt{C}_1 <: \mathtt{C}_2}{\Psi;\Gamma \vdash \mathsf{lhsvarmatch}(x, x_1) = \left\{\bar{x} : \bar{x}_1\right\}}$$

$\boxed{\mathsf{matchpoint}(\hat{\pi})}$

$$\frac{\begin{array}{c}\pi = (\bar{x} : \overline{\mathtt{C} \hookrightarrow \mathtt{C}})\{...\delta\ p_1; \delta\ p_2; ...\} \quad \delta = \mathbf{m} \mid - \\ \exists\ l_1.(p_1 : l_1 \in \{\bar{p} : \bar{l}\}) \wedge \neg\exists\ l_2.(p_2 : l_2 \in \{\bar{p} : \bar{l}\})\end{array}}{\mathsf{matchpoint}([\pi](\bar{x} : \bar{v})\{\bar{p} : \bar{l}\}) = p_2}$$

$$\frac{\pi = (\bar{x} : \overline{\mathtt{C} \hookrightarrow \mathtt{C}})\{...\delta\ p, I^+\ \} \quad \delta = \mathbf{m} \mid - \quad \exists\ l_1.(p_1 : l_1 \in \{\bar{p} : \bar{l}\})}{\mathsf{matchpoint}([\pi](\bar{x} : \bar{v})\{\bar{p} : \bar{l}\}) = \epsilon}$$

4.5.1 Matcher Checking

$\boxed{\hat{\Pi} \rightsquigarrow_{clean} \hat{\Pi}'}$

$$\frac{\hat{\Pi}' = \hat{\Pi} - \left\{\hat{\pi}\ :\ \hat{\pi} \in \hat{\Pi} \wedge \mathsf{matchpoint}(\hat{\pi}) \neq \epsilon\right\} \quad \forall \hat{\pi} \in \hat{\Pi}'.(\mathsf{check}(\hat{\pi}) \rightsquigarrow ok)}{\hat{\Pi} \rightsquigarrow_{clean} \hat{\Pi}'}$$

$\boxed{\mathsf{check}(\hat{\pi}) \rightsquigarrow}$

$$\frac{V = \left\{ *p : l\ [U]\ \middle|\ *p : l\ [U] \in \bar{\beta}\right\} \neq \emptyset}{\mathsf{check}([\pi](\ \bar{x} : \bar{S}\ )\{\bar{\beta}\}) \rightsquigarrow \textit{uncertain alias warning } V}$$

$$\frac{\exists\ p_1 : l_1, p_2 : l_2 \in \{\bar{p} : \bar{l}\}.(\mathsf{crossloop}(l_1, l_2))}{\mathsf{check}([\pi](\ \bar{x} : \bar{S}\ )\{\bar{\beta}\}) \rightsquigarrow \textit{cross loop boundary warning}}$$

$$\begin{array}{rl}\mathsf{crossloop}(l_1, l_2) = & \exists\ \bar{s}_1\ \mathtt{while}(x)\{\bar{s}_2\}\ \bar{s}_3.(\\ & \quad (\mathsf{stmt}(l_1) \in \bar{s}_1 \wedge \mathsf{stmt}(l_2) \in \bar{s}_2) \vee (\mathsf{stmt}(l_1) \in \bar{s}_2 \wedge \mathsf{stmt}(l_2) \in \bar{s}_3))\end{array}$$

$$\frac{\exists\ p_1 : l_1, p_2 : l_2 \in \{\bar{p} : \bar{l}\}.(\exists\ M_1, M_2.(\mathsf{stmt}(l_1) \in M_1 \wedge \mathsf{stmt}(l_2) \in M_2))}{\mathsf{check}([\pi](\ \bar{x} : \bar{S}\ )\{\bar{\beta}\}) \rightsquigarrow \textit{cross procedure warning}}$$

$$\frac{\neg\forall\ p_1 : l_1, p_2 : l_2 \in \{\bar{p} : \bar{l}\}.(p_1 \prec p_2 \Rightarrow l_1 \prec l_2)}{\mathsf{check}([\pi](\ \bar{x} : \bar{S}\ )\{\ \bar{p} : \bar{l}\}) \rightsquigarrow \textit{statement revisit warning}}$$

$$\frac{\text{otherwise}}{\mathsf{check}(\hat{\pi}) \rightsquigarrow ok}$$

4.6 Shifting

4.6.1 Definition

$$\begin{array}{rcll}\Theta & ::= & \emptyset \mid \Theta, \left\langle\hat{\pi}; l, ..., l\right\rangle & \text{(Shifting Rules)} \\ \mathsf{chunk}^{\hat{\pi}}[l] & ::= & \mathsf{stmt}^{\hat{\pi}}(l) & \text{(Statement Chunk Structure)} \\ & \mid & \mathtt{if}(x)\{\bar{s}_1\ \mathsf{chunk}^{\hat{\pi}}[l]\ \bar{s}_2\}\mathtt{else}\{\bar{s}_3\} & \\ & \mid & \mathtt{if}(x)\{\bar{s}_3\}\mathtt{else}\{\bar{s}_1\ \mathsf{chunk}^{\hat{\pi}}[l]\ \bar{s}_2\} & \end{array}$$

In the definition, we use Θ to represent the set of statement label sequences to represented the matched statements that are required to be shifted into basic blocks for transformation: $\left\langle\hat{\pi}; l_1, ..., l_n\right\rangle$ means that the statements labeled as $l_1, ..., l_n$ are required to be shifted into basic blocks, as they are matched by the matcher $\hat{\pi}$.

We also define a chunk structure given a matcher $\hat{\pi}$ and a statement label l: $\mathsf{stmt}^{\hat{\pi}}(l)$ is a statement s labeled as l, and it is currently activated by the matcher activation operation $s^{\uparrow\hat{\pi}}$, and chunk structures are defined over blocks that are activated by $\hat{\pi}$ and contains the statement labeled as l inside. Note that once a chunk is deactivated by the deactivated operation $\mathsf{chunk}^{\downarrow\hat{\pi}}[l]$, it is no longer a chunk of l over π, and all blocks inside the chunk are also deactivated.

In the shifting phase, we keep a set of "shifted sequences" $\Theta^{\#}$ to keep track of all completed shift operations, at the beginning of the shifting stage, the set $\Theta^{\#}$ is empty. We use the set $\Theta^{\#}$ in the shifting stage to ensure that the statements that are already shifted into basic blocks should always be shifted together to ensure congruency.

4.6.2 Shifting Algorithm

```
Input: Statement sequence $\bar{s}$, Shifting set $\Theta$, Dependency relation set $\Sigma$
Output: Shifted statement sequence $\bar{s}'$
begin
    $\Theta^{\#} \leftarrow \emptyset$;
    for each shifting relation $\theta = \langle \hat{\pi}; l_1, ..., l_n \rangle \in \Theta$ do
        for each adjacent labels $l_k, l_{k+1}$ do
            /* Perform the shift rule with the given $\theta$ */
            $\bar{s} \leftarrow \mathsf{shift}(\Theta, \Sigma, \Theta^{\#}, \langle \hat{\pi}; l_k, l_{k+1} \rangle, \bar{s})$;
            $\Theta^{\#} \leftarrow \Theta^{\#} \cup \{\langle \hat{\pi}; l_1, l_2 \rangle\}$;
        end
    end
end
```

Algorithm 1: Shifting algorithm

4.6.3 Semantic Rules

In the rules, we use $\mathsf{closure}^{\downarrow}(\bar{s}_1, \bar{s}_2)$ to represent iteratively find all statements in $\bar{s}_2$ that depends on one or more statements in $\bar{s}_1$, and $\mathsf{closure}^{\uparrow}(\bar{s}_1, \bar{s}_2)$ to present iteratively find all statements in $\bar{s}_2$ that $\bar{s}_1$ depends on.

We also use $s \not\leftarrow_{dep} x$ to denote that the variable x does not depends on s.

$$\frac{\vdash^{\Theta}_{\Sigma} \mathsf{upshift}(\langle \hat{\pi}; l_1, l_2 \rangle, \bar{s}) = \bar{s}'}{\mathsf{shift}(\Theta, \Sigma, \Theta^{\#}, \langle \hat{\pi}; l_1, l_2 \rangle, \bar{s}) = \bar{s}'} \text{ (Shift)}$$

$$\frac{\begin{array}{c}\mathsf{chunk}^{\hat{\pi}}[l_2] = \texttt{if}(x)\{\bar{s}_{11}\}\texttt{else}\{\bar{s}_{21}\} \quad \exists\, \mathsf{chunk}^{\hat{\pi}}[l_1] \in \bar{s}_{11} \vee \exists\, \mathsf{chunk}^{\hat{\pi}}[l_1] \in \bar{s}_{21} \\ \vdash^{\Theta;\Theta^{\#}}_{\Sigma} \mathsf{upshift}(\langle \hat{\pi}; l_1, l_2 \rangle, \bar{s}_{11}) = \bar{s}'_{11} \quad \vdash^{\Theta;\Theta^{\#}}_{\Sigma} \mathsf{upshift}(\langle \hat{\pi}; l_1, l_2 \rangle, \bar{s}_{21}) = \bar{s}'_{21}\end{array}}{\vdash^{\Theta;\Theta^{\#}}_{\Sigma} \mathsf{upshift}(\langle \hat{\pi}; l_1, l_2 \rangle, \bar{s}_1\ \mathsf{chunk}^{\hat{\pi}}[l_2]\ \bar{s}_2) = \bar{s}_1\ \texttt{if}(x)\{\bar{s}'_{11}\}\texttt{else}\{\bar{s}'_{21}\}\ \bar{s}_2} \text{ (Upshift-0)}$$

$$\frac{\vdash^{\Theta;\Theta^{\#}}_{\Sigma} \mathsf{downshift}(\langle \hat{\pi}; l_1, l_2 \rangle, \mathsf{stmt}^{\hat{\pi}}(l_1)\ \bar{s}_2) = \bar{s}'}{\vdash^{\Theta;\Theta^{\#}}_{\Sigma} \mathsf{upshift}(\langle \hat{\pi}; l_1, l_2 \rangle, \bar{s}_1\ \mathsf{stmt}^{\hat{\pi}}(l_1)\ \bar{s}_2) \;=\; \bar{s}_1\ \bar{s}'} \text{ (Upshift-1)}$$

$$\frac{\begin{array}{c}\exists\, \mathsf{chunk}^{\hat{\pi}}[l_1] \in \bar{s}_1 \vee \mathsf{chunk}^{\hat{\pi}}[l_1] \in \bar{s}_2 \quad \vdash^{\Theta;\Theta^{\#}}_{\Sigma} \mathsf{closure}^{\uparrow}(\mathsf{chunk}^{\hat{\pi}}[l_2], \bar{s}_3) = \bar{s}' \\ \mathsf{upshift}(\langle \hat{\pi}; l_1, l_2 \rangle, \bar{s}_1\ \bar{s}'\ \mathsf{chunk}^{\hat{\pi}}[l_2]) = \bar{s}''_1 \quad \mathsf{upshift}(\langle \hat{\pi}; l_1, l_2 \rangle, \bar{s}''_2\ \bar{s}'\ \mathsf{chunk}^{\hat{\pi}}[l_2]'') = \bar{s}''_2 \\ \exists \theta = \langle \hat{\pi}_x; l_u, l_v \rangle \in \Theta^{\#}.(\exists s_u \in \bar{s}' \wedge s_u = \mathsf{stmt}^{\hat{\pi}_x}(l_u)) \\ \mathsf{upshift}(\theta,\ \bar{s}_0\ \texttt{if}(x)\{\bar{s}''_1\}\texttt{else}\{\bar{s}''_2\}\ \bar{s}_3 \backslash \bar{s}'\ \bar{s}_4) = \bar{s}'''\end{array}}{\begin{array}{c}\vdash^{\Theta;\Theta^{\#}}_{\Sigma} \mathsf{upshift}(\langle \hat{\pi}; l_1, l_2 \rangle, \bar{s}_0\ \texttt{if}(x)\{\bar{s}_1\}\texttt{else}\{\bar{s}_2\}\ \bar{s}_3\ \mathsf{chunk}^{\hat{\pi}}[l_2]\ \bar{s}_4) \\ = \bar{s}'''\end{array}} \text{ (Upshift-2)}$$

$$\frac{\neg\exists\, \mathsf{chunk}^{\hat{\pi}}[l_1] \in \bar{s}_1}{\vdash^{\Theta;\Theta^{\#}}_{\Sigma} \mathsf{upshift}(\langle \hat{\pi}; l_1, l_2 \rangle, \bar{s}_1\ \mathsf{chunk}^{\hat{\pi}}[l_2]\ \bar{s}_3) \;=\; \bar{s}_1\ \mathsf{chunk}^{\uparrow\hat{\pi}}[l_2]\ \bar{s}_3} \text{ (Upshift-3)}$$

$$\frac{}{\begin{array}{c}\vdash^{\Theta;\Theta^{\#}}_{\Sigma} \mathsf{downshift}(\langle \hat{\pi}; l_1, l_2 \rangle, \mathsf{stmt}(l_1)^{\hat{\pi}}\ \bar{s}_1\ \mathsf{stmt}^{\hat{\pi}}(l_2)\ \bar{s}_2) \\ =\ \mathsf{stmt}(l_1)^{\hat{\pi}}\ \bar{s}_1\ \mathsf{stmt}^{\hat{\pi}}(l_2)\ \bar{s}_2\end{array}} \text{ (Downshift-1)}$$

$$\frac{\begin{array}{c}\exists \mathsf{chunk}^{\hat{\pi}}[l_2] \in \bar{s}_2 \vee \mathsf{chunk}^{\hat{\pi}}[l_2] \in \bar{s}_3 \quad \vdash_{\Sigma}^{\Theta;\Theta^{\#}} \mathsf{closure}^{\uparrow}(\mathsf{stmt}^{\hat{\pi}}(l_1), \bar{s}_1) = \bar{s}' \quad \mathsf{stmt}^{\hat{\pi}}(l_1) \not\leftarrow_{dep} x \\ \vdash_{\Sigma}^{\Theta;\Theta^{\#}} \mathsf{downshift}(\langle \hat{\pi}; l_1, l_2 \rangle, \mathsf{stmt}^{\hat{\pi}}(l_1)\ \bar{s}'\ \bar{s}_2) = \bar{s}_2'' \quad \vdash_{\Sigma}^{\Theta;\Theta^{\#}} \mathsf{downshift}(\langle \hat{\pi}; l_1, l_2 \rangle, \mathsf{stmt}^{\hat{\pi}}(l_1)\ \bar{s}'\ \bar{s}_3) = \bar{s}_3'' \\ \exists \theta = \langle \hat{\pi}_x; l_u, l_v \rangle \in \Theta^{\#}.(\exists s_v \in \bar{s}' \wedge s_v = \mathsf{stmt}^{\hat{\pi}_x}(l_v)) \quad \mathsf{downshift}(\theta,\ \bar{s}_0\ \bar{s}_1 \backslash \bar{s}'\ \mathtt{if}(x)\{\bar{s}_2''\}\mathtt{else}\{\bar{s}_3''\}) = \bar{s}''' \end{array}}{\vdash_{\Sigma}^{\Theta;\Theta^{\#}} \mathsf{downshift}(\langle \hat{\pi}; l_1, l_2 \rangle, \bar{s}_0\ \mathsf{stmt}^{\hat{\pi}}(l_1)\ \bar{s}_1\ \mathtt{if}(x)\{\bar{s}_2\}\mathtt{else}\{\bar{s}_3\}\ \bar{s}_4) \ = \ \bar{s}'''\ \bar{s}_4} \text{(Downshift-2)}$$

$$\frac{\text{otherwise}}{\begin{array}{c}\vdash_{\Sigma}^{\Theta;\Theta^{\#}} \mathsf{downshift}(\langle \hat{\pi}; l_1, l_2 \rangle, \bar{s}_0\ \mathsf{stmt}^{\hat{\pi}}(l_1)\ \bar{s}_1\ \mathtt{if}(x)\{\bar{s}_2\}\mathtt{else}\{\bar{s}_3\}\ \bar{s}_4) \\ \rightsquigarrow \textit{Cannot shift due to dependency of } x. \end{array}} \text{(Downshift-2-Warning)}$$

$$\frac{\neg\exists\ \mathsf{chunk}^{\hat{\pi}}[l_2] \in \bar{s}_2}{\vdash_{\Sigma}^{\Theta;\Theta^{\#}} \mathsf{downshift}(\langle \hat{\pi}; l_1, l_2 \rangle, \mathsf{stmt}^{\hat{\pi}}(l_1)\ \bar{s}_2) \ = \ \mathsf{chunk}^{\downarrow\hat{\pi}}[l_1]\ \bar{s}_2)} \text{(Downshift-3)}$$

Post shifting check:

$$\frac{\begin{array}{l}\neg\exists s \in \bar{s}.(\exists \pi_1, \pi_2 \in \hat{\Pi}.(\exists p_1 : l \in \pi_1, p_2 : l \in \pi_2.(\\ \quad \mathsf{label}(s) = l \wedge p_1 \textit{ and } p_2 \textit{ are deletion pattern}))) \end{array}}{\textit{Interleaving source pattern matching warning}}$$

4.7 Adaptation

4.7.1 Code Generation Point

Swappable Position Given a sequence of primitive statements $\overline{ps} = ps_1; ...ps_t; ...; ps_n$, and statements dependency relation set $\Sigma = \{(l_a, l_b), ...\}$, we define the swappable position of a statement ps_t as an index set $I_{swap}(ps_t) \subset [1, ..., n]$, which $\forall i \in I_{swap}(ps_t)$, we can move the statement ps_t to the position right after the current statement ps_i through statement swapping operation.

In the formal definition, we use the $ps.l$ to present getting the label of the statement ps.

$$\begin{array}{l}
I_{swap}(ps_0,\ \overline{ps}) = [I_{swap}^{\leftarrow}(ps_0,\ \overline{ps}),\ I_{swap}^{\rightarrow}(ps_0,\ \overline{ps})] \\
I_{swap}^{\leftarrow}(ps_0,\ ps_0; \overline{ps}) = \mathsf{Pos}(ps_0) \\
I_{swap}^{\leftarrow}(ps_0,\ \overline{ps}_1; ps_0; \overline{ps}_2) = \max\Big\{ \min\big\{ I_{swap}^{\leftarrow}(ps_i,\ \overline{ps}_1; ps_0; \overline{ps}_2) \ \big|\ ps_i \in \overline{ps}_1 \wedge (\mathsf{label}(ps_0), \mathsf{label}(ps_i)) \in \Sigma \big\} + 1, \\
\qquad\qquad\qquad\qquad\qquad\qquad \#\big\{ ps \ \big|\ ps \in \bar{p}s_1 \wedge ps_0 \textit{ depends on } ps \big\} + 1 \Big\} \\
I_{swap}^{\rightarrow}(ps_0,\ \overline{ps}; ps_0) = \mathsf{Pos}(ps_0) \\
I_{swap}^{\rightarrow}(ps_0,\ \overline{ps}_1; ps_0; \overline{ps}_2) = \min\Big\{ \max\big\{ I_{swap}^{\rightarrow}(ps_i,\ \overline{ps}_1; ps_0; \overline{ps}_2) \ \big|\ s_i \in \overline{ps}_2 \wedge (\mathsf{label}(ps_0), \mathsf{label}(ps_i)) \in \Sigma \big\} - 1, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{label}(ps_0) + |\overline{ps}_2| - \#\big\{ ps \ \big|\ ps \in \bar{p}s_2 \wedge ps \textit{ depends on } ps_0 \big\} \Big\}
\end{array}$$

Given a matcher $\hat{\pi} = [\pi](\bar{x} : \bar{S})\{p_1 : l_1; ..; p_r : l_r\}$ and a sequence of primitive statements $\overline{ps} = ps_1; ps_2; ...; ps_n;$, $\mathsf{genPoint}(\hat{\pi}, \overline{ps})$ is evaluated by:

$$\frac{\hat{\pi} = [\pi](\bar{x} : \bar{S})\{\bar{p} : \bar{l}\} \quad S = \big\{ ps_i \ \big|\ ps_i \in \overline{ps} \wedge \exists p_i : l_i \in \bar{p} : \bar{l}.(\mathsf{label}(ps_i) = l_i) \big\}}{\mathsf{genPoint}(\hat{\pi}, \overline{ps}) = \bigcap_{ps_i \in S} I_{swap}(ps_i,\ \overline{ps})} \text{(GenPoint)}$$

4.7.2 Partial Filled Statements Generation

In this part, we present how to generate a partial code fraction from a matcher $\hat{\pi} = [\pi](\bar{x} : \bar{S})\{\bar{p} : \bar{l}\}$, given the code context $\bar{s}$, by partial code, we will not directly substitute all metavariables with MJ program variables, instead, we generate a notation at the corresponding variable location with its constraints, and we will fill in program variables after code position is confirmed.

$$\frac{\pi = (\bar{x} : \overline{\mathtt{C} \hookrightarrow \mathtt{C}})\ \{I^m; I^-; I^+;\} \quad \mathsf{MetaVariables}(I^+) = \{x_1, ..., x_n\} \quad \forall i.x_i : S_i \in \bar{x} : \bar{S}}{\mathsf{genPartialStmts}([\pi](\bar{x} : \bar{S})\{\bar{p} : \bar{l}\}) = [x_1 \mapsto \mathsf{alias}(S_1), ..., x_n \mapsto \mathsf{alias}(S_n)]I^+}$$

```
Input: Statement sequence $\bar{s}$, corresponding matcher set $\hat{\Pi}$
Output: Transformed statement sequence $\bar{s}''$
begin
    /*Generate a map that maps each referenced variable $v^{\iota}$ to its definition $v_s^{\iota_s}$ */;
    $\phi \leftarrow \mathsf{aliasToDefMap}(\hat{\Pi}, \bar{s})$;
    $Frags \leftarrow \{\}$;
    for each basic block $\overline{ps}$ in $\bar{s}$ do
        for $\hat{\pi} \in \hat{\Pi}$ do
            if no statement tagged with active labels of $\hat{\pi}$ exists in $\overline{ps}$ then
                continue;
            end
            $\bar{s}^0 \leftarrow \mathsf{genPartialStmts}(\hat{\pi}, \overline{ps})$;
            $Loc \leftarrow \mathsf{genPos}(\hat{\pi}, \overline{ps})$;
            if $Loc! = \emptyset$ then
                Randomly choose $loc \in Loc$;
                $Frags \leftarrow Frags \cup \{(\bar{s}^0, loc, \hat{\pi})\}$;
            else
                Empty generating position warning
            end
        end
    end
    $\bar{s}' \leftarrow \bar{s}$;
    for each $(\bar{s}^0, loc, \hat{\pi}) \in Frags$ do
        /*Delete all primitive statements binded to a deletion pattern*/
        $\bar{s}_1 \leftarrow \mathsf{deleteMatchedStmts}(\bar{s}', \hat{\Pi})$;
        /*insert $\bar{s}^0$ in its corresponding location in $\bar{s}_1$ */
        $\bar{s}'' \leftarrow \mathsf{insertPartialCode}(\bar{s}^0, \bar{s}_1)$;
        /*Backfill program variables in $\bar{s}$ to annotated alias locations*/
        $\bar{s}'' \leftarrow \mathsf{backFill}(\bar{s}'', \bar{s}_1, \phi)$;
        $\phi \leftarrow \mathsf{updateLocationMap}(\phi, \bar{s}'', \bar{s}', \hat{\pi})$;
    end
end
```

Algorithm 2: Code Adaptation Algorithm, the location map function ϕ is a function that maps a set of aliases to the first definition of the alias, i.e. $\phi(S) = v^{\iota}$, s.t. $\forall v_1^{\iota_1} \in S.(v^{\iota}\ and\ v_1^{\iota_1}\ are\ aliases)$.

```
Input: A statement sequence $\bar{s}$, partial filled statements $\bar{s}'$, location map $\phi$
Output: Backfilled statement sequence $\bar{s}'$
begin
    for each alias notation $\chi = \mathsf{alias}(S)$ do
        /* extract the location of $\chi$ */
        $\iota_\chi \leftarrow \mathsf{location}(\chi)$;
        $v^\iota \leftarrow \phi(S)$;
        /* If the name $v$ remains an alias of $v^\iota$ */
        if $v^\iota$ exists in the $\bar{s}$ then
            if $v^\iota$ and $v^{\iota_\chi}$ are aliases then
                /* Fill the position with $v$ */
                $\chi \leftarrow v$;
            else
                $u \leftarrow$ fresh variable name;
                /*Generate a copy of $v^\iota$ using $u$ right after the location $\iota$;*/
                $\mathsf{insert}(u = v; , \iota + 1)$;
                /* Fill the position with $u$ */
                $\chi \leftarrow u$;
            end
        else
            /* The variable is defined in the deleted code, thus $v$ is a free name now. */
            $\chi \leftarrow v$;
        end
    end
end
```

Algorithm 3: Backfill algorithm $\mathsf{backfill}(\bar{s}', \bar{s})$

```
Input: The original location map $\phi$, new statements $\bar{s}''$, old statements $\bar{s}', \hat{\pi}$
Output: Updated alias definition mapping $\phi$
begin
    $\gamma \leftarrow \mathsf{id}$;
    for each $x : S \in \mathsf{varBindings}(\hat{\pi})$ do
        $v^\iota \leftarrow \phi(S)$;
        if $v^\iota \in \bar{s}' \wedge v^\iota \notin \bar{s}''$ then
            /* In this case, $v^\iota$ is defined in the deleted code. According to
              * the backfill algorithm, $v$ will be used as the generation name of $x$ in $\bar{s}''$ */
            $v$ is defined in loctaion $\iota'$ in $\bar{s}''$;
            $\gamma \leftarrow \gamma \circ [v^\iota \mapsto v^{\iota'}]$
        end
    end
    for all $x_1 : S_1, x_2 : S_2 \in \mathsf{varBindings}(\hat{\pi})$ do
        if $\phi(S_1)$ depends on $\phi(S_2)$ in $\bar{s}' \wedge (\gamma \circ \phi)(S_2)$ depends on $(\gamma \circ \phi)(S_1) \in \bar{s}''$ then
            Report contradictory dependency relation warning;
        end
    end
    $\phi \leftarrow \gamma \circ \phi$
end
```

Algorithm 4: The update algorithm for aliasToDefinition map. updateLocationMap

4.7.3 Program Adaptation Algorithm

5 Theory

5.1 Semantic-preserving Operations

5.1.1 Equivalence Relation Definition

Definition 6. *(Swapping) Given two statement sequences* $\bar{s} = \bar{s}_0; s_1; s_2; \bar{s}_3;$, $\bar{s}' = \bar{s}_0; s_2; s_1; \bar{s}_3;$, $\bar{s}$ *and* $\bar{s}'$ *are swapping equivalent* ($\bar{s} \doteq_s \bar{s}'$) *if* s_1 *and* s_2 *have no dependency in* $\bar{s}$.

Definition 7. *(Renaming) Given two statement sequences* $\bar{s}$, $\bar{s}'$, $\bar{s}$ *is renaming equivalent to* $\bar{s}'$ ($\bar{s} \doteq_r \bar{s}'$) *if there exists an alias map* α *over* $\bar{s}_1$, *s.t.* $\forall (v,l) \in \mathsf{dom}(\alpha)$, *by substituting* v *at location of* l *with its image* $\alpha((v,l))$, $\bar{s} \backslash \alpha \equiv \bar{s}'$.

Definition 8. *(Primitive Upshift) Given two statement sequences* $\bar{s}$ *and* $\bar{s}'$, *if* $\bar{s} = \bar{s}_0; \mathtt{if}(x)\{\bar{s}_{11}\}\mathtt{else}\{\bar{s}_{12}\}s_2; \bar{s}_3$ *and* $\bar{s}' = \bar{s}_0; \mathtt{if}(x)\{\bar{s}_{11}; s_2\}\mathtt{else}\{\bar{s}_{12}; s_2\}\bar{s}_3$, *then* $\bar{s}$ *and* $\bar{s}'$ *are upshifting equivalent.* ($\bar{s} \doteq_{us} \bar{s}'$)

Definition 9. *(Primitive Downshift) Given two statement sequences* $\bar{s} = \bar{s}_0; s_1; \mathtt{if}(x)\{\bar{s}_{21}\}\mathtt{else}\{\bar{s}_{22}\}\bar{s}_3$, $\bar{s}' = \bar{s}_0; \mathtt{if}(x)\{s_1; \bar{s}_{21};\}\mathtt{else}\{s_1; \bar{s}_{22};\}\bar{s}_3$, *if* x *have no dependency with the statement* s_1 *in* $\bar{s}$, *then* $\bar{s}$ *and* $\bar{s}'$ *are downshift equivalent.* ($\bar{s} \doteq_{ds} \bar{s}'$)

Definition 10. *(Free-Variable) Given a statement sequence* $\bar{s} = \bar{s}_1; \bar{s}_2$, *another statement sequences* $\bar{s}' = \bar{s}_1; x = v; \bar{s}_2$ *is fv-equivalent to* $\bar{s}$ *if* x *is a fresh variable in* $\bar{s}$ *and* v *is a variable in* $\bar{s}$, *and reversely, another statement sequences* $\bar{s}'' = \bar{s}_1 \backslash (x = v;); \bar{s}_2$ *is also fv-equivalent to* $\bar{s}$ *if* x *is never used in* $\bar{s}_2$. ($\bar{s}'' \doteq_{fv} \bar{s} \doteq_{fv} \bar{s}'$)

Definition 11. *(Statements Equivalence) Two statements sequences* $\bar{s}$ *and* $\bar{s}'$ *are equivalence* ($\bar{s} \doteq \bar{s}'$) *if there exists statements* $\bar{s}_1, ..., \bar{s}_n$, *s.t.* $\bar{s} \doteq_{i_1} \bar{s}_1 \doteq_{i_2} ... \doteq_{i_n} \bar{s}_n$, *where* $\doteq_{i_j}$ *is one of* $\doteq_s, \doteq_r, \doteq_{us}, \doteq_{ds}, \doteq_{fv}$.

Property 1. *The Statement Equivalence relation defined above is an equivalence relation between statement sequences.*

Proof. It's not hard to prove its reflexivity, symmetry and transitivity between statements. □

5.1.2 Main Theorem

Theorem 1. *For any program* p *and* p' *such that* $\Pi(p) = p'$, *there exists two programs* q *and* q' *such that* $p \doteq q$, $p' \doteq q'$, *and* $\Pi^0(q, q')$, *i.e. the following diagram commutes.*

$$\begin{tikzcd} q \arrow[r, "\Pi^0"] \arrow[d, leftrightarrow, "\doteq"] & q' \arrow[d, leftrightarrow, "\doteq"] \\ p \arrow[r, "\Pi"'] & p' \end{tikzcd} \qquad (1)$$

Proof Sketch. Firstly, in the shifting stage, the shift of all statements can be decomposed to primitive shift statements, as statements dependency relations are not contravened (We denote the shifting phase as α). Secondly, the generation point *pt* in the adaptation algorithm is a point that the all matched statements can be swapped to (we denote the imaginary swapping phase as β), as for each statement *pt* is a location that the statement can be swapped to. Thirdly, suppose there exists $x : S \in \mathsf{varBinding}(\Pi)$, and x is only bound to read variables, we can rename all bound values to the first definition of the alias. The variable bindings can then be mapped to bindings in Π^0, suppose the map is γ_1. Thus the transformation can be performed in term of Π^0, and the result is p'. After transformation, by renaming them back with γ_1^{-1} (by recognizing the location of each variable), we have that $\Pi^0(\gamma(\beta(\alpha(p))), \gamma^{-1}(p'))$, thus the theorem is proved. □

Proof. In our proof, we will decompose the transformation $\Pi(p)$ into the following three phases: 1) $shift(p)$ 2) $findGenPoint(p)$ and 3) $StatementSubstitution(p)$.

The key point of the proof is to prove 1) the program after shifting is semantically equivalent to the original program, i.e. $p \doteq p_{shifted}$, 2) The generation point in p is a place where we can switch all statements to, and 3) code generation preserve that variable condition specified in the abstract semantics.

According to Lemma 1, the shifting operation on the program is semantically equivalent.

According to Lemma 2, the code generation point is the place where all statements can be moved to through semantically equivalent operation primitive swap.

Then for the code generation phase, as the variables are matched with alias analysis and they are inside single blocks, we can generate a in-block SSA form for the matched statements and thus the transformation is semantically equivalent to the abstract semantics.

After transformation, by renaming them back with γ_1^{-1} (by recognizing the location of each variable), we have that $\Pi^0(\gamma(\beta(\alpha(p))), \gamma^{-1}(p'))$, thus the theorem is proved. □

Lemma 1. *Given the shifting environment* Θ, $\Theta^{\#}$ *and* Σ, *and a shift instruction* $\langle \hat{\pi}; l_1, l_2 \rangle$, *if* $\mathsf{shift}(\Theta, \Sigma, \Theta^{\#}, \langle \hat{\pi}; l_1, l_2 \rangle, \bar{s}) = \bar{s}'$, *then* $\bar{s} \doteq \bar{s}'$ *when there is no warning.*

Proof. Induction on the size of the blocks being transformed, as the size of the block decreases each steps we apply the transformation rules.

- Suppose the shifting operation is shifted down to one of the rules 1) Upshift-3, 2) Downshift-1 or 3) Downshift-3. Then the equivalence is direct, as the statements $\bar{s} \equiv \bar{s}'$ in these cases.

- Rule Upshift-0: In this case, the statement $\bar{s} = \texttt{if}(x)\{\bar{s}_{11}\}\texttt{else}\{\bar{s}_{21}\}$. By induction, the two subshifting operation leasto to semantically equivalent statements $\bar{s}'_{11}$ and $\bar{s}'_{21}$ s.t. $\bar{s}_{11} \doteq \bar{s}'_{11}$ and $\bar{s}'_{21} \doteq \bar{s}'_{21}$. Thus $\bar{s}' = \texttt{if}(x)\{\bar{s}'_{11}\}\texttt{else}\{\bar{s}'_{21}\} \doteq \texttt{if}(x)\{\bar{s}_{11}\}\texttt{else}\{\bar{s}_{21}\} = \bar{s}$, and the case holds.

- Rule Upshift-1: The property depends on the shifting function of **downshift**.

- Rule Upshift-2: In this case, $\bar{s} = \texttt{if}(x)\{\bar{s}_1\}\texttt{else}\{\bar{s}_2\}\bar{s}_3\mathsf{chunk}^{\hat{\pi}}[l_2]\bar{s}_4$. The transformation will firstly shift up a sequence of statements $\bar{s}'$ and $\mathsf{chunk}^{\hat{\pi}}[l_2]$ into the if else branches.

  Suppose the first statement to shift is s_x, and the statements between s_x and the end of the if else branch is $\bar{s}_y$, then we can perform switch operation on the statement s_x, as: 1) for any statement s in $\bar{s}_y$, s and s_x has no dependency relation (by the definition of the **closure** function) and 2) after switching, we can perform primitive upshift operation on s_x and move it into the end of the if else branches.

  Then we can step-wisely shift the statements in $\bar{s}'$ and $\mathsf{chunk}^{\hat{\pi}}[l_2]$ up to the if else branch through primitive operations. Thus after shifting $\texttt{if}(x)\{\bar{s}_1 s'\mathsf{chunk}^{\hat{\pi}}[l_2]\}\texttt{else}\{\bar{s}_2 s'\mathsf{chunk}^{\hat{\pi}}\}\bar{s}^3\backslash\bar{s}'\bar{s}_4 \doteq \bar{s}$.

  Then by induction, the shift on the two sub-terms are semantic preserving. So that the property holds for the rule.

- Rule Downshift-2: The proof of the case is similar to that of Upshift-2, except that primitive upshift is changed to primitive downshift.

With these cases proved, we proved that the shift operation on $\bar{s}$ to $\bar{s}'$ is semantically equivalent, so that $\bar{s} \doteq \bar{s}'$. □

Lemma 2. *A generation point* $\mathsf{genPoint}(\hat{\pi}, \overline{ps})$ *is a place where all matched statements* $\bar{ps}_0$ *can be moved together through swap operation.*

Proof. According to the definition of the $\mathsf{genPoint}(\hat{\pi}, \overline{ps})$, the generation point is calculated by intersect of all swappable position of the matched statements ($\bigcap\limits_{ps_i \in S} I_{swap}(ps_i,\ \overline{ps})$).

The definition of I_{swap} is the code lines where the statements can be swapped to according to the dependency relation: $I^{\leftarrow}_{swap}$ is the place of the last dependent statements can be moved to plus one, so that the the statements between them are independent thus can be swapped with the swapped operation. Similarly, the $I^{\rightarrow}_{swap}$ operation calculate the last position of the statement can be swapped to by counting its nearest following statement's position.

As it works for each statement, the intersect operation ensures that all of the statements can be shifted to the corresponding position. □