Constructing Structured SSA from FJ

Kenny Zhuo Ming Lu
Information Systems Technology and Design
Singapore University of Technology and Design
Singapore
kenny_lu@sutd.edu.sg

Daniel Yu Hian Low
Information Systems Technology and Design
Singapore University of Technology and Design
Singapore
daniel_low@mymail.sutd.edu.sg

ABSTRACT

We propose a novel approach of constructing structured SSA forms. Specifically, our declarative algorithm converts a Featherweight Java (FJ) program into its structured SSA form in a single pass. We prove that the proposed algorithm produces valid SSA forms which are semantically consistent with respect to the original source programs. We verify the resulting SSA forms are minimal. We demonstrate that structured SSA form can serve as a unified intermediate representation for both compiler optimization and program verification pipelines. We implemented the algorithm as a library.

CCS CONCEPTS

• **Theory of computation** → **Program semantics**.

KEYWORDS

Static single assignment, Program transformation, SSA construction

1 INTRODUCTION

Static Single Assignment (SSA) form is an intermediate representation for main-stream modern compiler implementations. Within a program in SSA form, variables are restricted to be assigned once. Reference to variables that are dependent on the control flow (e.g. an if-else statement) requires a merge via phi assignments. For instance, in Figure 1, we find the original program in sub figure (a) and the correspondent SSA form in sub figure (b). Note that N denotes some input argument of the program. Making the use-def relation explicit on the syntax level helps to simplify many optimization algorithms for compilation. In the programming verification context, the use of SSA form avoids the exponential explosion of verification conditions [7, 10, 11]. We note that from literatures, most of the SSA construction algorithms to date [5, 6, 9, 13, 15], produce results in some unstructured intermediate representations. In these unstructured representations, language constructs such as loops are compiled into labels and conditional jump operations, for instance,

```
        s = 0;
        i = 0;
        while ( i < N){
          s = s + i;
          i = i + 1;
        }
        return s;
```

(a) Original Program

```
0:  s0 = 0;
    i0 = 0;
1:  i1 = phi(0:i0,2:i2);
    s1 = phi(0:s0,2:s2);
    if (i1 >= N) goto 3;
2:  s2 = s1 + i1;
    i2 = i1 + 1;
    goto 1;
3:  return s1;
```

(b) Unstructured SSA

```
0:  s0 = 0;
    i0 = 0;
1:  join {i1=phi(0:i0,2:i2);
          s1=phi(0:s0,2:s2)} while (i1 < N){
2:      s2 = s1 + i1;
        i2 = i1 + 1;
    }
3:  return s1;
```

(c) Structured SSA

Figure 1: Unstructured SSA vs Structured SSA (type declaration omitted)

Figure 1 (b). Unstructured SSA form was originally designed for compiler optimization, it is not optimized for program verification. For example, one has to transform the source program into an intermediate form in Guarded Command Language by unrolling loops with a fixed number of iterations. Then the unrolled GCL has to undergo the passification operation to eliminate re-assignments, [7, 10, 11]. The passification transformation, which is actually a unstructured SSA construction, has to be performed after unrolling so that the loop structure and invariance are retained in GCL. The pipeline setup with loop unrolling followed by SSA construction is inefficient in the event of changing unrolling configurations. Note that the time complexity of SSA construction is dependent on the size of the source program, and the time complexity of the unrolling operation is dependent on the number loops found in the program. SSA construction does not change the number of loops, while applying unrolling increases the size of the program. Hence applying SSA construction before the unrolling allows us to construct the SSA once and use it for different unrolling configurations since the number of loops remains unchanged, whereas we need to recompute both unrolling and SSA if we apply unrolling first then SSA construction. The use of unstructured SSA prohibits the pipeline setup with SSA construction being applied first.

The structured SSA form, introduced by Ancona and Corradi [1], can be defined by extending the source language with phi

```
1   public int max() {
2     int res = this.arr[0]; int i = 1;
3     while (i < this.arr.length) {
4       if (res < this.arr[i]) {
5         res = this.arr[i];
6       }
7       i = i + 1;
8     }
9     return res;
10  }
```

Figure 2: A running example, a max method in a class finds the max value in the int array member of the class.

assignments embedded in the control flow statements, for instance, Figure 1 (c). As a result, language constructs such as loops are retained in the SSA form (as well as the loop invariants). This addresses the above-mentioned limitation with unstructured SSA. For instance, given that loop invariant s1 <=i1*(i1-1)/2, the program in Figure 1 (c) can be transformed into the following GCL form by unrolling the while loop once.

$$(\ s_0 = 0; \ i_0 = 0;$$
$$(i_1^1 = i_0; s_1^1 = s_0; assume \ (i_1^1 < N) \land s_1^1 <= i_1^1 * (i_1^1 - 1)/2;$$
$$s_2^1 = s_1^1 + i_1^1; i_2^1 = i_1^1 + 1); assert \ false; \ )$$
$$\square$$
$$(i_1 = i_0; s_1 = s_0; assume \ (i_1 >= N) \land s_1 <= i_1 * (i_1 - 1)/2;)$$

We argue that the use of structured SSA form offers better flexibility in program compilation and analysis pipelines such that the same structured SSA form can be re-used for multiple purposes.

In [1], Anconna and Corradi applied structured SSA in type checking and analysis of Java programs, however there was no construction algorithm given. In this paper, we propose a single-pass construction algorithm which converts an FJ-like program into its structured SSA form. We are interested in formalizing the algorithm in declarative style using term rewriting, from which the correctness and minimality results can be verified. Our key contributions are as follows.

(1) We formalize a declarative single-pass term-rewriting algorithm for constructing structured SSA forms from FJ.
(2) We show that the algorithm produces valid and consistent SSA forms.
(3) We prove that the resulting SSA forms are minimal.

The rest of this paper is organized as follows. In Section 2, we informally explain the proposed approach via an example. In Section 3 we formalize syntax and semantics of FJ and structured SSA languages. In Section 4, we present the SSA construction algorithm and the formal results. In Section 5 we discuss the related works and we report the implementation of the algorithm and conclude in Section 6.

2 A RUNNING EXAMPLE

Consider an FJ class contains a non-empty int array and a max method as members, in Figure 2. For brevity we omit the class declaration and the constructor methods. We focus on the max method which returns the maximum value in in the array this.arr.

To convert the method max into an SSA form, we traverse its body structurally and convert statements into SSA statement blocks. An SSA statement block is a sequence of statements associated with a label. At the outer most level, max's body consists of three sections, namely the variables declarations and initialization at line 2, the while statement at lines 3-8 and the return statement at line 9. For simplicity, we assume all local variables are declared at the start of the method body.

Line 2. The conversion of these declarations is trivial, since none contains references of previously assigned variables. We convert them into an SSA statement block.

```
0: int res0=this.arr[0]; int i0=1;
```

where 0 denotes a label associated with the SSA statement block, which could be used for phi assignment resolution in some later part. For each assignment statement found in the FJ program, we rename the variables on the left hand side by making use of the label. Assuming labels are unique within a method, renamed variables appearing on the left hand side of assignments in the SSA form will be unique. Since a (re)definition of a variable is introduced here, we record the information in a look-up table, namely the *variable mapping* table. Each entry of the variable mapping table contains a variable in the FJ program and its renamed variable in the SSA form, the location of the assignment statement in the FJ program as well as the location in the SSA form. We use program contexts to refer to statement locations in the FJ program and its SSA form. For example, given the max method definition in Figure 2, we use

- $\square; \bar{s}$ to refer to the first statement, int res=this.arr[0];
- $s; \square; \bar{s}$ to refer to the second statement, int i=1; and
- $s; s; while \ E\{if \ E \ \{\square\}; \bar{s}\}; \bar{s}$ to refer to the assignment statement at line 5.

Where $\square$ denotes a hole in a program structure. s symbolically denotes a statement, and $\bar{s}$ symbolically denotes a sequence of statements. The program contexts for the SSA form work in a similar way, except that instead of s, we have B denoting a statement block symbolically. A formal definition will be given in later parts of this paper. Note that the program contexts introduced here should not be confused with the standard evaluation context. The former serve as indices to AST identifying location where variable redefinition occurs, the latter specify which sub-expression in a program can be reduced. After converting the above statements, we have the following entries in the variable mapping table,

$$I: \quad (\text{res}, \quad \square; \bar{s}, \quad \square; \bar{B}, \quad \text{res0})$$
$$II: \quad (\text{i}, \quad s; \square; \bar{s}, \quad \square; \bar{B}, \quad \text{i0})$$

For ease of reference, we add one roman number ID for each entry.

In the above code fragment, the right hand sides of the assignment statements are constants, thus no conversion is required. In case that there exist references of previously assigned variables, we need to replace these variable references by the most recent redefinitions from the variable mapping table. We will give a detailed explanation when we encounter this situation shortly.

Lines 3-8. We convert a while statement. It requires the use of the following while statement in the SSA form.

```
join { /* phi assignments*/ }
while E { /* the converted body */ }
```

Firstly, we consider the set of phi assignments in the while statement block. It merges variable definitions coming from the preceding statements and the while loop body. For instance, we need to identify the operands to be placed in $i1 = phi(\cdot, \cdot)$. To determine the operands of the phi assignments, we introduce a relation among two SSA program contexts $\text{CTX}_d \leq \text{CTX}_r$. Intuitively speaking, the relation states that variables defined in CTX_d might be redefined in CTX_r. Given that current SSAFJ context is $\text{CTX}_1 = \mathbf{B}; \square; \overline{\mathbf{B}}$, we search in the variable mapping table to look for all the contexts CTX such that $\text{CTX} \leq \text{CTX}_1$. We found only one candidate $\text{CTX} = \square; \overline{\mathbf{B}}$. In the event that there are multiple candidates found, we assume that $\leq$ relation is a partial order relation w.r.t. the available contexts. The set of program contexts and $\leq$ forms a semi-join lattice, hence an upper bound exists. We look up the corresponding renamed variables w.r.t. this upper bound, we find $i1 = phi(0{:}i0, \cdot)$ and $res1 = phi(0{:}res0, \cdot)$. We extend the variable mapping table with two new entries as follows,

$$
\begin{array}{llll}
III: & (\text{res}, & \mathbf{s}; \mathbf{s}; \square; \overline{\mathbf{s}}, & \text{CTX}_2, & \text{res1}) \\
IV: & (\text{i}, & \mathbf{s}; \mathbf{s}; \square; \overline{\mathbf{s}}, & \text{CTX}_2, & \text{i1})
\end{array}
$$

where $\text{CTX}_2 = \text{CTX}_1[\text{join } \{\blacksquare\} \text{while } \mathbf{E}\{\overline{\mathbf{B}}\}]$. Let ctx and ctx$'$ denote FJ program contexts, we write ctx[ctx$'$] to refer to a program context obtained by substituting the $\square$ in ctx with ctx$'$. Similar idea applies to $\text{CTX}[\text{CTX}']$. $\blacksquare$ is similar to $\square$ except that it can't be substituted. To resolve the second operands of the phi assignment, we first convert the body of the while loop. We face an issue. The program context referring to the body of the while loop $\text{CTX}_1[\text{join } \{\overline{\phi}\} \text{while } \mathbf{E}\{\square\}]$ satisfies the following cyclical relation.

$$\text{CTX}_2 < \text{CTX}_1[\text{join } \{\overline{\phi}\} \text{while } \mathbf{E}\{\square\}] < \text{CTX}_2$$

To address this issue, we need to break the loop, otherwise the set of program contexts is no longer a poset. A key observation is that the first $<$ predicate is only needed during the conversion of the while loop body, and the second one is needed during the conversion of the statements that following the while loop. The trick is to set a versioning bit to the context $\text{join}\{\blacksquare^b\}\text{while } \mathbf{E}\{\overline{\mathbf{B}}\}$ where b can be either 0 or 1. Let $\text{CTX}_2^{0,1} = \text{CTX}_1[\text{join } \{\blacksquare^{0,1}\}\text{while } \mathbf{E}\{\overline{\mathbf{B}}\}]$. The $<$ relation is then adjusted to respect the bit b as follows

$$\text{CTX}_2^0 < \text{CTX}_1[\text{join } \{\overline{\phi}\}\text{while } \mathbf{E}\{\square\}]$$
$$\text{CTX}_1[\text{join } \{\overline{\phi}\}\text{while } \mathbf{E}\{\square\}] < \text{CTX}_2^1$$

When converting the while loop body, we only consider the first relation. We move on to convert the conditional expression of the while loop, (i < this.arr.length). By looking up the variable mapping table, we find that the most recent redefinition of i w.r.t to the current context is CTX_2, hence (i1 < this.arr.length) is the result expression in SSAFJ. Next, we consider the nested if-else statement at lines 4-6. For simplicity, we assume that an empty else branch is added to the if-statement that has no else branch. To convert if-else statements. we make use of the following if-else statement in the SSA form.

```
if E { /*the converted then-branch*/ }
else { /*the converted else-branch*/ }
join { /*phi assignments*/ }
```

We apply the conversion to condition expression, the then-branch and else-branch recursively and insert the results into the stub code above. The phi assignments at the end of the if-else statement merge variable definitions coming from the then- and the else- branches. We convert the conditional expression of the if-else, namely res < this.arr[i] into res1 < this.arr[i1].

- The then-branch at line 5 contains a single assignment statement. We apply the same treatment to convert this assignment statement into the following SSA form.

  ```
  3: res3 = this.arr[i1];
  ```
  A new entry is added to the variable mapping table

  $$V: \quad (\text{res}, \quad \text{ctx}_3, \quad \text{CTX}_3, \quad \text{res3})$$

  where
  – $\text{ctx}_3 = \mathbf{s}; \mathbf{s}; \text{while } \mathbf{e}\{\text{if } \mathbf{e} \{\square; \}\text{else}\{\overline{\mathbf{s}}\}\}; \overline{\mathbf{s}}$
  – $\text{CTX}_3 =$
    $\text{CTX}_1[\text{join } \{\overline{\phi}\}\text{while } \mathbf{E}\{\text{if } \mathbf{E} \{\square; \}\text{else}\{\overline{\mathbf{B}}\}\text{join}\{\overline{\phi}\}; \overline{\mathbf{B}}\}]$
- The empty else-branch added via the desugarer contains no statements.

We consider the exit point of the if-else statement, at which we need to insert another set of phi assignments to merge variable (re)definitions arising from the then- and else-branches. We examine the subset of entries from the variable mapping tables generated from the then branch, namely I-V and the subset from the else branch, namely I-IV. For variable res the most recent definition from the then branch is CTX_3, and the most recent definition from the else branch is CTX_2. Thus, the phi assignment at the exit of the if-else statement contains the following

```
res2 = phi(3: res3, 4: res2)
```

where 4 is the label of the else branch. Note that variable i is modified in neither branches, which can be verified by checking against those entries having contexts referring to the then- and else-branch. We do not include another phi assignment for i to achieve minimality. We add the following entry to the variable mapping table

$$VI: \quad (\text{res}, \quad \text{ctx}_4, \quad \text{CTX}_4, \quad \text{res}_2)$$

where

- $\text{ctx}_4 = \mathbf{s}; \mathbf{s}; \text{while } \mathbf{e}\{\square; \overline{\mathbf{s}}$
- $\text{CTX}_4 = \text{CTX}_1[\text{join } \{\overline{\phi}\}\text{while } \mathbf{E}\{\text{if } \mathbf{E} \{\overline{\mathbf{B}}; \}\text{else}\{\overline{\mathbf{B}}\}\text{join}\{\blacksquare\}; \overline{\mathbf{B}}\}]$

The statement at line 7 is an assignment. The RHS is i can be replaced by by looking up the most recent definition of i in from variable mapping table, which is CTX_2.

```
5: i2 = i1+ 1
```

We add the following entry to the variable mapping table

$$VII: \quad (\text{i}, \quad \text{ctx}_5, \quad \text{CTX}_5, \quad \text{i}_2)$$

where

- $\text{ctx}_5 = \mathbf{s}; \mathbf{s}; \text{while } \mathbf{e}\{\mathbf{s}; \square; \}$
- $\text{CTX}_5 = \text{CTX}_1[\text{join } \{\overline{\phi}\}\text{while } \mathbf{E}\{\mathbf{B}; \square; \}]$

We have completed the conversion of the while loop body. It is time to find resolve the second operands in $i1 = phi(0{:}i0, \cdot)$ and $res1 = phi(0{:}res0, \cdot)$.

By looking up the variable mapping table with CTX_2^0 replaced by CTX_2^1, we find that the most recent redefinition of i is i2 at CTX_5 and the most most recent redefinition of res is res2 at CTX_4. We collect the renamed variables and declare them at the first block in

```
1   public int max() {
2   0:int res0, res1, res2, res3;
3       in i0, i1, i2;
4       res0 = this.arr[0];
5       i0 = 1;
6   1:join {res1 = phi(0:res0, 5:res2),
7           i1 = phi(0:i0, 5:i2)}
8       while (i1 < this.arr.length) {
9       2:if (res1 < this.arr[i1]) {
10          3: res3 = this.arr[i1];
11          } else { 4: nop;
12          } join {res2 = phi(3:res3, 4:res1)}
13          5: i2 = i1 + 1;
14      }
15      6:return res1;
16  }
```

Figure 3: Method max in SSA form

the SSA form. Putting all these together, we find the SSA form of the max method in Figure 3.

3 FJ AND SSAFJ

In this section, we consider the formal definitions of FJ. In Figure 4, we report the syntax and semantics of FJ in the left column. We omitted the details for class declaration and construction methods. For brevity we skip the details of the treatment of sub-typing. We restrict ourselves to unary methods only. Local variable declarations must be defined at the beginning of a method. Besides these, we assume the following non-syntactical restriction. All methods end with one and only one return statement.

In the same figure, we report syntax of SSAFJ in the right column. The only part that differs from FJ counterpart is the statement block B. For simplicity we assume that each statement block contains only one statement. The assignment A and expressions E of SSAFJ are identical to the ones found in FJ. A minor difference from the example in the earlier section, instead of using integers as labels, we use SSAFJ program contexts. We assume that the SSAFJ context annotating a statement block is the exact location of the block in the program. We ignore the bit associated with the phi-assignment context in the while block for now since it only becomes relevant when we consider the SSA construction algorithm.

The semantics of FJ can be found in literatures hence omitted. The semantics of SSAFJ is similar to the one presented in [1], with the following minor presentation difference. In case of if-else and while statements, we merge different (re)definitions of the same (FJ) variable using the phi assignments. For each phi assignment $x =$ phi($\mathsf{CTX}_1 : x_1, ..., \mathsf{CTX}_n : x_n$), the semantic function $\mathbb{F}_{\mathsf{ssa}}[\![\cdot]\!]$ selects the variable x_i based on the matching of label CTX_i and updates the local environment with $(x, lenv(x_i))$. Given a set of pairs S, we write $S + (x, v)$ to denote $\{(y, v') \mid (y, v') \in S \land y \neq x\} \cup \{(x, v)\}$. We use Haskell's style list type $[\mathtt{I}]$ to denote a sequence of items $\mathtt{I}$. $[\,]$ denotes an empty sequence and $i; \bar{i}$ denotes a sequence with i as the head and $\bar{i}$ as the tail. In [1], the phi-assignments are resolved with a stack that keeps track of the most recent assignments.

4 SSA CONSTRUCTION

In this section, we consider the formal details of the SSA construction algorithm. In Figure 5 we define the conversion from FJ program contexts to SSAFJ program contexts and the $<$ relation among two SSAFJ contexts. In the same figure, we define the variable mapping as a quadruple consisting of a variable and its contexts in FJ and the correspondent variable and context in SSAFJ. We write (a, b, c, d) as short hand for $(((a, b), c), d)$. We define $dom(S) = \{a \mid (a, b) \in S\}$, $dom2(S) = dom(dom(S))$ and $dom3(S) = dom(dom2(S))$. Let S denote a set of contexts, we write $\mathsf{CTX}[S]$ to refer to $\{\mathsf{CTX}[\mathsf{CTX}'] \mid \mathsf{CTX}' \in S\}$. The auxiliary function $\mathbb{R}_{\leq\mathsf{CTX}}(\beta, x)$ finds the most recent (re)-definition of variable x w.r.t the current location CTX.

The SSA construction algorithm is described in terms of a set of conversion rules. $\beta, \mathsf{CTX} \vdash e \Rightarrow E$ converts an expression, $\vdash md \Rightarrow MD$ converts a method declaration. $\beta \vdash \overline{vd} \Rightarrow \overline{VD}$ converts a sequence of variable declarations. $\mathsf{ctx}, \beta, \mathsf{CTX} \vdash \bar{s} \Rightarrow \bar{B}, \beta, \mathsf{CTX}$ and $\mathsf{ctx}, \beta, \mathsf{CTX} \vdash s \Rightarrow B, \beta, \mathsf{CTX}$ convert FJ statement(s) into SSAFJ statement block(s). Let's consider the input to these rules. ctx refers to the current FJ program context, β denotes the variable mapping table obtained from the previous step and CTX refers to the SSAFJ program context processed in the previous step. We highlight the important cases.

Case $x = e$. We convert the FJ context to an SSAFJ context with the auxiliary function $[\![\cdot]\!]$. Given the SSAFJ context, we apply expression conversion rules to convert the FJ expression on the right hand side of the assignment statement. In case of a variable reference is found in the right hand side expression, we apply the auxiliary function $\mathbb{R}_{\leq\mathsf{CTX}}(\beta, x)$ to lookup the latest (re)definition of x in β w.r.t. the context CTX. The details of the function is presented in Figure 5. The main idea is based on the fact that SSAFJ contexts recorded in the variable mapping table form a semi-join lattice over the $\leq$ relation. The most recent redefinition is found at the least upper bound among these contexts. Function $lub_{(x,\_)}(S)$ returns a pair whose first component is the least upper bound among all pairs from S. With the label l (the SSAFJ context), we rename the left hand side variable. Note that the SSAFJ contexts are unique within a method, so are the renamed variables. We extend the variable mapping β with the left hand side FJ variable, FJ and SSAFJ contexts and the SSAFJ variable. Finally, we return an SSAFJ block with the updated state.

Case if e $\{\overline{s_1}\}$else$\{\overline{s_2}\}$. We first convert the conditional expression e. Then we apply the statement conversion rules to convert the then-branch and else-branch statements. Next we construct the phi assignments at the exit point of the if-else statement at which common variables from the branches are merged. For each updated variable x found in the domain of $(\beta_1 \cup \beta_2) - \beta_0$, we define a phi assignment by associating label CTX_1 with the most recent definition of x at CTX_1 and associating label CTX_2 with the most recent definition of x at CTX_2, where CTX_1 and CTX_2 the exiting contexts from the branches.

Case while e $\{\bar{s}\}$. Before converting the loop body recursively, we construct the phi assignment at the entry point of the while loop. To do so, we need to consider the context from the preceding block CTX_0. For each variable x in the domain of the incoming β_0, we construct a phi assignment of which the first operand has the

(MethDecl)	md	::=	$t\ m\ (t\ x)\ \{\overline{vd};\overline{s}\}$
(VarDecl)	vd	::=	$t\ x$
(Statement)	s	::=	$a\ \|\ \text{return}\ e\ \|\ x = e.m(e)$
			$\|\ \text{while}\ e\ \{\overline{s}\}\ \|\ \text{if}\ e\ \{\overline{s}\}\ \text{else}\ \{\overline{s}\}$
(Assignment)	a	::=	$x = e\|e.f = e$
(Expression)	e	::=	$v\ \|\ x\ \|\ e.f\ \|\ \text{this}\ \|\ e\ op\ e$
(Operator)	op	::=	$+\ \|\ -\ \|>\|<\|==\|\ ...$
(Var)	x	::=	$x\ \|\ y\ \|\ ...$
(Type)	t	::=	$int\ \|\ bool\ \|\ void\ \|\ C$
(Context)	ctx	::=	$\square\ \|\ ctx;\ \|\ ctx;\overline{s}\ \|\ s;ctx\ \|\ \text{if}\ e\ \{ctx\}\ \text{else}\ \{\overline{s}\}$
			$\|\ \text{if}\ e\ \{\overline{s}\}\ \text{else}\ \{ctx\}\ \|\ \text{while}\ e\ \{ctx\}$
(Local Decls)	LEnv	$\subseteq$	$(\text{Variable} \times \text{Value})$

(METHDECL)	MD	::=	$t\ m\ (t\ x)\ \{\overline{VD};\overline{B}\}$
(VARDECL)	VD	::=	$t\ x$
(BLOCK)	B	::=	$l : \{S\}$
(STATEMENT)	S	::=	$A\ \|\ \text{return}\ E\ \|\ \text{join}\ \{\overline{\phi}\}\ \text{while}\ E\ \{\overline{B}\}\ \|$
			$\|\ x = E.m(E)\ \|\ \text{if}\ E\ \{\overline{B}\}\ \text{else}\ \{\overline{B}\}\ \text{join}\ \{\overline{\phi}\}$
(PHI)	ϕ	::=	$x = \text{phi}(\overline{l : x})$
(LABEL)	l	::=	CTX
(CONTEXT)	CTX	::=	$\square\ \|\ \text{CTX};\ \|\ \text{CTX};\overline{B}\ \|\ B;\text{CTX}\ \|$
			$\text{if}\ E\ \{\text{CTX}\}\ \text{else}\ \{\overline{B}\}\ \text{join}\ \{\overline{\phi}\}$
			$\|\ \text{if}\ E\ \{\overline{B}\}\ \text{else}\ \{\text{CTX}\}\ \text{join}\ \{\overline{\phi}\}$
			$\|\ \text{if}\ E\ \{\overline{B}\}\ \text{else}\ \{\overline{B}\}\ \text{join}\{\blacksquare\}$
			$\|\ \text{join}\ \{\blacksquare^b\}\ \text{while}\ E\ \{\overline{B}\}$
			$\|\ \text{join}\ \{\overline{\phi}\}\ \text{while}\ E\ \{\text{CTX}\}$

$$\text{CTX}; lenv \vdash [\ ] \longrightarrow lenv \quad (\text{EmpPhiRes})$$

$$\frac{\text{CTX}; lenv \vdash \overline{\phi} \longrightarrow lenv'}{\text{CTX}_i; lenv \vdash x = \text{phi}(\text{CTX}_1 : x_1,,...,\text{CTX}_i : x_i,...,\text{CTX}_n : x_n);\overline{\phi} \longrightarrow lenv' + (x, lenv(x_i))} \quad (\text{PhiRes})$$

Figure 4: Syntax and Semantics of FJ and SSAFJ (excerpt)

label CTX_0 and the expression $\mathbb{R}_{\leq \text{CTX}_0}(\beta_0, x)$. Note that at this point, the phi assignment $\overline{\phi}$ is *incomplete* (because the second operands are missing) and *over-approximated* (because not all variables need to be merged). We update it with $\overline{\phi'}$ based on the returned state from converting the loop body. Given the exiting context from the loop body CTX_2, for each updated variable x in $dom3(\beta_2 - \beta_1)$, we *update* the second operand to the existing to phi function, with CTX_2 as the label and $\mathbb{R}_{\leq \text{CTX}_2}(\beta_2, x)$ as the expression. Now we need to eliminate those redundant phi assignment introduced in $\overline{\phi}$, but not in $\overline{\phi'}$, i.e. those of which the merged variables were not updated in the while loop body. This is achieved by building a substitution θ and applying it to E and $\overline{B}$. A substitution is a mapping from variable to variable. Lastly, we update the versioning bit of all the occurrences of $\text{CTX}[\text{join}\ \{\blacksquare^0\}\text{while}E\{\overline{B}\}]$ in the resulting variable mapping table to 1.

We say two method declarations are consistent if given the same class environment, memory environment and input values, both method should yield the same result value and memory environment if they terminate.

LEMMA 4.1 (SSA CONVERSION CONSISTENCY). *Let md be an FJ method declaration and MD be a SSAFJ method declaration such that $\vdash md \Rightarrow MD$. Then md and MD are consistent.*

We say an SSAFJ MD is minimal if there exist no SSAFJ method obtained by reducing the number of phi-assignments from MD, which produces the same result.

LEMMA 4.2 (SSA CONVERSION MINIMALITY). *Let md be an FJ method declaration and MD be a SSAFJ method declaration such that $\vdash md \Rightarrow MD$. Then MD is minimal.*

4.1 Time Complexity and Code Size of the SSA Construction Algorithm

Our structured SSA conversion algorithm has the worst-case time complexity $O(n^3)$ where n is the size of the program, since each call to $\mathbb{R}_{\leq}.(\cdot,\cdot,\cdot)$ is $O(n^2)$. If we table all the calls of $\mathbb{R}_{\leq}.(\cdot,\cdot,\cdot)$ via memoization, the worst case time complexity is reduced to $O(n^2)$.

Let V denote the number of variables. The total time complexity is $O(n^2 \cdot V)$. The size of the resulting SSA form is bounded by $n \cdot V$.

5 RELATED WORKS

Cytron et al's SSA algorithm [9] finds the minimal set of phi assignments to be inserted in a CFG by constructing the dominance frontiers. Aycock and Horspool proposed a two-phase approach in constructing SSA [2]. During the first phase, the algorithm "crudely" inserts phi assignments at every basic blocks; in the second phase, some of the redundant or trivial phi assignments are removed to achieve minimality. In their work [5], Braun et al proposed an SSA construction algorithm by traversing the AST backward and recursively. In a later work [8]. Buchwald et al considered a variant of Braun's SSA construction algorithm and verified that the algorithm is correct w.r.t. the program semantics and produces minimal SSA. Brandis and Mössenböck proposed a single-pass SSA construction algorithm for structured programming languages without goto statements [4]. The SSA produced by their algorithm is in a form of structured control flow graph. In his thesis [14], Pop studied the formalism of a high-level SSA language without goto statement. He formulated the conversion from IMP to SSA and proved that the result is consistent. In [3], Lourenço et al introduced a single assignment form of the While program in which loops and loop invariants are retained for verification condition generation and structured reasoning purposes. Their SA form contains syntactic controlled violation of single assignment requirement. It remains unknown to us whether their SA form can be used for compilation optimization purposes.

6 IMPLEMENTATION AND CONCLUSION

We proposed an algorithm which constructs structured SSA forms from FJ. We proved that the produced SSA form is consistent and minimal. We showed that structured SSA forms can be used for compiler optimization as well as program verification optimization. We implemented the construction algorithm in a library using Scala [12]. Detail proofs of the formal results can be found in the Appendix.

$$\llbracket \cdot \rrbracket \quad :: \quad \text{Context} \to \text{CONTEXT}$$
$$\llbracket \square \rrbracket \quad = \quad \square$$
$$\llbracket ctx; \rrbracket \quad = \quad \llbracket ctx \rrbracket;$$
$$\llbracket ctx; \overline{s} \rrbracket \quad = \quad \llbracket ctx \rrbracket; \overline{\mathbf{B}}$$
$$\llbracket s; ctx \rrbracket \quad = \quad \mathbf{B}; \llbracket ctx \rrbracket$$
$$\llbracket \text{if } e \ \{\overline{s}\} \text{ else } \{\overline{s}\} \rrbracket \quad = \quad \text{if } \mathbf{E} \ \{\llbracket ctx \rrbracket\} \text{ else } \{\overline{\mathbf{B}}\} \text{ join } \{\overline{\phi}\}$$
$$\llbracket \text{if } e \ \{\overline{s}\} \text{ else } \{ctx\} \rrbracket \quad = \quad \text{if } \mathbf{E} \ \{\overline{\mathbf{B}}\} \text{ else } \{\llbracket ctx \rrbracket\} \text{ join } \{\overline{\phi}\}$$
$$\llbracket \text{while } e \ \{ctx\} \rrbracket \quad = \quad \text{join } \{\overline{\phi}\} \text{ while } \mathbf{E} \ \{\llbracket ctx \rrbracket\}$$

(Variable Mapping) $\quad \beta \quad \subseteq \quad (\text{Var} \times \text{ctx} \times \text{CTX} \times \text{VAR})$

$\boxed{Last(\text{CTX})}$

$$Last(\square;) \quad Last(\text{if } \mathbf{E} \ \{\overline{\mathbf{B}}\} \text{else} \{\overline{\mathbf{B}}\} \text{join} \{\blacksquare\};)$$

$$\frac{Last(\text{CTX})}{Last(\mathbf{B}; \text{CTX})} \qquad Last(\text{join} \{\blacksquare\} \text{while } \mathbf{E} \ \{\overline{\mathbf{B}}\};)$$

$\boxed{\text{CTX} < \text{CTX}'}$

$$\frac{\square \neq \text{CTX}}{\square < \text{CTX}} \text{ (CtxHole)} \quad \frac{\text{CTX}_1 < \text{CTX}_2}{\text{CTX}[\text{CTX}_1] < \text{CTX}[\text{CTX}_2]} \text{ (CtxInd)} \quad \text{CTX}_1; \overline{\mathbf{B}} < \mathbf{B}; \text{CTX}_2 \text{ (CtxSeq)} \quad \frac{\text{CTX}_1 < \text{CTX}_2 \quad \text{CTX}_2 < \text{CTX}_3}{\text{CTX}_1 < \text{CTX}_3} \text{ (CtxTrans)}$$

$$\text{join } \{\blacksquare^0\} \text{ while } \mathbf{E} \ \{\overline{\mathbf{B}}\} < \text{join } \{\overline{\phi}\} \text{ while } \mathbf{E} \ \{\square\} \text{ (CtxWhile1)} \qquad \frac{Last(\text{CTX})}{\text{join } \{\overline{\phi}\} \text{ while } \mathbf{E} \ \{\text{CTX}\} < \text{join } \{\blacksquare^1\} \text{ while } \mathbf{E} \ \{\overline{\mathbf{B}}\}} \text{ (CtxWhile2)}$$

$$\frac{Last(\text{CTX})}{\text{if } \mathbf{E} \ \{\text{CTX}\} \text{ else } \{\overline{\mathbf{B}}\} \text{ join } \{\overline{\phi}\} < \text{if } \mathbf{E} \ \{\overline{\mathbf{B}}\} \text{ else } \{\overline{\mathbf{B}}\} \text{ join } \{\blacksquare\}} \text{ (CtxThen)} \qquad \frac{Last(\text{CTX})}{\text{if } \mathbf{E} \ \{\overline{\mathbf{B}}\} \text{ else } \{\text{CTX}\} \text{ join } \{\overline{\phi}\} < \text{if } \mathbf{E} \ \{\overline{\mathbf{B}}\} \text{ else } \{\overline{\mathbf{B}}\} \text{ join } \{\blacksquare\}} \text{ (CtxElse)}$$

$\boxed{\mathbb{R}_{\leq \text{CTX}}(\beta, x, d)}$

$$\mathbb{R}_{\leq \text{CTX}}(\beta, x, d) = snd(lub_{(x,\_)}(\{(\text{CTX}', x') \mid (x, \text{ctx}, \text{CTX}', x') \in \beta \wedge \text{CTX}' \leq \text{CTX}\}, d)) \qquad \mathbb{R}_{\leq \text{CTX}}(\beta, x) = \mathbb{R}_{\leq \text{CTX}}(\beta, x, null) \qquad snd((x, y)) = y$$

$\boxed{\beta, \text{CTX} \vdash e \Rightarrow E}$

$$\beta, \text{CTX} \vdash v \Rightarrow v \text{ (ValEConv)}$$
$$\beta, \text{CTX} \vdash x \Rightarrow \mathbb{R}_{\leq \text{CTX}}(\beta, x) \text{ (VarEConv)} \qquad \frac{\beta, \text{CTX} \vdash e_1 \Rightarrow E_1 \quad \beta, \text{CTX} \vdash e_2 \Rightarrow E_2}{\beta, \text{CTX} \vdash e_1 \ op \ e_2 \Rightarrow E_1 \ op \ E_2} \text{ (OpEConv)} \qquad \frac{\beta, \text{CTX} \vdash e \Rightarrow E}{\beta, \text{CTX} \vdash e.f \Rightarrow E.f} \text{ (FieldEConv)}$$
$$\beta, \text{CTX} \vdash this \Rightarrow this \text{ (ThisEConv)}$$

$\boxed{\vdash md \Rightarrow MD}$

$$\frac{\square, [(x, \square, \square, x)], \square \vdash \overline{s} \Rightarrow \overline{B}, \beta, \text{CTX} \quad \beta \vdash \overline{vd} \Rightarrow \overline{VD}}{\vdash t \ m \ (t' \ x) \ \{\overline{vd}; \overline{s}\} \Rightarrow t \ m \ (t' \ x)\{\overline{VD}; \overline{B}\}} \qquad \beta \vdash [\ ] \Rightarrow [\ ] \text{ (EmpVDConv)} \qquad \boxed{\beta \vdash \overline{vd} \Rightarrow \overline{VD}}$$

$$\frac{\beta \vdash \overline{vd} \Rightarrow \overline{VD} \quad \overline{VD'} = \{(t \ x') \mid (x, \_, \_, x') \in \beta\}}{\beta \vdash t \ x; \overline{vd} \Rightarrow t \ x; \overline{VD'}; \overline{VD}} \text{ (SeqVDConv)}$$

$\boxed{\text{ctx}, \beta, \text{CTX} \vdash \overline{s} \Rightarrow \overline{B}, \beta, \text{CTX}}$

$$\frac{\text{ctx}[\square;], \beta, \text{CTX} \vdash s \Rightarrow B, \beta', \text{CTX}'}{\text{ctx}, \beta, \text{CTX} \vdash s; \Rightarrow B;, \beta', \text{CTX}'} \text{ (LastStmtSeqConv)} \qquad \frac{\text{ctx}[\square; \overline{s}], \beta, \text{CTX} \vdash S \Rightarrow B, \beta', \text{CTX}' \quad \text{ctx}[s; \square], \beta', \text{CTX}' \vdash \overline{s} \Rightarrow \overline{B}, \beta'', \text{CTX}''}{\text{ctx}, \beta, \text{CTX} \vdash s; \overline{s} \Rightarrow B; \overline{B}, \beta'', \text{CTX}''} \text{ (StmtSeqConv)}$$

$\boxed{\text{ctx}, \beta, \text{CTX} \vdash s \Rightarrow B, \beta, \text{CTX}}$

$$\frac{\text{CTX}' = \llbracket ctx \rrbracket \quad \beta, \text{CTX} \vdash e \Rightarrow E \quad \beta' = \beta \cup \{(x, \text{ctx}, \text{CTX}', x_{\text{CTX}'})\}}{\text{ctx}, \beta, \text{CTX} \vdash x = e \Rightarrow \text{CTX}' : x_{\text{CTX}'} = E, \beta', \text{CTX}'} \text{ (AsmStmtConv)} \qquad \frac{\text{CTX}' = \llbracket ctx \rrbracket \quad \beta, \text{CTX} \vdash e \Rightarrow E \quad \beta, \text{CTX} \vdash e' \Rightarrow E'}{\text{ctx}, \beta, \text{CTX} \vdash e.f = e' \Rightarrow \text{CTX}' : E.f = E', \beta, \text{CTX}'} \text{ (FdStmtConv)}$$

$$\frac{\text{CTX}' = \llbracket ctx \rrbracket \quad \beta, \text{CTX} \vdash e \Rightarrow E}{\text{ctx}, \beta, \text{CTX} \vdash return \ e \Rightarrow \text{CTX}' : return \ E, \beta, undef} \text{ (RetStmtConv)} \qquad \frac{\begin{array}{c} \text{CTX} = \llbracket ctx \rrbracket \quad \beta, \text{CTX}' \vdash e \Rightarrow E \quad \beta, \text{CTX}' \vdash e' \Rightarrow E' \\ \beta' = \beta \cup \{(x, \text{ctx}, \text{CTX}, x_{\text{CTX}})\} \end{array}}{\text{ctx}, \beta, \text{CTX}' \vdash e.m(e') \Rightarrow \text{CTX} : x_{\text{CTX}} = E.m(E'), \beta', \text{CTX}} \text{ (MethdInvStmtConv)}$$

$$\frac{\begin{array}{c} \text{CTX} = \llbracket ctx \rrbracket \quad \beta, \text{CTX} \vdash e : E \quad \text{ctx}[\text{if } e\{\square\}\text{else}\{\overline{s}\}], \beta_0, \text{CTX} \vdash \overline{s_1} \Rightarrow \overline{B_1}, \beta_1, \text{CTX}_1 \quad \text{ctx}[\text{if } e\{\overline{s}\}\text{else}\{\square\}], \beta_0, \text{CTX} \vdash \overline{s_2} \Rightarrow \overline{B_2}, \beta_2, \text{CTX}_2 \\ \text{CTX}_3 = \text{CTX}[\text{if } \mathbf{E} \ \{\overline{\mathbf{B}}\}\text{else}\{\overline{\mathbf{B}}\}\text{join} \{\blacksquare\}] \quad \overline{\phi} = \{x_{\text{CTX}_3} = phi(\text{CTX}_1 : \mathbb{R}_{\leq \text{CTX}_1}(\beta_1, x), \text{CTX}_2 : \mathbb{R}_{\leq \text{CTX}_2}(\beta_2, x)) \mid x \in dom3((\beta_1 \cup \beta_2) - \beta_3)\} \\ \beta_3 = \{(x, \text{ctx}, \text{CTX}_3, x_{\text{CTX}_3}) \mid x \in dom3((\beta_1 \cup \beta_2) - \beta_0)\} \cup \beta_0 \cup \beta_1 \cup \beta_2 \end{array}}{\text{ctx}, \beta_0, \text{CTX}_0 \vdash \text{if } e\{\overline{s_1}\}\text{else}\{\overline{s_1}\} \Rightarrow \text{CTX} : \text{if } E\{\overline{B_1}\}\text{else}\{\overline{B_2}\}\text{join}\{\overline{\phi}\}, \beta_3, \text{CTX}_3} \text{ (IfStmtConv)}$$

$$\frac{\begin{array}{c} \text{CTX} = \llbracket ctx \rrbracket \quad \text{CTX}_1^{0,1} = \text{CTX}[\text{join } \{\blacksquare^{0,1}\}\text{while } \mathbf{E}\{\overline{\mathbf{B}}\}] \quad \beta_1, \text{CTX}_1^0 \vdash e \Rightarrow E \quad \overline{\phi} = \{x_{\text{CTX}_1} = phi(\text{CTX}_0 : \mathbb{R}_{\leq \text{CTX}_0}(\beta_0, x)) \mid x \in dom3(\beta_0)\} \\ \beta_1 = \{(x, \text{ctx}, \text{CTX}_1^0, x_{\text{CTX}_1}) \mid x \in dom3(\beta_0)\} \cup \beta_0 \quad \text{ctx}[\text{while } e\{\square\}], \beta_1, \text{CTX}_1^0 \vdash \overline{s} \Rightarrow \overline{B}, \beta_2, \text{CTX}_2 \\ \overline{\phi'} = \{x_{\text{CTX}1} = phi(\text{CTX}_0 : \mathbb{R}_{\leq \text{CTX}_0}(\beta_0, x), \text{CTX}_2 : \mathbb{R}_{\leq \text{CTX}_2}(\beta_2, x)) \mid x \in dom3(\beta_2 - \beta_1)\} \\ \theta = \{x_{\text{CTX}_1}/(\mathbb{R}_{\leq \text{CTX}_0}(\beta_0, x)) \mid x \in dom3(\beta_0) - dom3(\beta_2 - \beta_1)\} \quad \beta_3 = \{(x, \text{ctx}, \text{CTX}_1^1, x_{\text{CTX}_1}) \in dom3(\beta_2 - \beta_1)\} \cup \beta_0 \cup (\beta_2 - \beta_1) \end{array}}{\text{ctx}, \beta_0, \text{CTX}_0 \vdash \text{while } e\{\overline{s}\} \Rightarrow \text{CTX} : \text{join } \{\overline{\phi'}\}\text{while } \theta(E)\{\theta(\overline{B})\}, \beta_3, \text{CTX}_1^1} \text{ (WhileStmtConv)}$$

Figure 5: FJ to SSAFJ conversion

REFERENCES

[1] Davide Ancona and Andrea Corradi. 2016. A formal account of SSA in Java-like languages. In *Proceedings of the 18th Workshop on Formal Techniques for Java-like Programs, FTfJP@ECOOP 2016, Rome, Italy, July 17-22, 2016*, Vladimir Klebanov

(Ed.). ACM, 2. https://doi.org/10.1145/2955811.2955813

[2] John Aycock and Nigel Horspool. 2000. Simple Generation of Static Single-Assignment Form. In *Compiler Construction*, David A. Watt (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 110–125.

[3] Cláudio Belo Lourenço, Maria João Frade, and Jorge Sousa Pinto. 2016. Formalizing Single-Assignment Program Verification: An Adaptation-Complete Approach. In *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632*. Springer-Verlag, Berlin, Heidelberg, 41–67.

[4] Marc M. Brandis and Hanspeter Mössenböck. 1994. Single-Pass Generation of Static Single-Assignment Form for Structured Languages. *ACM Trans. Program. Lang. Syst.* 16, 6 (Nov. 1994), 1684–1698. https://doi.org/10.1145/197320.197331

[5] Matthias Braun, Sebastian Buchwald, Sebastian Hack, Roland Leißa, Christoph Mallon, and Andreas Zwinkau. 2013. Simple and Efficient Construction of Static Single Assignment Form. In *Compiler Construction*, Ranjit Jhala and Koen De Bosschere (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 102–122.

[6] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. 1998. Practical Improvements to the Construction and Destruction of Static Single Assignment Form. *Softw. Pract. Exper.* 28, 8 (July 1998), 859–881.

[7] David Brumley and Ivan Jager. 2011. Efficient Directionless Weakest Preconditions.

[8] Sebastian Buchwald, Denis Lohner, and Sebastian Ullrich. 2016. Verified Construction of Static Single Assignment Form. In *Proceedings of the 25th International Conference on Compiler Construction (CC 2016)*. Association for Computing Machinery, New York, NY, USA, 67–76. https://doi.org/10.1145/2892208.2892211

[9] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1989. An Efficient Method of Computing Static Single Assignment Form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '89)*. ACM, New York, NY, USA, 25–35. https://doi.org/10.1145/75277.75280

[10] Cormac Flanagan and James B. Saxe. 2001. Avoiding Exponential Explosion: Generating Compact Verification Conditions. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '01)*. Association for Computing Machinery, New York, NY, USA, 193–205. https://doi.org/10.1145/360204.360220

[11] K. Rustan M. Leino. 2005. Efficient weakest preconditions. *Inform. Process. Lett.* 93, 6 (2005), 281–288. https://doi.org/10.1016/j.ipl.2004.10.015

[12] Kenny Zhuo Ming Lu. 2022. Control flow obfuscation for Java code. https://github.com/obsidian-java.

[13] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1988. Global Value Numbers and Redundant Computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '88)*. Association for Computing Machinery, New York, NY, USA, 12–27. https://doi.org/10.1145/73560.73562

[14] Pop Sebastian. 2007. *The SSA Representation Framework: Semantics, Analyses and GCC Implementation*. Ph.D. Dissertation. Mines ParisTech. HAL Id: pastel-00002281.

[15] Vugranam C. Sreedhar and Guang R. Gao. 1995. A Linear Time Algorithm for Placing φ-Nodes. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*. Association for Computing Machinery, New York, NY, USA, 62–73. https://doi.org/10.1145/199448.199464

A ADDITIONAL FORMAL DEFINITIONS AND RESULTS

A.1 Semantics of FJ and SSAFJ

In Figure 6, we report the syntax and semantics of FJ.

In Figure 7, we consider syntax and semantics of SSAFJ.

A.2 SSA Conversion Consistency

We would like to show that the SSA conversion algorithm described earlier is consistent w.r.t. semantics of FJ and its SSA form. First of all we need to show that all the invocations of $\mathbb{R}_{\leq}.(\cdot, \cdot)$ are well defined.

DEFINITION 1 (PROGRAM CONTEXT JOIN SEMI-LATTICE). *Let β be a variable mapping and CTX be an SSAFJ program context. We say $\beta \vdash_L$ CTX iff $\{$CTX$' | (x, \text{ctx}, \text{CTX}', x') \in \beta, \text{CTX}' \leq \text{CTX}\}$ is a join semi-lattice.*

LEMMA A.1 (JOIN SEMI-LATTICE PRESERVATION). *Let β be a variable mapping. Let CTX be an SSAFJ program context, such that $\beta \vdash_L$*

CTX. *Let s be an FJ statement, and ctx be an FJ program context such that $\text{ctx}, \beta, \text{CTX} \vdash s \Rightarrow B, \beta', \text{CTX}'$. Then $\beta' \vdash_L \text{CTX}'$.*

PROOF SKETCH. We prove by structure induction over the derivation of $\text{ctx}, \beta, \text{CTX} \vdash s \Rightarrow B, \beta, \text{CTX}$. Most of the cases can be verified directly by the application of induction hypothesis.

□

DEFINITION 2 (CONSISTENT GLOBAL ENVIRONMENTS). *Let $genv \in$ GEnv and $genv' \in$ GENV. Then we say $genv \vdash genv'$ iff $\forall (C, m) \in dom(genv) : \vdash genv(C, m) \Rightarrow genv'(C, m)$.*

DEFINITION 3 (CONSISTENT LOCAL ENVIRONMENTS). *Let $lenv \in$ LEnv and $lenv' \in$ LEnv. Let β be a variable mapping, CTX be a SSAFJ program context. Then we say $lenv \vdash_{(CTX, \beta)} lenv'$ iff for all $x \in dom(lenv)$ we have $lenv(x) = lenv'(\mathbb{R}_{\leq CTX}(\beta, x))$.*

The following lemma states that the environment consistency is maintained by the SSA conversion algorithm and the SSAFJ program produces the same result as the original FJ program.

LEMMA A.2 (STATEMENT CONVERSION CONSISTENCY). *Let $\text{ctx}, \beta, \text{CTX} \vdash s \Rightarrow B, \beta', \text{CTX}'$. Let $genv \in$ GEnv and $genv' \in$ GENV and $genv \vdash genv'$. Let $lenv \in$ LEnv and $lenv' \in$ LEnv and $lenv \vdash_{(CTX, \beta)} lenv'$. Let str be a memory store. Then $\mathbb{S}_{fj}[\![s]\!]\, genv\, lenv\, str = (v, lenv'', str'')$ implies $\mathbb{B}_{ssa}[\![B]\!]\, genv'\, lenv'\, str = (v', lenv''', str''', l)$ such that $v = v'$ and $lenv'' \vdash_{(\beta', CTX')} lenv'''$ and $str'' = str'''$.*

PROOF SKETCH. We apply the result of Lemma A.1. We then prove the above lemma by induction over the derivation of $\text{ctx}, \beta, \text{CTX} \vdash s \Rightarrow B, \beta, \text{CTX}$. □

A.3 SSA Conversion Minimality

In this section, we show that the SSA programs produced by the presented algorithm are minimal. We consider the definition of minimal SSA based on dominance frontier adopted from Cytron's work [9]. First we define a control flow graph as a set of vertices (i.e. pairs of FJ program contexts). The control flow graph construction function is given in Figure 8. Let S denote a set of program contexts, we write $\text{ctx}[S]$ to denote $\{\text{ctx}[\text{ctx}'] \mid \text{ctx}' \in S\}$. Let P denote a set of pairs of program contexts, we write $\text{ctx}[P]$ to denote $\{(\text{ctx}[\text{ctx}_1], \text{ctx}[\text{ctx}_2]) \mid (\text{ctx}_1, \text{ctx}_2) \in P\}$.

For convenience, in this subsection, we treat statement sequence $\bar{s}$ as a special case of statement s.

DEFINITION 4 (PATH). *Let s be an FJ statement, G be a control flow graph such that $G = \mathbb{G}[\![s]\!]$. Then $p = [\text{ctx}_1, ..., \text{ctx}_n]$ is a path iff for all $i \in [1, n-1]$, we have $(ctx_i, ctx_{i+1}) \in G$.*

We consider the definition of the dominance relation among two program contexts.

DEFINITION 5 (DOMINANCE). *Let ctx_1 and ctx_2 be FJ program contexts. Then we say ctx_1 dominates ctx_2 iff for all path $p = [\Box, ..., \text{ctx}_2]$ we have $\text{ctx}_1 \in p$.*

We write $\text{ctx} \geqslant \text{ctx}'$ to denote ctx dominating ctx' and $\text{ctx} > > \text{ctx}'$ to denote ctx strictly dominating ctx'.

Next we consider the following definition of dominance frontier adopted from Cytron's work [9].

(ClassDecl)	cd	$::=$	class C $\{\overline{fd}; \overline{md}\}$		(Var)	x	$::=$	$x \mid y \mid ...$
(FieldDecl)	fd	$::=$	$t\ f$		(FieldName)	f	$::=$	$x \mid y \mid ...$
(MethDecl)	md	$::=$	$t\ m\ (t\ x)\ \{\overline{vd}; \overline{s}\}$		(Value)	v	$::=$	$c \mid loc \mid$ null
(VarDecl)	vd	$::=$	$t\ x$		(MemLoc)	loc	$::=$	$\text{loc}(0) \mid \text{loc}(1) \mid ...$
(Statement)	s	$::=$	$a \mid$ return $e \mid x = e.m(e)$		(Global Decls)	GEnv	$\subseteq$	(Class $\times$ [FieldDecl]) $\cup$
			$\mid$ while $e\ \{\overline{s}\} \mid$ if $e\ \{\overline{s}\}$ else $\{\overline{s}\}$					((Class $\times$ MethName) $\times$ MethDecl)
(Assignment)	a	$::=$	$x = e \mid e.f = e$		(Local Decls)	LEnv	$\subseteq$	(Variable $\times$ Value)
(Expression)	e	$::=$	$v \mid x \mid e.f \mid$ new $t() \mid$ this $\mid e\ op\ e$		(Memory Store)	Store	$\subseteq$	(MemLoc $\times$ Object)
(Operator)	op	$::=$	$+ \mid - \mid > \mid < \mid == \mid ...$		(Object)	obj	$::=$	obj(t, ρ)
(Type)	t	$::=$	$int \mid bool \mid void \mid C$		(Obj Field Map)	ρ	$\subseteq$	(FieldName $\times$ Value)

(Context) ctx $::=$ $\Box \mid$ ctx; $\mid$ ctx; $\overline{s} \mid s$; ctx $\mid$ if e {ctx} else {$\overline{s}$} $\mid$ if e {$\overline{s}$} else {ctx} $\mid$ while e {ctx}

$\overline{\mathbb{S}}_{fj}[\![\cdot]\!] :: [\text{Statement}] \to \text{GEnv} \to \text{LEnv} \to \text{Store} \to (\text{Value}, \text{LEnv}, \text{Store})$
$\mathbb{E}_{fj}[\![\cdot]\!] :: \text{Expression} \to \text{GEnv} \to \text{LEnv} \to \text{Store} \to (\text{Value}, \text{Store})$
$\mathbb{S}_{fj}[\![\cdot]\!] :: \text{Statement} \to \text{GEnv} \to \text{LEnv} \to \text{Store} \to (\text{Value}, \text{LEnv}, \text{Store})$
$\mathbb{S}_{fj}[\![\text{if } e\ \{\overline{s_1}\}\text{else}\{\overline{s_2}\}]\!]\ genv\ lenv\ str = \text{case } \mathbb{E}_{fj}[\![e]\!]\ genv\ lenv\ str \text{ of}$
$\quad \{(true, str') \to \overline{\mathbb{S}}_{fj}[\![\overline{s_1}]\!]\ genv\ lenv\ str'; (false, str') \to \overline{\mathbb{S}}_{fj}[\![\overline{s_2}]\!]\ genv\ lenv\ str'\}$
$\mathbb{S}_{fj}[\![\text{while } e\{\overline{s}\}]\!]\ genv\ lenv\ str = \text{case } \mathbb{E}_{fj}[\![e]\!]\ genv\ lenv\ str \text{ of}$
$\quad \{(false, str') \to (null, lenv, str'); (true, str') \to \text{case } \overline{\mathbb{S}}_{fj}[\![\overline{s}]\!]\ genv\ lenv\ str' \text{ of}(v, lenv'\ str'') \to \mathbb{S}_{fj}[\![\text{while } e\{\overline{s}\}]\!]\ genv\ lenv'\ str''\}$

Figure 6: Syntax and Semantics of FJ (excerpt)

(METHDECL)	MD	$::=$	$t\ m\ (t\ x)\ \{\overline{VD}; \overline{B}\}$		(Phi)	ϕ	$::=$	$x = \text{phi}(\overline{l : x})$
(VARDECL)	VD	$::=$	$t\ x$		(Label)	l	$::=$	CTX
(BLOCK)	B	$::=$	$l : \{S\}$		(Global Decl)	GENV	$\subseteq$	(Class $\times$ [FieldDecl])
(STATEMENT)	S	$::=$	$A \mid$ return $E \mid x = E.m(E)$					$\cup$((Class $\times$ MethName)
			$\mid$ join $\{\overline{\phi}\}$ while E $\{\overline{B}\}$					$\times$METHODDECL)
			$\mid$ if E $\{\overline{B}\}$ else $\{\overline{B}\}$ join $\{\overline{\phi}\}$					

(CONTEXT) CTX $::=$ $\Box \mid$ CTX; $\mid$ CTX; $\overline{B} \mid B$; CTX $\mid$ if E {CTX} else {$\overline{B}$} join {$\overline{\phi}$} $\mid$ if E {$\overline{B}$} else {CTX} join {$\overline{\phi}$}
$\quad \mid$ if E {$\overline{B}$} else {$\overline{B}$} join{$\blacksquare$} $\mid$ join {$\blacksquare^b$} while E {$\overline{B}$} $\mid$ join {$\overline{\phi}$} while E {CTX}

$\overline{\mathbb{B}}_{ssa}[\![\cdot]\!] :: [\text{Block}] \to \text{CONTEXT} \to \text{GENV} \to \text{LEnv} \to \text{Store} \to (\text{Value}, \text{LEnv}, \text{Store}, \text{CONTEXT})$
$\mathbb{E}_{ssa}[\![\cdot]\!] :: \text{Expression} \to \text{GENV} \to \text{LEnv} \to \text{Store} \to (\text{Value}, \text{Store})$
$\mathbb{B}_{ssa}[\![\cdot]\!] :: \text{Block} \to \text{CONTEXT} \to \text{GENV} \to \text{LEnv} \to \text{Store} \to (\text{Value}, \text{LEnv}, \text{Store}, \text{CONTEXT})$
$\mathbb{B}_{ssa}[\![\text{CTX} : \{\text{if } E\ \{\overline{B_1}\} \text{ else } \{\overline{B_2}\} \text{join}\{\overline{\phi}\}\}]\!]\ \text{CTX}_p\ genv\ lenv\ str = \text{case } \mathbb{E}_{ssa}[\![E]\!]\ genv\ lenv\ str \text{ of}$
$\quad (true, str') \to \text{case } \overline{\mathbb{B}}_{ssa}[\![\overline{B_1}]\!]\ \text{CTX}\ genv\ lenv\ str' \text{ of } (v, lenv', str'', \text{CTX}') \to (v, \mathbb{F}_{ssa}[\![\overline{\phi}]\!]\ \text{CTX}'\ lenv', str'', \text{CTX}[\text{if } E\ \{\overline{B}\}\text{ else}\{\overline{B}\}\text{join}\{\blacksquare\}\}])$
$\quad (false, str') \to \text{case } \overline{\mathbb{B}}_{ssa}[\![\overline{B_2}]\!]\ \text{CTX}\ genv\ lenv\ str' \text{ of } (v, lenv', str'', \text{CTX}') \to (v, \mathbb{F}_{ssa}[\![\overline{\phi}]\!]\ \text{CTX}'\ lenv', str'', \text{CTX}[\text{if } E\ \{\overline{B}\}\text{else}\{\overline{B}\}\text{join}\{\blacksquare\}\}])$
$\mathbb{B}_{ssa}[\![\text{CTX} : \{\text{join } \{\overline{\phi}\} \text{ while } E\ \{\overline{B}\}\}]\!]\ \text{CTX}_p\ genv\ lenv\ str = \text{let } lenv' = \mathbb{F}_{ssa}[\![\overline{\phi}]\!]\ \text{CTX}_p\ lenv$
$\quad \text{in case } \mathbb{E}_{ssa}[\![E]\!]\ genv\ lenv'\ str \text{ of}$
$\quad (false, str') \to (null, lenv', str', \text{CTX}[\text{join}\{\blacksquare\}\text{while}E\{\overline{B}\}])$
$\quad (true, str') \to \text{case } \overline{\mathbb{B}}_{ssa}[\![\overline{B}]\!]\ \text{CTX}[\text{join}\{\blacksquare\}\text{while}E\{\overline{B}\}\ genv\ lenv'\ str'\text{of}$
$\quad\quad \{(v, lenv'', str'', \text{CTX}') \to \mathbb{B}_{ssa}[\![\text{CTX} : \{ \text{join } \{\overline{\phi}\} \text{ while } E\ \{\overline{B}\}\}]\!]\ \text{CTX}'\ genv\ lenv''\ str''\}$
$\mathbb{F}_{ssa}[\![\cdot]\!] :: [\text{Phi}] \to \text{CONTEXT} \to \text{LEnv} \to \text{LEnv}$
$\mathbb{F}_{ssa}[\![[]]\!]\ \text{CTX}\ lenv = lenv$
$\mathbb{F}_{ssa}[\![x = \text{phi}(\text{CTX}_1 : x_1, ..., \text{CTX}_i : x_i, ..., \text{CTX}_n : x_n); \overline{\phi}]\!]\ \text{CTX}_i\ lenv = \mathbb{F}_{ssa}[\![\overline{\phi}]\!]\ \text{CTX}_i\ lenv + (x, lenv(x_i))$

Figure 7: Syntax and Semantics of SSAFJ (excerpt)

DEFINITION 6 (DOMINANCE FRONTIER). *Let s be an FJ program statement and* ctx *be an FJ program context. We define the dominance frontier of* ctx *as follows.*

$DF(\text{ctx}|s) = \{\text{ctx}_2 \mid \exists(\text{ctx}_1, \text{ctx}_2) \in \mathbb{G}[\![s]\!] : \text{ctx} \geqq \text{ctx}_1 \wedge \neg(\text{ctx} \gg \text{ctx}_2)\}$

The following lemma states that dominance frontiers of an FJ program are unique if exist.

LEMMA A.3 (DF IS UNIQUE). *Let s be an FJ statement,* ctx$_0$ *be an FJ program context. Then for all* ctx $\in DF(\text{ctx}_0|s)$ *and* ctx$'$ $\in DF(\text{ctx}_0|s)$, *we have* ctx $=$ ctx$'$.

To prove the above lemma we need a supporting lemma

LEMMA A.4 (BOX DOMINATES ALL). *Let* ctx *be an FJ program context. Then $\Box$ dominates* ctx.

PROOF. It follows from Definition 5. $\qquad\square$

$\mathbb{G}[\![\cdot]\!]$ :: Statement $\rightarrow$ {(Context, Context)}
$\mathbb{G}[\![a]\!] = \{\}$
$\mathbb{G}[\![\text{return } e]\!] = \{\}$
$\mathbb{G}[\![\text{if } e\{s_1\}\text{else}\{s_2\}]\!] = \text{let } \{G_1 = \mathbb{G}[\![s_1]\!]; G_2 = \mathbb{G}[\![s_2]\!]\}$
  in $\{(\Box, \text{if } e \{\Box\}\text{else}\{s\}), (\Box, \text{if } e \{s\}\text{else}\{\Box\})\} \cup$
  $(\text{if } e\{\Box\}\text{else}\{\bar{s}\})[G_1] \cup (\text{if } e\{\bar{s}\}\text{else}\{\Box\})[G_2]$
$\mathbb{G}[\![\text{while } e \{s\}]\!] = \text{let } \{G = \mathbb{G}[\![s]\!]; O = \mathbb{X}[\![s]\!]\}$
  in $\{(\Box, \text{while } e \{\Box\})\} \cup (\text{while } e \{\Box\})[G]$
  $\cup \{((\text{while } e \{\Box\})[\text{ctx}], \Box) \mid \text{ctx} \in O\}$

$\mathbb{G}[\![\cdot]\!]$ :: [Statement] $\rightarrow$ {(Context, Context)}
$\mathbb{G}[\![s;]\!] = \mathbb{G}[\![s]\!]$
$\mathbb{G}[\![s_1; \overline{s_2}]\!] = \text{let } \{G_1 = \mathbb{G}[\![s_1]\!]; G_2 = \mathbb{G}[\![\overline{s_2}]\!]; O_1 = \mathbb{X}[\![s_1]\!]\}$
  in $\{(\Box, (\Box; \bar{s}))\} \cup (\Box; \bar{s})[G_1] \cup$
  $s; \Box)[G_2] \cup \{((\Box; \bar{s})[\text{ctx}], (s; \Box)) \mid \text{ctx} \in O_1\}$

$\mathbb{X}[\![\cdot]\!]$ :: Statement $\rightarrow$ {Context}
$\mathbb{X}[\![a]\!] = \{\Box\}$
$\mathbb{X}[\![\text{return } e]\!] = \{\}$
$\mathbb{X}[\![\text{if } e\{s_1\}\text{else}\{s_2\}]\!] = \{(\text{if } e\{\Box\}\text{else}\{\bar{s}\})[\text{ctx}] \mid \text{ctx} \in \mathbb{X}[\![s_1]\!]\} \cup$
  $\{(\text{if } e\{\bar{s}\}\text{else}\{\Box\})[\text{ctx}] \mid \text{ctx} \in \mathbb{X}[\![s_2]\!]\}$
$\mathbb{X}[\![\text{while } e \{s\}]\!] = \{\Box\}$

$\mathbb{X}[\![\cdot]\!]$ :: [Statement] $\rightarrow$ {Context}
$\mathbb{X}[\![s;]\!] = \mathbb{X}[\![s]\!]$
$\mathbb{X}[\![s_1; \overline{s_2}]\!] = \{s; \text{ctx} \mid \text{ctx} \in \mathbb{X}[\![\overline{s_2}]\!]\}$

Figure 8: Control flow graph construction

Now we can prove Lemma A.3.

PROOF. We prove this lemma together with another sub lemma for the case of $\bar{s}$.

Let $\bar{s}$ be an FJ statement sequence, ctx_0 be an FJ program context such that Then for all $\text{ctx} \in DF(\text{ctx}_0 \mid \bar{s})$ and $\text{ctx}' \in DF(\text{ctx}_0 \mid \bar{s})$, we have $\text{ctx} = \text{ctx}'$.

We prove the induction of s (and $\bar{s}$).

- Case a: $DF(\text{ctx}_0 \mid a) = \{\}$. We have verified this case.
- Case return e: same as the previous case.
- Case if $e \{\overline{s_1}\}$ else $\{\overline{s_2}\}$. Applying Definition 6, we have

$DF(\text{ctx}_0 \mid \text{if } e \{\overline{s_1}\} \text{ else } \{\overline{s_2}\}) =$
$\{\text{ctx}_2 \mid \exists (\text{ctx}_1, \text{ctx}_2) \in \mathbb{G}[\![\text{if } e \{\overline{s_1}\} \text{ else } \{\overline{s_2}\}]\!] : \text{ctx}_0 \gneqq \text{ctx}_1$
$\wedge \neg(\text{ctx}_0 \gg \text{ctx}_2)\}$

By applying the definition of control flow graph, we have

$\mathbb{G}[\![\text{if } e \{\overline{s_1}\} \text{ else } \{\overline{s_2}\}]\!] = \text{let } \{G_1 = \mathbb{G}[\![\overline{s_1}]\!]; G_2 = \mathbb{G}[\![\overline{s_2}]\!]\}$
in $\{(\Box, \text{if } e\{\Box\}\text{else}\{\bar{s}\}), (\Box, \text{if } e\{\bar{s}\}\text{else}\{\Box\})\} \cup$ (1)
$(\text{if } e \{\Box\} \text{ else } \{\bar{s}\})[G_1] \cup$ (2)
$(\text{if } e \{\bar{s}\} \text{ else } \{\Box\})[G_2]$ (3)

By Lemma A.4, we can ignore edges coming from (1). Note that ctx_0 can be either in s_1 or s_2 exclusively. For edges arising from (2) or (3), we apply I.H. show that the result is valid. We have verified this case
- Case while $e \{\bar{s}\}$: Applying Definition 6, we have

$DF(\text{ctx}_0 \mid \text{while } e \{\bar{s}\}) =$
$\{\text{ctx}_2 \mid \exists (\text{ctx}_1, \text{ctx}_2) \in \mathbb{G}[\![\text{while } e \{\bar{s}\}]\!] : \text{ctx}_0 \gneqq \text{ctx}_1$
$\wedge \neg(\text{ctx}_0 \gg \text{ctx}_2)\}$

By applying the definition of control flow graph, we have

$\mathbb{G}[\![\text{while } e \{\bar{s}\}]\!] = \text{let } \{G = \mathbb{G}[\![\bar{s}]\!]; O = \mathbb{X}[\![\bar{s}]\!]\}$
in $\{(\Box, \text{while } e \{\Box\})\} \cup$ (4)
$(\text{while } e \{\Box\})[G] \cup$ (5)
$\{((\text{while } e \{\Box\})[\text{ctx}], \Box) \mid \text{ctx} \in O\}$ (6)

By Lemma A.4, we can ignore edges coming from (4). For edges arising from (5), we apply I.H. to show that the result is valid. Edges arising from (6) share the same destination context $\Box$. We have verified this case.
- Case $s_1; \overline{s_2}$: Applying Definition 6, we have

$DF(\text{ctx}_0 \mid s_1; \overline{s_2}) =$
$\{\text{ctx}_2 \mid \exists (\text{ctx}_1, \text{ctx}_2) \in \mathbb{G}[\![s_1; \overline{s_2}]\!] : \text{ctx}_0 \gneqq \text{ctx}_1$
$\wedge \neg(\text{ctx}_0 \gg \text{ctx}_2)\}$

By applying the definition of control flow graph, we have

$\mathbb{G}[\![s_1; \overline{s_2}]\!] = \text{let } \{G_1 = \mathbb{G}[\![s_1]\!]; G_2 = \mathbb{G}[\![\overline{s_2}]\!]; O = \mathbb{X}[\![s_1]\!]\}$
in $\{(\Box, (\Box; \bar{s}))\} \cup$ (7)
$(\Box; \bar{s})[G_1] \cup$ (8)
$(s; \Box)[G_2] \cup$ (9)
$\{((\Box; \bar{s})[\text{ctx}], (s; \Box)) \mid \text{ctx} \in O\}$ (10)

By Lemma A.4, we can ignore edges coming from (7). For edges arising from (8) and (9), we apply I.H. to show that the result is valid. Edges arising from (6) share the same destination context $(s; \Box)$. We have verified this case.

$\square$

To establish the minimality result, we introduce the definition of the dominance frontier handle.

DEFINITION 7 (DOMINANCE FRONTIER HANDLE). *Let s be an FJ statement and ctx be an FJ program context. We define the dominance frontier handle of ctx as follows.*

$DF_h(\text{ctx} \mid s) = \{<\text{ctx}_1, \text{ctx}'_1> \mid \exists (\text{ctx}_1, \text{ctx}_2) \in \mathbb{G}[\![s]\!]$
$\wedge \exists (\text{ctx}'_1, \text{ctx}_2) \in \mathbb{G}[\![s]\!] : \text{ctx} \gneqq \text{ctx}_1 \wedge \neg(\text{ctx} \gg \text{ctx}'_1)\}$
$\cup \{<\text{ctx}_1, \text{ctx}'_1> \mid \text{ctx}_1 \in \mathbb{X}[\![s]\!] \wedge \text{ctx}'_1 \in \mathbb{X}[\![s]\!]$
$\wedge \text{ctx} \gneqq \text{ctx}_1 \wedge \neg(\text{ctx} \gg \text{ctx}'_1)\}$

Intuitively speaking, the dominance frontier returns the (transitive) successor of the input context ctx where ctx loses its dominance. The dominance frontier handle computes the set of pairs of immediate predecessors to this successor. As a convention, given $<\text{ctx}_1, \text{ctx}'_1> \in DF_h(\text{ctx}_0 \mid s)$, we assume that ctx_1 is the one that is dominated by ctx_0.

We consider the follow definition of iterative dominance frontier adopted from Cytron's work [9].

DEFINITION 8 (ITERATIVE DOMINANCE FRONTIER).

$$DF(\text{ctx} \mid s)^0 = DF(\text{ctx} \mid s)$$
$$DF(\text{ctx} \mid s)^{n+1} = \text{let } F = DF(\text{ctx} \mid s)^n$$
$$\text{in } DF(F \mid s) \cup DF(\text{ctx} \mid s)^n$$

where $DF(F \mid s) = \bigcup_{\text{ctx} \in F} DF(\text{ctx} \mid s)$

The iterative dominance frontier captures the dominance frontier of the input context ctx and the cascaded dominance frontiers. We define the handle of iterative dominance frontiers.

DEFINITION 9 (ITERATIVE DOMINANCE FRONTIER HANDLE).

$$
\begin{aligned}
DF_h(\mathsf{ctx}\,|\,s)^0 \;=\;& DF_h(\mathsf{ctx}\,|\,s) \\
DF_h(\mathsf{ctx}\,|\,s)^{n+1} \;=\;& \text{let } F = \{\mathsf{ctx}_2 \mid (\mathsf{ctx}_1, \mathsf{ctx}_2) \in \mathbb{G}[\![s]\!] \\
& \quad \wedge (\mathsf{ctx}'_1, \mathsf{ctx}_2) \in \mathbb{G}[\![s]\!] \wedge \\
& \quad <\mathsf{ctx}_1, \mathsf{ctx}'_1> \in DF_h(\mathsf{ctx}\,|\,s)^n\} \\
& \text{in } DF_h(F\,|\,s) \cup DF_h(\mathsf{ctx}\,|\,s)^n
\end{aligned}
$$

where $DF_h(F\,|\,s) = \bigcup_{\mathsf{ctx} \in F} DF_h(\mathsf{ctx}\,|\,s)$.

The following lemma states that there exists a correspondence between the iterative dominance frontiers and the iterative dominance frontier handles.

LEMMA A.5 (DF-DFH CORRESPONDENCE). *Let s be an FJ statement such that $\mathbb{X}[\![s]\!] = \{\}$ and ctx_0 be an FJ program context referring to some location in s. Then*

(1) *$\mathsf{ctx}_2 \in DF(\mathsf{ctx}_0\,|\,s)^n$ implies that $\exists <\mathsf{ctx}_1, \mathsf{ctx}'_1> \in DF_h(\mathsf{ctx}_0\,|\,s)^n$ such that $(\mathsf{ctx}_1, \mathsf{ctx}_2) \in \mathbb{G}[\![s]\!]$ and $(\mathsf{ctx}'_1, \mathsf{ctx}_2) \in \mathbb{G}[\![s]\!]$; and*

(2) *$<\mathsf{ctx}_1, \mathsf{ctx}'_1> \in DF_h(\mathsf{ctx}_0\,|\,s)^n$ implies that $\exists \mathsf{ctx}_2 \in DF(\mathsf{ctx}_0\,|\,s)^n$ such that $(\mathsf{ctx}_1, \mathsf{ctx}_2) \in \mathbb{G}[\![s]\!]$ and $(\mathsf{ctx}'_1, \mathsf{ctx}_2) \in \mathbb{G}[\![s]\!]$.*

PROOF. (1) We consider the proof of $\mathsf{ctx}_2 \in DF(\mathsf{ctx}_0\,|\,s)^n$ implies that $\exists <\mathsf{ctx}_1, \mathsf{ctx}'_1> \in DF_h(\mathsf{ctx}_0\,|\,s)^n$ such that $(\mathsf{ctx}_1, \mathsf{ctx}_2) \in \mathbb{G}[\![s]\!]$ and $(\mathsf{ctx}'_1, \mathsf{ctx}_2) \in \mathbb{G}[\![s]\!]$.
We prove by induction of n.

- Case 0. We would like to show $\mathsf{ctx}_2 \in DF(\mathsf{ctx}_0\,|\,s)$ implies $\exists <\mathsf{ctx}_1, \mathsf{ctx}'_1> \in DF_h(\mathsf{ctx}_0\,|\,s)$ such that $(\mathsf{ctx}_1, \mathsf{ctx}_2) \in \mathbb{G}[\![s]\!]$ and $(\mathsf{ctx}'_1, \mathsf{ctx}_2) \in \mathbb{G}[\![s]\!]$.
  By Definition 6, we have $\exists (\mathsf{ctx}''_1, \mathsf{ctx}_2) \in \mathbb{G}[\![s]\!]$ such that $\mathsf{ctx}_0 \ggeq \mathsf{ctx}''_1$ and $\neg(\mathsf{ctx}_0 \gg \mathsf{ctx}_2)$. It follows that there exists a path $p = \square, ..., \mathsf{ctx}_2$ such that $\mathsf{ctx}_0 \in p$. By Lemma A.4, $\mathsf{ctx}_0 \neq \square$ and by definition 6, $\mathsf{ctx}_2 \neq \square$. Thus $p = \square, ..., \mathsf{ctx}'''_1, \mathsf{ctx}_2$. Let $p = p', \mathsf{ctx}_2$. It follows that $\mathsf{ctx}_0 \notin p'$, therefore $\neg(\mathsf{ctx}_0 \gg \mathsf{ctx}'''_1)$. We conclude that we found $<\mathsf{ctx}''_1, \mathsf{ctx}'''_1> \in DF_h(\mathsf{ctx}_0\,|\,s)$ and $(\mathsf{ctx}''_1, \mathsf{ctx}_2) \in \mathbb{G}[\![s]\!]$ and $(\mathsf{ctx}'''_1, \mathsf{ctx}_2) \in \mathbb{G}[\![s]\!]$. We have verified this case.

- Case $n + 1$. We would like to show that $\mathsf{ctx}_2 \in DF(\mathsf{ctx}_0\,|\,s)^{n+1}$ implies that $\exists <\mathsf{ctx}_1, \mathsf{ctx}'_1> \in DF_h(\mathsf{ctx}_0\,|\,s)^{n+1}$ such that $(\mathsf{ctx}_1, \mathsf{ctx}_2) \in \mathbb{G}[\![s]\!]$ and $(\mathsf{ctx}'_1, \mathsf{ctx}_2) \in \mathbb{G}[\![s]\!]$.
  By Definition 8, we have

$$
\begin{aligned}
DF(\mathsf{ctx}_0\,|\,s)^{n+1} =\;& \text{let } F = DF(\mathsf{ctx}_0\,|\,s)^n \\
& \text{in } DF(F\,|\,s) \cup DF(\mathsf{ctx}_0\,|\,s)^n
\end{aligned}
$$

By Definition 9, we have

$$
\begin{aligned}
DF_h(\mathsf{ctx}_0\,|\,s)^{n+1} =\;& \text{let } F' = \\
& \{\mathsf{ctx}'_2 \mid (\mathsf{ctx}_1, \mathsf{ctx}'_2) \in \mathbb{G}[\![s]\!] \wedge \\
& (\mathsf{ctx}'_1, \mathsf{ctx}'_2) \in \mathbb{G}[\![s]\!] \wedge \\
& <\mathsf{ctx}_1, \mathsf{ctx}'_1> \in DF_h(\mathsf{ctx}_0\,|\,s)^n\} \\
& \text{in } DF_h(F'\,|\,s) \cup DF_h(\mathsf{ctx}_0\,|\,s)^n
\end{aligned}
$$

If $\mathsf{ctx}_2 \in DF(\mathsf{ctx}_0\,|\,s)^n$, we can apply I.H. to show that $\exists <\mathsf{ctx}_1, \mathsf{ctx}'_1> \in DF_h(\mathsf{ctx}_0\,|\,s)^n$.
Otherwise, we consider the case where $\mathsf{ctx}_2 \in DF(F\,|\,s) - DF(\mathsf{ctx}_0\,|\,s)^n$. Recall that

$$
DF(F\,|\,s) = \bigcup_{\mathsf{ctx}'_2 \in DF(\mathsf{ctx}_0|s)} DF(\mathsf{ctx}'_2\,|\,s) \quad (1)
$$

The goal now is to find $<\mathsf{ctx}_1, \mathsf{ctx}'_1> \in DF_h(F'\,|\,s) \cup DF_h(\mathsf{ctx}_0\,|\,s)^n$. Recall that

$$
DF_h(F'\,|\,s) = \bigcup_{\mathsf{ctx}''_2 \in H} DF_h(\mathsf{ctx}''_2\,|\,s) \quad (2)
$$

where $H = \{\mathsf{ctx}' \mid (\mathsf{ctx}_1, \mathsf{ctx}') \in \mathbb{G}[\![s]\!] \wedge (\mathsf{ctx}'_1, \mathsf{ctx}') \in \mathbb{G}[\![s]\!] \wedge <ctx_1, \mathsf{ctx}'_1> \in DF_h(\mathsf{ctx}_0\,|\,s)^n$
W.l.o.g., let's fix a particular

$$
\mathsf{ctx}'_2 \in DF(\mathsf{ctx}_0\,|\,s)^n \quad (3)
$$

such that $\mathsf{ctx}_2 \in DF(ctx'_2\,|\,s)$ and $\mathsf{ctx}_2 \notin DF(\mathsf{ctx}_0\,|\,s)^n$.
By applying I.H. to (3), we must be able find $<\mathsf{ctx}''_1, \mathsf{ctx}'''_1> \in DF_h(\mathsf{ctx}_0\,|\,s)^n$ such that $(\mathsf{ctx}''_1, \mathsf{ctx}'_2) \in \mathbb{G}[\![s]\!]$ and $(\mathsf{ctx}'''_1, \mathsf{ctx}'_2) \in \mathbb{G}[\![s]\!]$. By Lemma A.3, the ctx''_2 in (2) must be the same as ctx'_2.
From (3), we can apply the same steps used in Case 0 to show that $\exists <\mathsf{ctx}_1, \mathsf{ctx}'_1> \in DF_h(ctx'_2\,|\,s)$ such that $(\mathsf{ctx}_1, \mathsf{ctx}_2) \in \mathbb{G}[\![s]\!]$ and $(\mathsf{ctx}'_1, \mathsf{ctx}_2) \in \mathbb{G}[\![s]\!]$. We have verified this case.

(2) We consider the proof of
$<\mathsf{ctx}_1, \mathsf{ctx}'_1> \in DF_h(\mathsf{ctx}_0\,|\,s)^n$ implies that $\exists \mathsf{ctx}_2 \in DF(\mathsf{ctx}_0\,|\,s)^n$ such that $(\mathsf{ctx}_1, \mathsf{ctx}_2) \in \mathbb{G}[\![s]\!]$ and $(\mathsf{ctx}'_1, \mathsf{ctx}_2) \in \mathbb{G}[\![s]\!]$
We prove by induction of n.

- Case 0. We would like to show that $<\mathsf{ctx}_1, \mathsf{ctx}'_1> \in DF_h(\mathsf{ctx}_0\,|\,s)$ implies $\exists \mathsf{ctx}_2 \in DF(\mathsf{ctx}_0\,|\,s)$ such that $(\mathsf{ctx}_1, \mathsf{ctx}_2) \in \mathbb{G}[\![s]\!]$ and $(\mathsf{ctx}'_1, \mathsf{ctx}_2) \in \mathbb{G}[\![s]\!]$.
  By Definition 7, we have

$$
\begin{aligned}
& DF_h(\mathsf{ctx}_0\,|\,s) = \\
& \{<\mathsf{ctx}_1, \mathsf{ctx}'_1> \mid (\mathsf{ctx}_1, \mathsf{ctx}_2) \in \mathbb{G}[\![s]\!] \wedge (\mathsf{ctx}'_1, \mathsf{ctx}_2) \in \mathbb{G}[\![s]\!] \\
& \wedge \mathsf{ctx}_0 \ggeq \mathsf{ctx}_1 \wedge \neg(\mathsf{ctx}_0 \gg \mathsf{ctx}'_1)\} \quad (4) \\
& \cup \\
& \{<\mathsf{ctx}_1, \mathsf{ctx}'_1> \mid \mathsf{ctx}_1 \in \mathbb{X}[\![s]\!] \wedge \mathsf{ctx}'_1 \in \mathbb{X}[\![s]\!] \wedge \\
& \mathsf{ctx}_0 \ggeq \mathsf{ctx}_1 \wedge \neg(\mathsf{ctx}_0 \gg \mathsf{ctx}'_1)\} \quad (5)
\end{aligned}
$$

  Note that the set (5) must be empty since $\mathbb{X}[\![s]\!] = \{\}$.
  $<\mathsf{ctx}_1, \mathsf{ctx}'_1>$ is from set (4). It follows that we find $\mathsf{ctx}_2 \in DF(\mathsf{ctx}_0\,|\,s)$ such that $(\mathsf{ctx}_1, \mathsf{ctx}_2) \in \mathbb{G}[\![s]\!]$ and $(\mathsf{ctx}'_1 \mathsf{ctx}_2) \in \mathbb{G}[\![s]\!]$. We have verified this case.

- Case $n + 1$. We would like to show that $<\mathsf{ctx}_1, \mathsf{ctx}'_1> \in DF_h(\mathsf{ctx}_0\,|\,s)^{n+1}$ implies $\exists \mathsf{ctx}_2 \in DF(\mathsf{ctx}_0\,|\,s)^{n+1}$ such that $(\mathsf{ctx}_1, \mathsf{ctx}_2) \in \mathbb{G}[\![s]\!]$ and $(\mathsf{ctx}'_1, \mathsf{ctx}_2) \in \mathbb{G}[\![s]\!]$.
  By Definition 9, we have

$$
\begin{aligned}
DF_h(\mathsf{ctx}_0\,|\,s)^{n+1} =\;& \text{let } F' = \\
& \{\mathsf{ctx}'_2 \mid (\mathsf{ctx}_1, \mathsf{ctx}'_2) \in \mathbb{G}[\![s]\!] \wedge \\
& (\mathsf{ctx}'_1, \mathsf{ctx}'_2) \in \mathbb{G}[\![s]\!] \wedge \\
& <\mathsf{ctx}_1, \mathsf{ctx}'_1> \in DF_h(\mathsf{ctx}_0\,|\,s)^n\} \\
& \text{in } DF_h(F'\,|\,s) \cup DF_h(\mathsf{ctx}_0\,|\,s)^n
\end{aligned}
$$

  By Definition 8, we have

$$
\begin{aligned}
DF(\mathsf{ctx}_0\,|\,s)^{n+1} =\;& \text{let } F = DF(\mathsf{ctx}_0\,|\,s)^n \\
& \text{in } DF(F\,|\,s) \cup DF(\mathsf{ctx}_0\,|\,s)^n
\end{aligned}
$$

If $<\mathsf{ctx}_1, \mathsf{ctx}'_1> \in DF_h(\mathsf{ctx}_0\,|\,s)^n$, we can apply I.H. to show that $\exists \mathsf{ctx}_2 \in DF(\mathsf{ctx}_0\,|\,s)^n$.
Otherwise, we consider the case where $<\mathsf{ctx}_1, \mathsf{ctx}'_1> \in DF_h(F'\,|\,s) - DF_h(\mathsf{ctx}_0\,|\,s)^n$.

Recall that

$$DF_h(F' \mid s) = \bigcup_{\mathsf{ctx}_2'' \in H} DF_h(\mathsf{ctx}_2'' \mid s) \quad (6)$$

where $H = \{\mathsf{ctx}_2'' \mid (\mathsf{ctx}_1, \mathsf{ctx}_2'') \in \mathbb{G}[\![s]\!] \wedge (\mathsf{ctx}_1', \mathsf{ctx}_2'') \in \mathbb{G}[\![s]\!] \wedge <\mathsf{ctx}_1, \mathsf{ctx}_1'> \in DF_h(\mathsf{ctx}_0 \mid s)^n$
We would like to find $\mathsf{ctx}_2 \in DF(F \mid s) \cup DF(\mathsf{ctx}_0 \mid s)^n$.
Recall that

$$DF(F \mid s) = \bigcup_{\mathsf{ctx}_2' \in DF(\mathsf{ctx}_0 \mid s)} DF(\mathsf{ctx}_2' \mid s) \quad (7)$$

W.l.o.g., let's fix a particular pair $<\mathsf{ctx}_1'', \mathsf{ctx}_1'''> \in DF_h(\mathsf{ctx}_0 \mid s)^n$ such that $(\mathsf{ctx}_1'', \mathsf{ctx}_2'') \in \mathbb{G}[\![s]\!]$ and $(\mathsf{ctx}_1''', \mathsf{ctx}_2'') \in \mathbb{G}[\![s]\!]$ (8). It also fixes a particular $\mathsf{ctx}_2'' \in H$.
It follows that

$$<\mathsf{ctx}_1, \mathsf{ctx}_1'> \in DF_h(\mathsf{ctx}_2'' \mid s) \quad (9)$$

and

$$<\mathsf{ctx}_1, \mathsf{ctx}_1'> \notin DF_h(\mathsf{ctx}_0 \mid s)^n$$

By I.H. to (8), we must be able to find $\mathsf{ctx}_2' \in DF(\mathsf{ctx}_0 \mid s)^n$ such that $(\mathsf{ctx}_1'', \mathsf{ctx}_2') \in \mathbb{G}[\![s]\!]$ and $(\mathsf{ctx}_1''', \mathsf{ctx}_2') \in \mathbb{G}[\![s]\!]$. By Lemma A.3, we conclude that $\mathsf{ctx}_2' = \mathsf{ctx}_2''$.
Now from (9), we can apply the same set of steps as in Case 0 to show that $\exists \mathsf{ctx}_2 \in DF(\mathsf{ctx}_2'' \mid s)$ such that $(\mathsf{ctx}_1, \mathsf{ctx}_2) \in \mathbb{G}[\![s]\!]$ and $(\mathsf{ctx}_1', \mathsf{ctx}_2) \in \mathbb{G}[\![s]\!]$. We have verified this case.

□

Next we would like to show that the SSA constructed by the conversion functions defined in Figure 5 is minimal with respect to the iterative dominance frontier handles.

In Figure 9, we introduce an auxiliary function. $\mathsf{phiparam}\, v\, B\, \mathsf{CTX}\, \beta$ extracts a program context from the top level phi function in B. The extracted program context must be a non-join context associated with the value of v which is propagated down from v's (re)definition at location CTX. For clarity, we use V to denote a variable from SSAFJ and v to denote a variable from FJ. The interesting cases are the if-else block and the while block. In case of an if-else block, we check whether the top level set $\overline{\phi}$ contains one particular phi function that "merges" the (re)definitions of v. Depending on the input context is in the then- or else-branch, we extract the respective context from the phi operands if it is not a join context, otherwise, we apply the function recursively. In case of a while block, we examine the set of phi functions at the entry of the while block. The extraction is similar to the case of if-else. One subtle difference is that we do not need to extract the operand context coming from the preceding block, i.e. CTX_1. This is because the contexts extracted by $\mathsf{phiparam}$ function must be local to the current block. And the existence of the phi function at while block is determined by the existence of the (re)definition of the variable v coming from the loop body. Consider the code in Figure 2, applying $\mathsf{phiparam}$ to the if-else block at lines 8-21 to extract phi parameter associated with the variable i redefined at location line 13 yields (the SSAFJ context correspondent to) label 8.

The following lemma states that the all phi assignments from the SSAFJ are correspondent to some contexts found in iterative dominance frontier handle. Therefore, the SSAFJ is minimal.

LEMMA A.6 (SSAFJ IS MINIMAL). *Let s be an FJ statement such that $\mathbb{X}[\![s]\!] = \{\}$. Let β be a variable mapping. Let B be an SSA block. Let CTX be an SSAFJ program context such that $\square, \{\}, \square \vdash s \Rightarrow B, \beta, \mathsf{CTX}$ Let v be a variable such that $(v, \mathsf{ctx}', \mathsf{CTX}', V) \in \beta$ and CTX' is not a join context. Then $(\mathsf{phiparam}\, v\, B\, \mathsf{CTX}'\, \beta) \subseteq [\![fst(DF_h(\mathsf{ctx}' \mid s)^n)]\!]$.*

PROOF. Instead of directly proving this lemma, we prove a stronger version as follows.

Let s be an FJ statement. ctx be an FJ program context such that ctx denotes the program location of s. Let β, β' be variable mappings. Let B be an SSA block. Let $\mathsf{CTX}, \mathsf{CTX}'$ be SSAFJ program contexts such that $\mathsf{ctx}, \beta, \mathsf{CTX} \vdash s \Rightarrow B, \beta', \mathsf{CTX}'$.. Let v be a variable such that $(v, \mathsf{ctx}'', \mathsf{CTX}'', V) \in (\beta' - \beta)_{|\mathsf{ctx}}$ where CTX'' is not a join program context. Then $(\mathsf{phiparam}\, v\, B\, \mathsf{CTX}''\, (\beta' - \beta)_{|\mathsf{ctx}}) \subseteq [\![fst(DF_h(\mathsf{ctx}'' \mid s)^n)]\!]$.
We prove the lemma by induction over s.

- Case a. $\mathsf{phiparam}\, v\, A\, \mathsf{CTX}'\, \beta = \{\}$ trivial.
- Case return e. $\mathsf{phiparam}\, v\, (\mathsf{return}\, E)\, \mathsf{CTX}'\, \beta = \{\}$ trivial.
- Case if $e\, \{\overline{s_1}\}$ else$\{\overline{s_2}\}$. Recall that

$$
\begin{array}{c}
\mathsf{CTX} = [\![\mathsf{ctx}]\!] \quad \beta, \mathsf{CTX} \vdash e : E \\
\mathsf{ctx}[\mathsf{if}\, e\{\square\}\mathsf{else}\{\overline{\mathsf{s}}\}], \beta_0, \mathsf{CTX} \vdash \overline{s_1} \Rightarrow \overline{B_1}, \beta_1, \mathsf{CTX}_1 \\
\mathsf{ctx}[\mathsf{if}\, e\{\overline{\mathsf{s}}\}\mathsf{else}\{\square\}], \beta_0, \mathsf{CTX} \vdash \overline{s_2} \Rightarrow \overline{B_2}, \beta_2, \mathsf{CTX}_2 \\
\mathsf{CTX}_3 = \mathsf{CTX}[\mathsf{if}\, \mathbf{E}\, \{\overline{\mathbf{B}}\}\mathsf{else}\{\overline{\mathbf{B}}\}\mathsf{join}\, \{\blacksquare\}] \\
\overline{\phi} = \{x_{\mathsf{CTX}_3} = phi(\mathsf{CTX}_1 : \mathbb{R}_{\leq \mathsf{CTX}_1}(\beta_1, x), \mathsf{CTX}_2 : \mathbb{R}_{\leq \mathsf{CTX}_2}(\beta_2, x)) \\
\mid x \in dom3((\beta_1 \cup \beta_2) - \beta_3)\} \\
\beta_3 = \{(x, \mathsf{ctx}, \mathsf{CTX}_3, x_{\mathsf{CTX}_3}) \mid x \in dom3((\beta_1 \cup \beta_2) - \beta_0)\} \cup \\
\beta_0 \cup \beta_1 \cup \beta_2 \\
\hline
\mathsf{ctx}, \beta_0, \mathsf{CTX}_0 \vdash \mathsf{if}\, e\{\overline{s_1}\}\mathsf{else}\{\overline{s_1}\} \Rightarrow \\
\mathsf{CTX} : \mathsf{if}\, E\{\overline{B_1}\}\mathsf{else}\{\overline{B_2}\}\mathsf{join}\{\overline{\phi}\}, \beta_3, \mathsf{CTX}_3
\end{array}
$$

And the definition of $\mathsf{phiparam}$ for the if-else statement with some renaming

$$
\begin{array}{l}
\mathsf{phiparam}\, v\, (\mathsf{if}\, E\, \{\overline{B_1}\}\, \mathsf{else}\, \{\overline{B_2}\}\, \mathsf{join}\, \{\overline{\phi}\})\, \mathsf{CTX}''\, (\beta_2 - \beta_0) = \\
\quad \mathsf{case}\, (\mathsf{CTX}'', \overline{\phi})\, \mathsf{of} \\
\quad ((\mathsf{if}\, \mathbf{E}\, \{\square\}\mathsf{else}\{\overline{\mathbf{B}}\}\mathsf{join}\{\overline{\phi}\})[\mathsf{CTX}'''] \\
\quad , V = \mathsf{phi}(\, \mathsf{CTX}_1 : V_1,\, \mathsf{CTX}_2 : V_2); \overline{\phi'}) \\
\quad\quad \mid (v, \mathsf{ctx}''', \mathsf{CTX}''', V') \in (\beta_2 - \beta_0)_{|\mathsf{ife}\{\square\}\mathsf{else}\{\overline{\mathsf{s}}\}} \\
\quad\quad \wedge isJoin(\mathsf{CTX}_1) \quad\quad\quad\quad\quad\quad --(1.1) \\
\quad\quad \rightarrow \mathsf{phiparam}\, v\, \overline{B_1}\, \mathsf{CTX}'''\, (\beta_2 - \beta_0)_{|\mathsf{ife}\{\square\}\mathsf{else}\{\overline{\mathsf{s}}\}} \\
\quad\quad \mid (v, \mathsf{ctx}''', \mathsf{CTX}''', V') \in (\beta_2 - \beta_0)_{|\mathsf{ife}\{\square\}\mathsf{else}\{\overline{\mathsf{s}}\}} \wedge \\
\quad\quad \neg(isJoin(\mathsf{CTX}_1)) \quad\quad\quad\quad\quad\quad --(1.2) \\
\quad\quad \rightarrow \{\mathsf{CTX}_1\} \\
\quad (((\mathsf{if}\, E\, \{\overline{\mathbf{B}}\}\mathsf{else}\{\square\}\mathsf{join}\{\overline{\phi}\})[\mathsf{CTX}'''] \\
\quad , V = \mathsf{phi}(\, \mathsf{CTX}_1 : V_1,\, \mathsf{CTX}_2 : V_2); \overline{\phi'}) \\
\quad\quad \mid (v, \mathsf{ctx}''', \mathsf{CTX}''', V') \in (\beta_2 - \beta_0)_{|\mathsf{ife}\{\overline{\mathsf{s}}\}\mathsf{else}\{\square\}} \\
\quad\quad \wedge isJoin(\mathsf{CTX}_2) \quad\quad\quad\quad\quad\quad --(2.1) \\
\quad\quad \rightarrow \mathsf{phiparam}\, v\, \overline{B_2}\, \mathsf{CTX}'''\, (\beta_2 - \beta_0)_{|\mathsf{ife}\{\overline{\mathsf{s}}\}\mathsf{else}\{\square\}} \\
\quad\quad \mid (v, \mathsf{ctx}''', \mathsf{CTX}''', V') \in (\beta_2 - \beta_0)_{|\mathsf{ife}\{\overline{\mathsf{s}}\}\mathsf{else}\{\square\}} \\
\quad\quad \wedge \neg(isJoin(\mathsf{CTX}_2)) \quad\quad\quad\quad\quad --(2.2) \\
\quad\quad \rightarrow \{\mathsf{CTX}_2\}
\end{array}
$$

We consider the following sub-cases.

phiparam :: Var → BLOCK → CONTEXT → VarMap → {CONTEXT}
phiparam v (if E $\{\overline{B_1}\}$ else $\{\overline{B_2}\}$ join $\{\overline{\phi}\}$) CTX' β = case (CTX', $\overline{\phi}$) of
$\quad$ ((if **E** $\{\square\}$else$\{\overline{\mathbf{B}}\}$join$\{\overline{\phi}\}$)[CTX''], V = phi(CTX$_1$: V_1, CTX$_2$: V_2); $\overline{\phi'}$)
$\qquad$ | $(v, \text{ctx''}, \text{CTX''}, V') \in \beta_{|\text{ife}\{\square\}\text{else}\{\overline{s}\}} \wedge isJoin(\text{CTX}_1) \rightarrow$ phiparam $v \overline{B_1}$ CTX'' $\beta_{|\text{ife}\{\square\}\text{else}\{\overline{s}\}}$
$\qquad$ | $(v, \text{ctx''}, \text{CTX''}, V') \in \beta_{|\text{ife}\{\square\}\text{else}\{\overline{s}\}} \wedge \neg(isJoin(\text{CTX}_1)) \rightarrow \{\text{CTX}_1\}$
$\quad$ (((if **E** $\{\overline{\mathbf{B}}\}$else$\{\square\}$join$\{\overline{\phi}\}$)[CTX''], V = phi(CTX$_1$: V_1, CTX$_2$: V_2); $\overline{\phi'}$)
$\qquad$ | $(v, \text{ctx''}, \text{CTX''}, V') \in \beta_{|\text{ife}\{\overline{s}\}\text{else}\{\square\}} \wedge isJoin(\text{CTX}_2) \rightarrow$ phiparam $v \overline{B_2}$ CTX'' $\beta_{|\text{ife}\{\overline{s}\}\text{else}\{\square\}}$
$\qquad$ | $(v, \text{ctx''}, \text{CTX''}, V') \in \beta_{|\text{ife}\{\overline{s}\}\text{else}\{\square\}} \wedge \neg(isJoin(\text{CTX}_2)) \rightarrow \{\text{CTX}_2\}$
phiparam v (join $\{\overline{\phi}\}$ while E $\{\overline{B}\}$) CTX' β = case (CTX', $\overline{\phi}$) of
$\quad$ ((join$\{\overline{\phi}\}$whileE$\{\square\}$)[CTX''], V = phi(CTX$_1$: V_1, CTX$_2$: V_2); $\overline{\phi'}$)
$\qquad$ | $(v, \text{ctx''}, \text{CTX''}, V') \in \beta_{|\text{while e }\{\square\}} \wedge isJoin(\text{CTX}_2) \rightarrow$ phiparam $v \overline{B}$ CTX'' $\beta_{|\text{while e }\{\square\}}$
$\qquad$ | $(v, \text{ctx''}, \text{CTX''}, V') \in \beta_{|\text{while e }\{\square\}} \wedge \neg(isJoin(\text{CTX}_2)) \rightarrow \{\text{CTX}_2\}$
phiparam v _ _ _ = {}
phiparam :: Var → [BLOCK] → CONTEXT → VarMap → {CONTEXT}
phiparam v (B;) CTX' β = case CTX' of($\square$;)[CTX''] → phiparam v B CTX'' β
phiparam v (B;$\overline{B}$) CTX' β = case CTX'of
$\quad$ ($\square$;$\overline{B}$)[CTX''] | $(v, \text{ctx''}, \text{CTX''}, V') \in \beta_{|\square;\overline{B}} \rightarrow$ phiparam v B CTX'' $\beta_{|\square;\overline{B}}$
$\quad$ (B;$\square$)[CTX''] | $(v, \text{ctx''}, \text{CTX''}, V') \in \beta_{|B;\square} \rightarrow$ phiparam $v \overline{B}$ CTX'' $\beta_{|B;\square}$
$isJoin$(if **E** $\{\overline{\mathbf{B}}\}$else$\{\overline{\mathbf{B}}\}$ join $\{\blacksquare\}$) = $True$
$isJoin$(join $\{\blacksquare\}$ while **E** $\{\overline{\mathbf{B}}\}$) = $True$
$isJoin$(_) = $False$
$\beta_{|\text{ctx}} = \{(v, \text{ctx''}, \text{CTX''}, V') \mid (v, \text{ctx'}, \text{CTX'}, V') \in \beta \wedge \exists \text{ctx''}, \text{CTX''} : \text{ctx'} = \text{ctx}[\text{ctx''}] \wedge \text{CTX'} = [\![\text{ctx}]\!][\text{CTX''}]\}$

Figure 9: Phi Parameter Extraction

– Subcase (1.1). Note that $(\beta_2 - \beta_0)_{|\text{ife}\{\square\}\text{else}\{\overline{s}\}} = (\beta_1 - \beta_0)_{|\text{ife}\{\square\}\text{else}\{\overline{s}\}}$. By I.H, we find

$$(\text{phiparam } v \, \overline{B_1} \, \text{CTX'''} \, (\beta_1 - \beta_0)_{|\text{ife}\{\square\}\text{else}\{\overline{s}\}})$$
$$\subseteq$$
$$[\![fst(DF_h(\text{ctx'''}|\overline{s_1}))^n)]\!]$$

where ife$\{\overline{s}\}$else$\{\square\}$[ctx'''] = ctx''. By Definition 7 and Definition 9 we find that

$$(\text{if } \mathbf{e} \, \{\square\}\text{else}\{\overline{s}\})[DF_h(\text{ctx'''}|\overline{s_1})^n]$$
$$\subseteq$$
$$DF_h(\text{ctx''}|\text{if } e \, \{\overline{s_1}\}\text{else}\{\overline{s_2}\})^n$$

since the CFG from the LHS is a strict subset of the RHS's CFG, (similar observation applies to the set of exits).

– Subcase (1.2). Since CTX$_1$ is not a join context, it follows that $\mathbb{X}[\![\overline{s_1}]\!] = \{\text{ctx}_1\}$ such that

$$[\![\text{if } \mathbf{e} \, \{\square\} \, \text{else}\{\overline{s}\}[\text{ctx}_1]]\!] = \text{CTX}_1$$

We consider two possibilities.

* Sub-subcase $DF_h(\text{ctx'''}|\overline{s_1}) = \{\}$, i.e. ctx''' is dominating all its successors, including ctx$_1$. By Definition 7 and 9, we have

$$DF_h(\text{ctx''}|\text{if } e \, \{\overline{s_1}\}\text{else}\{\overline{s_2}\})^n =$$
$$\{<(\text{if } \mathbf{e} \, \{\square\}\text{else}\{\overline{s}\})[\text{ctx}_1], (\text{if } \mathbf{e} \, \{\overline{s}\}\text{else}\{\square\})[\text{ctx}_2]>$$
$$| \, \text{ctx}_2 \in \mathbb{X}[\![\overline{s_2}]\!]\}$$

Note that ctx$_2$ is not dominated by ctx''. We verified this case.

* Sub-subcase $DF_h(\text{ctx'''}|\overline{s_1}) \neq \{\}$, By Definition 7,

$$DF_h(\text{ctx'''}|\overline{s_1})^0 =$$
$$DF_h(\text{ctx'''}|\overline{s_1}) =$$
$$\{<\text{ctx}_1'', \text{ctx}_1'''>| \, (\text{ctx}_1'', \text{ctx}_2) \in \mathbb{G}[\![\overline{s_1}]\!]$$
$$\wedge (\text{ctx}_1''', \text{ctx}_2) \in \mathbb{G}[\![\overline{s_1}]\!]$$
$$\wedge \text{ctx}''' \gg \text{ctx}_1'' \wedge \neg(\text{ctx}''' \gg \text{ctx}_1''')\} \quad (3)$$
$$\cup$$
$$\{<\text{ctx}_1'', \text{ctx}_1'''>| \, \text{ctx}_1'' \in \mathbb{X}[\![\overline{s_1}]\!] \wedge \text{ctx}_2'' \in \mathbb{X}[\![\overline{s_2}]\!]$$
$$\wedge \text{ctx}''' \gg \text{ctx}_1'' \wedge \neg(\text{ctx}''' \gg \text{ctx}_1''')\} \quad (4)$$

Since $\mathbb{X}[\![\overline{s_1}]\!] = \{\text{ctx}_1\}$, (4) must be empty. Let $<\text{ctx}_1', \text{ctx}_1''>$ be the outermost dominance frontier handle in $DF_h(\text{ctx'''}|\overline{s_1})^n$, i.e. $(\text{ctx}_1', \text{ctx}_2') \in \mathbb{G}[\![s_1]\!]$ and $(\text{ctx}_1'', \text{ctx}_2') \in \mathbb{G}[\![s_1]\!]$ such that $DF_h(\text{ctx}_2'|s_1) = \{\}$. It follows that $\text{ctx}_2' \gg \text{ctx}_1$, recall ctx$_1$ is the exit context from $\overline{s_1}$. By Definition 7 and Definition 9, we have

$$DF_h(\text{ctx''}|\text{if } e \, \{\overline{s_1}\}\text{else}\{\overline{s_2}\})^n =$$
$$\{<(\text{if } \mathbf{e} \, \{\square\}\text{else}\{\overline{s}\})[\text{ctx}_1], (\text{if } \mathbf{e} \, \{\overline{s}\}\text{else}\{\square\})[\text{ctx}_2]>$$
$$| \, \text{ctx}_2 \in \mathbb{X}[\![\overline{s_2}]\!]\}$$

Note that ctx$_2$ is not dominated by ctx''. We verified this case.

– Subcase (2.1). Similar to Subcase (1.1).
– Subcase (2.2). Similar to Subcase (1.2).

- Case while $e \{\bar{s}\}$. Recall that

$$\text{CTX} = [\![\text{ctx}]\!] \quad \text{CTX}_1^{0,1} = \text{CTX}[\text{join } \{\blacksquare^{0,1}\}\text{while } \mathbf{E}\{\overline{\mathbf{B}}\}]$$
$$\beta_1, \text{CTX}_1^0 \vdash e \Rightarrow E$$
$$\overline{\phi} = \{x_{\text{CTX}_1} = phi(\text{CTX}_0 : \mathbb{R}_{\leq \text{CTX}_0}(\beta_0, x)) \mid x \in dom3(\beta_0)\}$$
$$\beta_1 = \{(x, \text{ctx}, \text{CTX}_1^0, x_{\text{CTX}_1}) \mid x \in dom3(\beta_0)\} \cup \beta_0$$
$$\text{ctx}[\text{while } \mathbf{e}\{\Box\}], \beta_1, \text{CTX}_1^0 \vdash \bar{s} \Rightarrow \overline{B}, \beta_2, \text{CTX}_2$$
$$\overline{\phi'} = \{x_{\text{CTX}1} = phi(\text{CTX}_0 : \mathbb{R}_{\leq \text{CTX}_0}(\beta_0, x), \text{CTX}_2 : \mathbb{R}_{\leq \text{CTX}_2}(\beta_2, x)) \mid$$
$$x \in dom3(\beta_2 - \beta_1)\}$$
$$\theta = \{x_{\text{CTX}_1}/(\mathbb{R}_{\leq \text{CTX}_0}(\beta_0, x)) \mid x \in dom3(\beta_0) - dom3(\beta_2 - \beta_1)\}$$
$$\beta_3 = \{(x, \text{ctx}, \text{CTX}_1^1, x_{\text{CTX}_1}) \in dom3(\beta_2 - \beta_1)\} \cup \beta_0 \cup (\beta_2 - \beta_1)$$
$$\overline{\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad}$$
$$\text{ctx}, \beta_0, \text{CTX}_0 \vdash \text{while } e\{\bar{s}\} \Rightarrow$$
$$\text{CTX} : \text{join } \{\overline{\phi'}\}\text{while } \theta(E)\{\theta(\overline{B})\}, \beta_3, \text{CTX}_1^1$$

And the definition of phiparam for while statement with some renaming

phiparam v (join $\{\overline{\phi}\}$ while $E \{\overline{B}\}$) CTX' $(\beta_3 - \beta_0) =$
  case $(\text{CTX}', \overline{\phi})$ of
    $((\text{join}\{\overline{\phi}\}\text{while}\mathbf{E}\{\Box\})[\text{CTX}'']$
    $, V = \text{phi}(\text{CTX}_1 : V_1, \text{CTX}_2 : V_2); \overline{\phi'})$
    $\mid (v, \text{ctx}'', \text{CTX}'', V') \in (\beta_3 - \beta_0)_{|\text{while } \mathbf{e} \{\Box\}}$
    $\wedge isJoin(\text{CTX}_2) \rightarrow \quad\quad\quad\quad\quad\quad -- (5.1)$
    phiparam $v \overline{B}$ CTX'' $(\beta_3 - \beta_0)_{|\text{while } \mathbf{e} \{\Box\}}$
    $\mid (v, \text{ctx}'', \text{CTX}'', V') \in (\beta_3 - \beta_0)_{|\text{while } \mathbf{e} \{\Box\}}$
    $\wedge \neg(isJoin(\text{CTX}_2)) \rightarrow \quad\quad\quad\quad -- (5.2)$
    $\{\text{CTX}_2\}$

Note that we are only interested in variables $v \in dom3(\beta_2 - \beta_1)$, i.e. those variables are updated in the while loop body. For these variables we have $v \notin dom(\theta)$. We consider the following sub-cases.

– Subcase (5.1). Note that

$$(\beta_3 - \beta_0)_{|\text{while } \mathbf{e} \{\Box\}} =$$
$$(\beta_3)_{|\text{while } \mathbf{e} \{\Box\}} =$$
$$(\beta_2 - \beta_1)_{|\text{while } \mathbf{e} \{\Box\}}$$

since there is no entry in β_0 containing FJ context of shape (while $\mathbf{e}\{\Box\}[\text{ctx}']$). By I.H, we find

$$(\text{phiparam } v \overline{B} \text{ CTX}''' (\beta_2 - \beta_1)_{|\text{while } \mathbf{e} \{\Box\}})$$
$$\subseteq$$
$$[\![fst(DF_h(\text{ctx}'''|\bar{s}))^n)]\!]$$

where (while $\mathbf{e} \{\Box\})[\text{ctx}'''] = \text{ctx}''$.
By Definition 7 and Definition 9 we find that

$$(\text{while } \mathbf{e} \{\Box\})[DF_h(\text{ctx}'''|\bar{s})^n] \subseteq DF_h(\text{ctx}''|\text{while } e \{\bar{s}\})^n$$

  since the CFG from the LHS is a strict subset of the RHS's CFG, (similar observation applies to the set of exits).
– Subcase (5.2). Since CTX_2 is not a join context, it follows that $\mathbb{X}[\![\bar{s}]\!] = \{\text{ctx}_2\}$ such that

$$[\![\text{while } \mathbf{e} \{\Box\}[\text{ctx}_2]]\!] = \text{CTX}_2$$

  The remain steps of proving this subcase is similar to the ones found in the Subcase (1.2) in the if-else statement.

- Case $s; \bar{s}$. Recall that

$$\frac{\text{ctx}[\Box; \bar{s}], \beta, \text{CTX} \vdash S \Rightarrow B, \beta', \text{CTX}' \quad \text{ctx}[s; \Box], \beta', \text{CTX}' \vdash \bar{s} \Rightarrow \overline{B}, \beta'', \text{CTX}''}{\text{ctx}, \beta, \text{CTX} \vdash s; \bar{s} \Rightarrow B; \overline{B}, \beta'', \text{CTX}''}$$

And

phiparam v $(B; \overline{B})$ CTX' $(\beta_2 - \beta) =$
  case CTX' of
    $(\Box; \overline{B})[\text{CTX}'']$
    $\mid (v, \text{ctx}'', \text{CTX}'', V') \in (\beta_2 - \beta)_{|\Box; \overline{B}} \rightarrow \quad -- (6.1)$
    phiparam v B CTX'' $(\beta_2 - \beta)_{|\Box; \overline{B}}$
    $(B; \Box)[\text{CTX}'']$
    $\mid (v, \text{ctx}'', \text{CTX}'', V') \in (\beta_2 - \beta)_{|B; \Box} \rightarrow \quad -- (6.2)$
    phiparam v $\overline{B}$ CTX'' $(\beta_2 - \beta)_{|B; \Box}$

We consider the following sub-cases
– Subcase (6.1). Note that

$$(\beta_2 - \beta)_{|\Box; \overline{B}} = (\beta_1 - \beta)_{|\Box; \overline{B}}$$

since we accumulate the variable mapping entry over sequence during the SSA construction. By I.H. we have

phiparam v B CTX'' $(\beta_1 - \beta)_{|\Box; \overline{B}} \subseteq [\![fst(DF_h(\text{ctx}''|s))^n)]\!]$

  where $(\Box; \bar{s})[\text{ctx}''] = \text{ctx}'$. By Definition 7 and Definition 9 we find that

$$(\Box; \overline{B})[DF_h(\text{ctx}''|s)^n] \subseteq DF_h(\text{ctx}'|s; \bar{s})^n$$

  since the CFG from the LHS is a strict subset of the RHS's CFG, (similar observation applies to the set of exits).
– Subcase (6.2). Note that

$$(\beta_2 - \beta)_{|B; \Box} = (\beta_2 - \beta_1)_{|B; \Box}$$

since we accumulate the variable mapping entry over sequence during the SSA construction By I.H. we have

phiparam v $\overline{B}$ CTX'' $(\beta_2 - \beta_1)_{|B; \Box} \subseteq [\![fst(DF_h(\text{ctx}''|\bar{s}))^n)]\!]$

  where $(s; \Box)[\text{ctx}''] = \text{ctx}'$. By Definition 7 and Definition 9 we find that

$$(B; \Box)[DF_h(\text{ctx}''|\bar{s})^n] \subseteq DF_h(\text{ctx}'|s; \bar{s})^n$$

  since the CFG from the LHS is a strict subset of the RHS's CFG, (similar observation applies to the set of exits).

$\square$

B APPLICATION

B.1 Unrolling Structured SSA into GCL

In Figure 10, we report syntax of GCL statement adopted from [10]. We define *nop* as a syntactic sugar of *assume true*.

In the same figure, we present the unrolling algorithm that translates SSAFJ into GCL. For brevity, we exclude the statement of method call. We also omit the contexts associated with the operands in the phi functions. We extend SSAFJ while loop to include the invariant I. Function $\text{gcl}[\![\cdot]\!]$ converts a SSAFJ statement block into a GCL statement recursively. Function $\text{expand}[\![\cdot]\!]$ unrolls the loop with the fixed number of times. Applying $\text{gcl}[\![\cdot]\!]$ 1 to the running example in Figure 1 (c), we have

(GCL) $\mathcal{S}$ $::=$ $x = E \mid assert\ E \mid assume\ E \mid \mathcal{S}; \mathcal{S} \mid \mathcal{S}\square\mathcal{S}$

$\mathtt{gcl}[\![\cdot]\!] :: \mathtt{STATEMENT} \to Int \to \mathtt{GCL}$
$\mathtt{gcl}[\![\mathtt{if}\ E\{\overline{S_1}\}\mathtt{else}\{\overline{S_2}\}\ \mathtt{join}\{\overline{\phi}\}]\!]\ n =$
   $(assume\ E; \mathtt{gcl}[\![S_1]\!]\ n; \mathtt{fsta}[\![\overline{\phi}]\!])\square(assume\ \neg E; \mathtt{gcl}[\![S_2]\!]\ n;$
   $\mathtt{snda}[\![\overline{\phi}]\!])$
$\mathtt{gcl}[\![\mathtt{join}\{\overline{\phi}\}\ \mathtt{while}\ E\ I\{\overline{S}\}]\!]\ n =$
   $\mathtt{let}\ \overline{S} = \mathtt{gcl}[\![\overline{S}]\!]\ n\ \mathtt{in}\ \mathtt{expand}[\![\overline{\phi}]\!]\ E\ I\ \overline{S}\ n$
$\mathtt{gcl}[\![x = E]\!]\ n = x = E$
$\mathtt{gcl}[\![\mathtt{return}\ E]\!]\ n = assume\ true$

$\mathtt{expand}[\![\cdot]\!] :: [\mathtt{Phi}] \to \mathtt{EXPRESSION} \to \mathtt{EXPRESSION}$
   $\to \mathtt{GCL} \to Int \to \mathtt{GCL}$
$\mathtt{expand}[\![\overline{\phi}]\!]\ E\ I\ \mathcal{S}\ 0 = assert\ false;$
$\mathtt{expand}[\![\overline{\phi}]\!]\ E\ I\ \mathcal{S}\ n =$
   $\mathtt{let}\ \theta_1 = \mathtt{subst}_1[\![\overline{\phi}]\!]\ n;\ \theta_2 = \mathtt{subst}_2[\![\overline{\phi}]\!]\ \theta_1$
   $\mathtt{in}(\theta_1(\mathtt{fsta}[\![\overline{\phi}]\!]; assume\ E \wedge I; \mathcal{S}); \mathtt{expand}[\![\theta_2(\overline{\phi})]\!]\ E\ I\ \mathcal{S}\ (n-1))\ \square$
   $(\mathtt{fsta}[\![\overline{\phi}]\!]; assume\neg(E) \wedge I;)$

$\mathtt{fsta}[\![\cdot]\!] :: [\mathtt{Phi}] \to [\mathtt{GCL}]$
$\mathtt{fsta}[\![\{\}]\!] = nop$
$\mathtt{fsta}[\![x = phi(x_1, x_2); \overline{\phi}]\!] = x = x_1; \mathtt{fsta}[\![\overline{\phi}]\!]$

$\mathtt{snda}[\![\cdot]\!] :: [\mathtt{Phi}] \to [\mathtt{GCL}]$
$\mathtt{snda}[\![\{\}]\!] = nop$
$\mathtt{snda}[\![x = phi(x_1, x_2); \overline{\phi}]\!] = x = x_2; \mathtt{snda}[\![\overline{\phi}]\!]$

$\mathtt{subst}_1[\![\cdot]\!] :: [\mathtt{Phi}] \to Int \to \{(\mathtt{Var}, \mathtt{Var})\}$
$\mathtt{subst}_1[\![\{\}]\!]\ n = [\,]$
$\mathtt{subst}_1[\![x = phi(x_1, x_2); \overline{\phi}]\!] = [x^n/x, x_2^n/x_2] \cup \mathtt{subst}_1[\![\overline{\phi}]\!]\ n$

$\mathtt{subst}_2[\![\cdot]\!] :: [\mathtt{Phi}] \to \{(\mathtt{Var}, \mathtt{Var})\} \to \{(\mathtt{Var}, \mathtt{Var})\}$
$\mathtt{subst}_2[\![\{\}]\!]\ \theta_1 = [\,]$
$\mathtt{subst}_2[\![x = phi(x_1, x_2); \overline{\phi}]\!]\ \theta_1 = [\theta_1(x_2)/x_1] \cup \mathtt{subst}_2[\![\overline{\phi}]\!]\ \theta_1$

Figure 10: Unrolling Structured SSA into GCL

$s_0 = 0;\ i_0 = 0;$
$\theta_1(i_1 = i_0; s_1 = s_0; assume\ (i_1 < N) \wedge s_1 <= i_1 * (i_1 - 1)/2;$
$s_2 = s_1 + i_1; i_2 = i_1 + 1); assert\ false;$
$\square\ i_1 = i_0; s_1 = s_0; assume\ (i_1 >= N) \wedge s_1 <= i_1 * (i_1 - 1)/2;$
   where $\theta_1 = [i_1^1/i_1, s_1^1/s_1, i_2^1/i_2, s_2^1/s_2]$

By applying the substitution, we obtain the GCL snippet reported in Section 1. The algorithm is visit each node in the AST once except for $\mathtt{expand}[\![\cdot]\!]$ which depends on n. Overall time complexity is $O(n * s)$ where s is the size of the program.

We define single assignment as follows

DEFINITION 10 (SINGLE ASSIGNMENT).

$$sa(x = E)\quad sa(assume\ E)\quad sa(assert\ E)$$
$$\frac{sa(\mathcal{S}_1)\quad sa(\mathcal{S}_2)}{lhs(\mathcal{S}_1) \cap lhs(\mathcal{S}_2) = \{\}}\qquad \frac{sa(\mathcal{S}_1)\quad sa(\mathcal{S}_2)}{sa(\mathcal{S}_1\square\mathcal{S}_2)}$$
$$sa(\mathcal{S}_1; \mathcal{S}_2)$$

where

$lhs(x = E) = \{x\}\quad lhs(assert\ E) = \{\}\quad lhs(assume\ E) = \{\}$
$lhs(\mathcal{S}_1; \mathcal{S}_2) = \bigcup_{i \in \{1,2\}} lhs(\mathcal{S}_i)\quad lhs(\mathcal{S}_1\square\mathcal{S}_2) = \bigcup_{i \in \{1,2\}} lhs(\mathcal{S}_i)$

LEMMA B.1 (UNROLLING YIELDS SA). *Let S be an SSAFJ statement, n be a non-negative integer and $\mathcal{S}$ be a GCL statement such that $\mathtt{gcl}[\![S]\!]\ n = \mathcal{S}$. Then $sa(\mathcal{S})$.*

PROOF. We prove by induction of S and n.   □

B.2 Time Complexity Analysis of Using SSA with Unrolling in GCL conversion

In this section, we conduct an analysis of comparing Unroll-then-SSA pipeline against SSA-before-Unroll pipeline in GCL construction.

B.2.1 Unroll-then-SSA. Given a source program with L number of loops, I if-else and V variables, the time complexity of the unrolling the source program with k iterations is bounded by $O(k^L \cdot I \cdot V)$, assuming the worst case we have L nested loops. After the unrolling, there is no more loop, and $k^L \cdot I$ if-elses. ' The time complexity is $O((k^L \cdot I)^2 \cdot V)$. The total time complexity is $O(k^L \cdot I \cdot V + k^{2L} \cdot I^2 \cdot V)$.

B.2.2 SSA-then-Unroll. Starting with SSA construction, the time complexity is $O((L + I)^2 \cdot V)$. The resulting SSA is of size $(L + I) \cdot V$. Applying the unrolling on the SSA with k iterations, the time complexity is bounded by $O(K^L \cdot I \cdot V \cdot V)$ since the number of loops does not change after SSA construction. The total time complexity is $O((L + I)^2 \cdot V + K^L \cdot I \cdot V \cdot V)$.

Comparing the two approaches above, we find that SSA-then-Unroll is more efficient when k^L is large, i.e. when the program has more nested loops. The time complexity of SSA construction is dependent on the size of the program, the time complexity of the unrolling operation is dependent on the number loops. Applying SSA construction does not change the number of loops, while applying unrolling first will increase the size of the program. Hence applying SSA construction before the unrolling allows us to construct the SSA once and use it for different unrolling configurations.

B.3 Flattening Structured SSA into Unstructured SSA

We formalize the syntax of unstructured SSA and the flattening algorithm in Figure 11. We omit the semantics of unstructured SSA which can be found in the literature. The $\mathtt{flatten}[\![\cdot]\!]$ function takes a sequence of SSAFJ statement blocks, a set of phi assignments and a control flow graph of the given SSAFJ program, returns a sequence of unstructured SSA statements and the trailing phi assignments. We omit the details of the construction of the control flow graph from a given SSAFJ program, which is similar the one constructing from FJ programs found in Figure 8. In case of a leading while statement block, function $\mathtt{flatten}[\![\cdot]\!]$ is applied recursively to the body of the loop as well as the statement blocks that follow. The trailing phi assignments from the recursive call are merged with the entry phi assignments of the while loop. Helper function $params(\overline{\phi})$ returns the set of parameters in $\overline{\phi}$. The loop is translated into a conditional jump with the condition negated. Helper function $\mathtt{succs}$ finds the set of non-join successors of the given context(s). The intuition behind is that all join contexts are merged into the non-join contexts during the flattening operation. The helper function $\mathtt{nnsuccs}$ finds the set of nearest non-nested non-join successor of CTX. Function $\mathtt{nnsucc}$ is partial and expects the

$$\begin{array}{rcl}
\text{(PhiSubst)} & \vartheta & ::= & \overline{[(\mathsf{CTX}_1 : x_1, ..., \mathsf{CTX}_n : x_n)/(\mathsf{CTX} : y)]} \\
\text{(UnSSA)} & U & ::= & x = E \mid \text{return } E \mid \text{if } E \text{ goto CTX} \mid \\
& & & \text{goto CTX} \mid \mathsf{CTX} : \overline{\phi}
\end{array}$$

$\mathsf{flatten}[\![\cdot]\!] :: [\mathsf{BLOCK}] \to [\mathsf{Phi}] \to \{(\mathsf{CONTEXT}, \mathsf{CONTEXT})\} \to$
  $\quad (\mathsf{UnSSA}, [\mathsf{Phi}])$
$\mathsf{flatten}[\![\mathsf{CTX} : \{\mathsf{join}\,\{\overline{\phi'}\}\mathsf{while}\,E\,\{\overline{B}\}\}; \overline{B'}]\!]\,\overline{\phi}\,G =$
  $\quad \text{let } \{(\overline{U}, \overline{\phi''}) = \mathsf{flatten}[\![\overline{B}]\!]\,[\,]\,G; \ (\overline{U'}, \overline{\phi'''}) = \mathsf{flatten}[\![\overline{B'}]\!]\,[\,]\,G\}$
  $\quad \text{in } (\mathsf{CTX} : \mathsf{merge}\,\overline{\phi}\,(\mathsf{merge}\,\overline{\phi''}\,\overline{\phi'}); \mathsf{if}\,\neg(E)\,\mathsf{goto}\,(\mathsf{nnsucc}\,G\,\mathsf{CTX});$
  $\quad \overline{U}; \mathsf{goto}\,\mathsf{CTX}; \overline{U'}, \overline{\phi'''})$
$\mathsf{flatten}[\![\mathsf{CTX} : \{\mathsf{if}\,E\,\{\overline{B_1}\}\mathsf{else}\{\overline{B_2}\}\,\mathsf{join}\{\overline{\phi'}\}\}; \overline{B_3}]\!]\,\overline{\phi}\,G =$
  $\quad \text{let } \{(\overline{U_1}, \overline{\phi_1}) = \mathsf{flatten}[\![\overline{B_1}]\!]\,[\,]\,G; \ (\overline{U_2}, \overline{\phi_2}) = \mathsf{flatten}[\![\overline{B_2}]\!]\,[\,]\,G;$
  $\quad \mathsf{CTX}_2 = \mathsf{CTX}[\mathsf{if}\,\mathbf{E}\,\{\overline{\mathbf{B}}\}\mathsf{else}\{\Box\}\,\mathsf{join}\{\overline{\phi}\}];$
  $\quad \{\mathsf{CTX}_3\} = (\mathsf{succs}\,G\,\mathbb{X}[\![\overline{B_1}]\!]) \cap (\mathsf{succs}\,G\,\mathbb{X}[\![\overline{B_2}]\!]);$
  $\quad \overline{\phi_3} = \mathsf{merge}\,\overline{\phi_1}\,(\mathsf{merge}\,\overline{\phi_2}\,\overline{\phi'}); \ (\overline{U_4}, \overline{\phi_4}) = \mathsf{flatten}[\![\overline{B_3}]\!]\,\overline{\phi_3}\,G\}$
  $\quad \text{in } (\mathsf{CTX} : \overline{\phi}; \mathsf{if}\,\neg(E)\,\mathsf{goto}\,\mathsf{CTX}_2; \overline{U_1}; \mathsf{goto}\,\mathsf{CTX}_3; \overline{U_2}; \overline{U_4}, \overline{\phi_4})$
$\mathsf{flatten}[\![\mathsf{CTX} : x = E; \overline{B}]\!]\,\overline{\phi}\,G =$
  $\quad \text{let } (\overline{U}, \overline{\phi'}) = \mathsf{flatten}[\![\overline{B}]\!]\,[\,]\,(\mathsf{CTX} : \overline{\phi}; x = E;)$
  $\quad \text{in } (\mathsf{CTX} : \overline{\phi}; x = E; \overline{U}, \overline{\phi'})$
$\mathsf{flatten}[\![\mathsf{CTX} : \mathsf{return}\,E;]\!]\,\overline{\phi}\,G = (\mathsf{CTX} : \overline{\phi}; \mathsf{return}\,E, [\,])$

$\mathsf{merge} :: [\mathsf{Phi}] \to [\mathsf{Phi}] \to [\mathsf{Phi}]$
$\mathsf{merge}\,\overline{\phi_1}\,\overline{\phi_2} =$
  $\quad \text{let } \vartheta_1 = [(\mathsf{CTX}_1 : x_1, ..., \mathsf{CTX}_n : x_n)/\mathsf{CTX'} : x \mid$
  $\quad\quad x = \underline{phi}(\mathsf{CTX}_1 : x_1, ..., \mathsf{CTX}_n : x_n) \in \overline{\phi_1}) \wedge (\mathsf{CTX'} : x) \in params(\overline{\phi_2})]$
  $\quad \text{in } \vartheta_1(\overline{\phi_2})$

$\mathsf{succs} :: \{(\mathsf{CONTEXT}, \mathsf{CONTEXT})\} \to \mathsf{CONTEXT} \to \{\mathsf{CONTEXT}\}$
$\mathsf{succs}\,G\,\mathsf{CTX} = \text{let } cs_\Box = \{\mathsf{CTX'} \mid (\mathsf{CTX}, \mathsf{CTX'}) \in G \wedge \neg isJoin(\mathsf{CTX'})\}$
  $\quad cs_\blacksquare = \{\mathsf{CTX'} \mid (\mathsf{CTX}, \mathsf{CTX'}) \in G \wedge isJoin(\mathsf{CTX'})\}$
  $\quad \text{in } cs_\Box \cup (\mathsf{succs}\,G\,cs_\blacksquare)$

$\mathsf{succs} :: \{(\mathsf{CONTEXT}, \mathsf{CONTEXT})\} \to \{\mathsf{CONTEXT}\} \to \{\mathsf{CONTEXT}\}$
$\mathsf{succs}\,G\,cs = \{\mathsf{CTX'} \mid \mathsf{CTX} \in cs \wedge \mathsf{CTX'} \in \mathsf{succs}\,G\,\mathsf{CTX}\}$

$\mathsf{nnsuccs} :: \{(\mathsf{CONTEXT}, \mathsf{CONTEXT})\} \to \mathsf{CONTEXT} \to \{\mathsf{CONTEXT}\}$
$\mathsf{nnsuccs}\,G\,\mathsf{CTX} =$
  $\quad \{\mathsf{CTX'} \mid \mathsf{CTX'} \in \mathsf{succs}\,G\,\mathsf{CTX} \wedge \neg(\exists \mathsf{CTX''}.\mathsf{CTX}[\mathsf{CTX''}] = \mathsf{CTX'})\}$

$\mathsf{nnsucc} :: \{(\mathsf{CONTEXT}, \mathsf{CONTEXT})\} \to \mathsf{CONTEXT} \to \mathsf{CONTEXT}$
$\mathsf{nnsucc}\,G\,\mathsf{CTX} = \mathsf{case}\,\mathsf{nnsuccs}\,G\,\mathsf{CTX}\,\mathsf{of}\{\mathsf{CTX'}\} \to \mathsf{CTX'}$

Figure 11: Flattening SSAFJ into Unstructured SSA

result set is singleton. It is safe because all while loop has exactly one non-nested successor. In case of a leading if statement block, we apply $\mathsf{flatten}[\![\cdot]\!]$ recursively to the then- and else-branches. We merge the trailing phi assignments from both branches to $\overline{\phi'}$. The cases of a leading assignment statement block and the return statement are trivial. We omit the other cases which are similar to the first three cases. For instance, applying the flattening algorithm to the SSAFJ found in Figure 1 (c) yields the unstructured SSA found in Figure 1 (b).

LEMMA B.2 (FLATTENING IS CONSISTENT). *Let $\overline{B}$ be an SSAFJ program. Let G be a control flow graph constructed from $\overline{B}$. Let $\overline{U}$ be an unstructured SSA program obtained from $\mathsf{flatten}[\![\overline{B}]\!]\,[\,]\,G$ Then $\overline{U}$ is semantically consistent with $\overline{B}$.*

C PROOF SKETCH OF FORMAL RESULTS FOUND IN SECTION B.3

C.1 Extra Definitions

In Figure 12, we define the control flow graph construction algorithm for SSAFJ. Most of the cases are similar to the the one for FJ except for the following

- In case of while statement and if-else statement the $\mathbb{X}[\![\cdot]\!]$ function returns the phi contexts of the blocks.
- In case of if-else statement, the $\mathbb{G}[\![\cdot]\!]$ function includes the additional edges from the exits of the then- and the else-branches to the phi context.
- In case of while statement, the $\mathbb{G}[\![\cdot]\!]$ function generates the following edges
  - from the current statement to the phi context,
  - from the phi context to the body of the while statement,
  - from exits of the while body to the phi context,
  - and those edges arising from the while body.

In Figure 13, we report the dynamic semantics of unstructured SSA.

The following lemma states that the result of the merge operation is consistent with its input.

LEMMA C.1 (MERGE IS CONSISTENT). *Let $lenv \in \mathsf{LEnv}$. Let $\mathsf{merge}\,\overline{\phi_1}\,\overline{\phi_2} = \overline{\phi_3}$ and $\forall x = \mathsf{phi}(\overline{\mathsf{CTX} : x}) \in \overline{\phi_1}$ we have $(\mathsf{CTX}_2 : x) \in params(\overline{\phi_2})$ and $\mathsf{CTX}_1 \in \overline{\mathsf{CTX}}$. Then $\mathbb{F}_{\mathsf{ssa}}[\![\overline{\phi_3}]\!]\,\mathsf{CTX}_1\,lenv = \mathbb{F}_{\mathsf{ssa}}[\![\overline{\phi_2}]\!]\,\mathsf{CTX}_2\,(\mathbb{F}_{\mathsf{ssa}}[\![\overline{\phi_1}]\!]\,\mathsf{CTX}_1\,lenv)$*

PROOF. By Definition of $\mathbb{F}_{\mathsf{ssa}}[\![\cdot]\!]$ and merge. □

DEFINITION 11 (PRECEDED JOIN RELATION). *We define $\mathsf{CTX}_1 \overset{G}{\prec} \mathsf{CTX}_2$ iff*

1. *$(\mathsf{CTX}_1, \mathsf{CTX}_2) \in G$ and $(\mathsf{succs}\,\mathsf{CTX}_1\,G) = (\mathsf{succs}\,\mathsf{CTX}_2\,G)$; or*
2. *there exists CTX_3 such that $\mathsf{CTX}_1 \overset{G}{\prec} \mathsf{CTX}_3$ and $\mathsf{CTX}_3 \overset{G}{\prec} \mathsf{CTX}_2$.*

Intuitively speaking, $\mathsf{CTX}_1 \overset{G}{\prec} \mathsf{CTX}_2$ means that CTX_2 is a join context and CTX_1 is a preceding context where there is no non-join context going in between them.

DEFINITION 12 (PRECEDED-EQUAL JOIN RELATION). *We define $\mathsf{CTX}_1 \overset{G}{\preceq} \mathsf{CTX}_2$ iff $\mathsf{CTX}_1 \overset{G}{\prec} \mathsf{CTX}_2$ or $\mathsf{CTX}_1 = \mathsf{CTX}_2$.*

C.2 Proof of Lemma B.2

PROOF. We prove a formalized version of Lemma B.2.

Let $\overline{B}$ be an SSAFJ program. Let G be a control flow graph constructed from $\overline{B}$. Let $genv \in \mathsf{GENV}$ and $lenv_s \in \mathsf{LEnv}$ and $lenv_u \in \mathsf{LEnv}$. Let str be a memory store. Let $\overline{\phi}$ and $\overline{\phi'}$ be phi assignments. Let $\mathsf{flatten}[\![\overline{B}]\!]\,\overline{\phi}\,G = (\overline{U}, \overline{\phi'})$. Let $it = \mathsf{buildIT}\,\overline{U}$. Let CTX_s and CTX_u be SSAFJ program contexts such that $\mathsf{CTX}_u \overset{G}{\preceq} \mathsf{CTX}_s$ and $lenv_s = \mathbb{F}_{\mathsf{ssa}}[\![\overline{\phi}]\!]\,\mathsf{CTX}_u\,lenv_u$. Then $\overline{\mathbb{B}}_{\mathsf{ssa}}[\![\overline{B}]\!]\,\mathsf{CTX}_s\,genv\,lenv_s\,str = (v_s, lenv'_s, str'_s, \mathsf{CTX}'_s)$ implies that

$$\mathbb{U}[\![\overline{U}]\!]\,genv\,lenv\,str\,m = (v_u, lenv'_u, str'_u, \mathsf{CTX}'_u)$$

such that $v_s = v_u$, $lenv'_s = \mathbb{F}_{\mathsf{ssa}}[\![\overline{\phi'}]\!]\,\mathsf{CTX}'_u\,lenv'_u$ and $str' = str''$ and $\mathsf{CTX}'_u \overset{G}{\preceq} \mathsf{CTX}'_s$.

$\mathbb{G}[\![\cdot]\!] :: \text{STATEMENT} \rightarrow \{(\text{CONTEXT}, \text{CONTEXT})\}$

$\mathbb{G}[\![A]\!] = \{\}$

$\mathbb{G}[\![\text{return } E]\!] = \{\}$

$\mathbb{G}[\![\text{if } E\{\overline{B_1}\}\text{else}\{\overline{B_2}\} \text{ join}\{\overline{\phi}\}]\!] = \text{let } \{G_1 = \mathbb{G}[\![\overline{B_1}]\!]; G_2 = \mathbb{G}[\![\overline{B_2}]\!]\}$
$\quad \text{in } \{(\square, \text{if } E \{\square\}\text{else}\{\mathbf{B}\}\text{join}\{\overline{\phi}\}), (\square, \text{if } E \{\mathbf{B}\}\text{else}\{\square\})\text{join}\{\overline{\phi}\}\} \cup$
$\quad (\text{if } E\{\square\}\text{else}\{\overline{\mathbf{B}}\}\text{join}\{\overline{\phi}\})[G_1] \cup (\text{if } E\{\overline{\mathbf{B}}\}\text{else}\{\square\}\text{join}\{\overline{\phi}\})[G_2] \cup$
$\quad \{(\text{if } E\{\square\}\text{else}\{\overline{\mathbf{B}}\}\text{join}\{\overline{\phi}\}[\text{CTX}], \text{if } E\{\mathbf{B}\}\text{else}\{\overline{\mathbf{B}}\}\text{join}\{\blacksquare\}) \mid \text{CTX} \in \mathbb{X}[\![\overline{B_1}]\!]\} \cup$
$\quad \{(\text{if } E\{\overline{\mathbf{B}}\}\text{else}\{\square\}\text{join}\{\overline{\phi}\}[\text{CTX}], \text{if } E\{\mathbf{B}\}\text{else}\{\overline{\mathbf{B}}\}\text{join}\{\blacksquare\}) \mid \text{CTX} \in \mathbb{X}[\![\overline{B_2}]\!]\}$

$\mathbb{G}[\![\text{join}\{\overline{\phi}\}\text{while } E \{\overline{B}\}]\!] = \text{let } \{G = \mathbb{G}[\![\overline{B}]\!]; O = \mathbb{X}[\![\overline{B}]\!]\}$
$\quad \text{in } \{(\square, \text{join } \{\blacksquare\}\text{while } E \{\overline{\mathbf{B}}\}), (\text{join } \{\blacksquare\}\text{while } E \{\overline{\mathbf{B}}\}, \text{join } \{\overline{\phi}\}\text{while } E \{\square\})\} \cup$
$\quad (\text{join } \{\overline{\phi}\}\text{while } E \{\square\})[G] \cup \{((\text{join } \{\overline{\phi}\}\text{while } E \{\square\})[\text{CTX}], \text{join } \{\blacksquare\}\text{while } E \{\overline{\mathbf{B}}\}) \mid \text{CTX} \in O\}$

$\mathbb{G}[\![\cdot]\!] :: \text{BLOCK} \rightarrow \{(\text{CONTEXT}, \text{CONTEXT})\}$

$\mathbb{G}[\![l : \{S\}]\!] = \mathbb{G}[\![S]\!]$

$\mathbb{G}[\![\cdot]\!] :: [\text{BLOCK}] \rightarrow \{(\text{CONTEXT}, \text{CONTEXT})\}$

$\mathbb{G}[\![B;]\!] = \mathbb{G}[\![B]\!]$

$\mathbb{G}[\![B_1; \overline{B_2}]\!] = \text{let } \{G_1 = \mathbb{G}[\![B_1]\!]; G_2 = \mathbb{G}[\![\overline{B_2}]\!]; O_1 = \mathbb{X}[\![B_1]\!]\}$
$\quad \text{in } \{(\square, (\square; \overline{\mathbf{B}}))\} \cup (\square; \overline{\mathbf{B}})[G_1] \cup (\mathbf{B}; \square)[G_2] \cup \{((\square; \overline{\mathbf{B}})[\text{CTX}], (\mathbf{B}; \square)) \mid \text{CTX} \in O_1\}$

$\mathbb{X}[\![\cdot]\!] :: \text{STATEMENT} \rightarrow \{\text{CONTEXT}\}$

$\mathbb{X}[\![A]\!] = \{\square\}$

$\mathbb{X}[\![\text{return } E]\!] = \{\}$

$\mathbb{X}[\![\text{if } E\{\overline{B_1}\}\text{else}\{\overline{B_2}\}\text{join}\{\overline{\phi}\}]\!] = \{\text{if } E \{\overline{\mathbf{B}}\}\text{else}\{\overline{\mathbf{B}}\}\text{join}\{\blacksquare\}\}$

$\mathbb{X}[\![\text{join } \{\overline{\phi}\}\text{while } E \{\overline{B}\}]\!] = \{\text{join } \{\blacksquare\}\text{while } E\{\overline{\mathbf{B}}\}\}$

$\mathbb{X}[\![\cdot]\!] :: \text{BLOCK} \rightarrow \{\text{CONTEXT}\}$

$\mathbb{X}[\![l : \{S\}]\!] = \mathbb{X}[\![S]\!]$

$\mathbb{X}[\![\cdot]\!] :: [\text{BLOCK}] \rightarrow \{\text{CONTEXT}\}$

$\mathbb{X}[\![B;]\!] = \mathbb{X}[\![B]\!]$

$\mathbb{X}[\![B_1; \overline{B_2}]\!] = \{\mathbf{B}; \text{CTX} \mid \text{CTX} \in \mathbb{X}[\![\overline{B_2}]\!]\}$

Figure 12: CFG construction of SSAFJ

$\mathbb{U}[\![\cdot]\!] :: [\text{UnSSA}] \rightarrow \text{CONTEXT} \rightarrow \text{GEnv} \rightarrow \text{LEnv} \rightarrow \text{Store} \rightarrow \{(\text{CONTEXT}, [\text{UnSSA}])\} \rightarrow (\text{Value}, \text{LEnv}, \text{Store}, \text{CONTEXT})$

$\mathbb{U}[\![\text{return } E;]\!] \text{ CTX } genv\ lenv\ str\ it =$
$\quad \text{let } (V, str') = \mathbb{E}_{\text{ssa}}[\![E]\!] \ genv\ lenv\ str$
$\quad \text{in } (null, lenv, str', \text{CTX})$

$\mathbb{U}[\![x = E; \overline{U}]\!] \text{ CTX } genv\ lenv\ str\ it =$
$\quad \text{let } (V, str') = \mathbb{E}_{\text{ssa}}[\![E]\!] \ genv\ lenv\ str$
$\quad \text{in} \mathbb{U}[\![\overline{U}]\!] \text{ CTX } genv\ lenv \cup (x, V)\ str'\ it, \text{CTX})$

$\mathbb{U}[\![\text{if } E \text{ goto } \text{CTX}'; \overline{U}]\!] \text{ CTX } genv\ lenv\ str\ it =$
$\quad \text{case } \mathbb{E}_{\text{ssa}}[\![E]\!] \ genv\ lenv\ str \text{ of}$
$\quad \{(true, str') \rightarrow \mathbb{U}[\![it[\text{CTX}']]\!] \text{ CTX } genv\ lenv\ str'\ it$
$\quad ; (false, str') \rightarrow \mathbb{U}[\![\overline{U}]\!] \text{ CTX } genv\ lenv\ str'\ it\}$

$\mathbb{U}[\![\text{goto } \text{CTX}'; \overline{U}]\!] \text{ CTX } genv\ lenv\ str\ it = \mathbb{U}[\![it[\text{CTX}']]\!] \text{ CTX } genv\ lenv\ str\ it$

$\mathbb{U}[\![\text{CTX}' : \overline{\phi}; \overline{U}]\!] \text{ CTX } genv\ lenv\ str\ it = \mathbb{U}[\![\overline{U}]\!] \text{ CTX}' \ genv\ (\mathbb{F}_{\text{ssa}}[\![\overline{\phi}]\!] \text{ CTX } lenv)\ str\ it$

$\text{buildIT} :: [\text{UnSSA}] \rightarrow \{(\text{CONTEXT}, [\text{UnSSA}])\}$

$\text{buildIT } [] = \{\}$

$\text{buildIT } (U; \overline{U}) = \text{buildIT } \overline{U}$

$\text{buildIT } (\text{CTX} : U; \overline{U}) = \text{let } it = \text{buildIT } (U; \overline{U}) \text{ in } it \cup \{(\text{CTX}, (U; \overline{U}))\}$

Figure 13: Dynamic Semantics of unstructured SSA

We prove by induction over $\overline{B}$. We only consider the two most interesting cases.

- Case CTX : $x = E; \overline{B}$ and CTX : return $E;$. Trivial.
- Case CTX : join $\{\overline{\phi'}\}$while $E \{\overline{B}\}; \overline{B'}$.
  Recall that

$$
\begin{aligned}
&\texttt{flatten}[\![CTX : \texttt{join } \{\overline{\phi'}\}\texttt{while } E \{\overline{B}\}; \overline{B'}]\!] \; \overline{\phi} \; G = \\
&\quad \texttt{let } \{(\overline{U}, \overline{\phi''}) = \texttt{flatten}[\![\overline{B}]\!] \; [] \; G \\
&\quad ; (\overline{U'}, \overline{\phi'''}) = \texttt{flatten}[\![\overline{B'}]\!] \; [] \; G\} \quad --(1) \\
&\quad \texttt{in } (CTX : \texttt{merge } \overline{\phi} \; (\texttt{merge } \overline{\phi''} \; \overline{\phi'}); \\
&\qquad \texttt{if } \neg(E) \texttt{ goto } (\texttt{nnsucc } G \; CTX); \\
&\qquad \overline{U}; \texttt{goto } CTX; \overline{U'}, \overline{\phi'''})
\end{aligned}
$$

and

$$
\begin{aligned}
&\overline{\mathbb{B}}_{\textsf{ssa}}[\![CTX : \texttt{join } \{\overline{\phi'}\}\texttt{while } E \{\overline{B}\}; \overline{B'}]\!] \; CTX_s \; \textit{genv lenv}_s \; \textit{str} \longrightarrow \\
&\texttt{case } \mathbb{B}_{\textsf{ssa}}[\![CTX : \texttt{join } \{\overline{\phi'}\}\texttt{while } E \{\overline{B}\}]\!] \; CTX_s \; \textit{genv lenv}_s \; \textit{str} \texttt{of} \\
&\quad (\_, \textit{lenv}_s', \textit{str}', CTX_s') \to \overline{\mathbb{B}}_{\textsf{ssa}}[\![\overline{B'}]\!] \; CTX_s' \; \textit{genv lenv}_s' \; \textit{str}' \quad --(2)
\end{aligned}
$$

where

$$
\begin{aligned}
&\mathbb{B}_{\textsf{ssa}}[\![CTX : \{\texttt{join } \{\overline{\phi}\} \texttt{ while } E \{\overline{B}\}\}]\!] \; CTX_s \; \textit{genv lenv}_s \; \textit{str} \\
&\longrightarrow \\
&\texttt{let } \textit{lenv}_s^1 = \mathbb{F}_{\textsf{ssa}}[\![\overline{\phi}]\!] \; CTX_s \; \textit{lenv}_s \\
&\texttt{in case } \mathbb{E}_{\textsf{ssa}}[\![E]\!] \; \textit{genv lenv}_s^1 \; \textit{str} \texttt{ of} \\
&(\textit{false}, \textit{str}_s^2) \to (\texttt{null}, \textit{lenv}_s^2, \textit{str}_s^2, CTX[\texttt{join}\{\blacksquare\}\texttt{while } \mathbf{E} \{\overline{\mathbf{B}}\}]) \\
&(\textit{true}, \textit{str}_s^2) \to \\
&\texttt{case } \overline{\mathbb{B}}_{\textsf{ssa}}[\![\overline{B}]\!] \; CTX[\texttt{join}\{\blacksquare\}\texttt{while } \mathbf{E} \{\overline{\mathbf{B}}\}] \; \textit{genv lenv}_s^1 \; \textit{str}_s^2 \texttt{of} \\
&\{(v_s^3, \textit{lenv}_s^2, \textit{str}_s^3, CTX_s^2) \to \\
&\quad \mathbb{B}_{\textsf{ssa}}[\![CTX : \{ \texttt{join } \{\overline{\phi}\} \texttt{ while } E \{\overline{B}\}\}]\!] \; CTX_s^2 \; \textit{genv lenv}_s^2 \; \textit{str}_s^3\}
\end{aligned}
$$

Recall

$$
\begin{aligned}
&\mathbb{U}[\![CTX : \texttt{merge } \overline{\phi} \; (\texttt{merge } \overline{\phi''} \; \overline{\phi'}); \\
&\quad \texttt{if } \neg(E)\texttt{goto } (\texttt{nnsucc } G \; CTX); \overline{U}; \\
&\quad \texttt{goto } CTX; \overline{U'}]\!] \; CTX_u \; \textit{genv lenv}_u \; \textit{str} \; \textit{it} \\
&\longrightarrow \\
&\mathbb{U}[\![\texttt{if } \neg(E)\texttt{goto } (\texttt{nnsucc } G \; CTX); \overline{U}; \\
&\quad \texttt{goto } CTX; \overline{U'}]\!] \; CTX \; \textit{genv lenv}_u^1 \; \textit{str} \; \textit{it} \quad --(3)
\end{aligned}
$$

where $\textit{lenv}_u^1 = \mathbb{F}_{\textsf{ssa}}[\![\texttt{merge } \overline{\phi} \; (\texttt{merge } \overline{\phi''} \; \overline{\phi'})]\!] \; CTX_u \; \textit{lenv}_u$
We consider the possible value of $\textit{lenv}_u^1$. Note that $\overline{\phi}$ is the result from flattening the preceding block and $\overline{\phi''}$ is from the while loop body. From the premise, we note that $CTX_u \overset{G}{\le} CTX_s$. There are two sub-cases:

– Subcase 1: CTX_u and CTX_s are from the preceding block.

$$
\begin{aligned}
\textit{lenv}_u^1 &= \mathbb{F}_{\textsf{ssa}}[\![\texttt{merge } \overline{\phi} \; (\texttt{merge } \overline{\phi''} \; \overline{\phi'})]\!] \; CTX_u \; \textit{lenv}_u \\
&\quad \text{drop } \overline{\phi''} \text{ since both contexts} \\
&\quad \text{are from the preceding block.} \\
&= \mathbb{F}_{\textsf{ssa}}[\![\texttt{merge } \overline{\phi} \; \overline{\phi'}]\!] \; CTX_u \; \textit{lenv}_u \\
&\quad \text{By Lemma C.1.} \\
&\quad \text{Note that } CTX_u \in \textit{params}(\overline{\phi}), CTX_s \in \textit{params}(\overline{\phi'}). \\
&= \mathbb{F}_{\textsf{ssa}}[\![\overline{\phi'}]\!] \; CTX_s \; (\mathbb{F}_{\textsf{ssa}}[\![\overline{\phi}]\!] \; CTX_u \; \textit{lenv}_u) \\
&\quad \text{From the premise.} \\
&= \mathbb{F}_{\textsf{ssa}}[\![\overline{\phi'}]\!] \; CTX_s \; \textit{lenv}_s \\
&= \textit{lenv}_s^1
\end{aligned}
$$

– Subcase 2: CTX_u and CTX_s are from the while loop body.

$$
\begin{aligned}
\textit{lenv}_u^1 &= \mathbb{F}_{\textsf{ssa}}[\![\texttt{merge } \overline{\phi} \; (\texttt{merge } \overline{\phi''} \; \overline{\phi'})]\!] \; CTX_u \; \textit{lenv}_u \\
&\quad \text{drop } \overline{\phi} \text{ since both contexts} \\
&\quad \text{are from the preceding block.} \\
&= \mathbb{F}_{\textsf{ssa}}[\![\texttt{merge } \overline{\phi''} \; \overline{\phi'}]\!] \; CTX_u \; \textit{lenv}_u \\
&\quad \text{By Lemma C.1.} \\
&\quad \text{Note that } CTX_u \in \textit{params}(\overline{\phi''}), CTX_s \in \textit{params}(\overline{\phi'}). \\
&= \mathbb{F}_{\textsf{ssa}}[\![\overline{\phi'}]\!] \; CTX_s \; (\mathbb{F}_{\textsf{ssa}}[\![\overline{\phi''}]\!] \; CTX_u \; \textit{lenv}_u) \\
&\quad \text{By I.H.} \\
&= \mathbb{F}_{\textsf{ssa}}[\![\overline{\phi'}]\!] \; CTX_s \; \textit{lenv}_s \\
&= \textit{lenv}_s^1
\end{aligned}
$$

Note that in the above we apply I.H. Co-induction is not required, since we get I.H. condition from the previous iteration of the loop. The first iteration is covered by Subcase 1.
We conclude that $\textit{lenv}_u^1 = \textit{lenv}_s^1$ (4). Continuing from (3),

$$
\begin{aligned}
&\mathbb{U}[\![\texttt{if } \neg(E) \texttt{ goto } (\texttt{nnsucc } G \; CTX); \overline{U}; \\
&\texttt{goto } CTX; \overline{U'}]\!] \; CTX \; \textit{genv lenv}_u^1 \; \textit{str} \; \textit{it} \\
&\longrightarrow \\
&\texttt{case } \mathbb{E}_{\textsf{ssa}}[\![E]\!] \; \textit{genv lenv}_u^1 \; \textit{str} \texttt{ of} \quad --(5) \\
&\quad (\textit{false}, \textit{str}_u^2) \to \mathbb{U}[\![\textit{it}[\texttt{nnsucc } G \; CTX]]\!] \; CTX \; \textit{genv lenv}_u^1 \; \textit{str}_u^2 \\
&\quad (\textit{true}, \textit{str}_u^2) \to \mathbb{U}[\![\overline{U}; \texttt{goto } CTX; \overline{U'}]\!] \; CTX \; \textit{genv lenv}_u^1 \; \textit{str}_u^2
\end{aligned}
$$

Given (4), we consider the following two sub-cases.

- Subcase 1: $\mathbb{E}_{\textsf{ssa}}[\![E]\!] \; \textit{genv lenv}_u^1 \; \textit{str}$ evaluates to $(\textit{false}, \textit{str}_u^2)$ and $\mathbb{E}_{\textsf{ssa}}[\![E]\!] \; \textit{genv lenv}_s^1 \; \textit{str}$ evaluates to $(\textit{false}, \textit{str}_s^2)$ and $\textit{str}_u^2 = \textit{str}_s^2$. Continuing from (2), we have

$$
\overline{\mathbb{B}}_{\textsf{ssa}}[\![\overline{B'}]\!] \; CTX[\texttt{join } \{\blacksquare\} \texttt{ while } \mathbf{E} \{\overline{\mathbf{B}}\}] \; \textit{genv lenv}_s' \; \textit{str}_s' \quad --(6)
$$

Continuing from (5), we have

$$
\mathbb{U}[\![\overline{U'}]\!] \; CTX \; \textit{genv lenv}_u^1 \; \textit{str}_u^2 \; \textit{it} \quad --(7)
$$

where $\textit{lenv}_s' = \textit{lenv}_s^1 = \textit{lenv}_u^1$ and $\textit{str}_s' = \textit{str}_s^2 = \textit{str}_u^2$. Note that CTX is the context containing the while loop. $CTX \overset{G}{\le} CTX[\texttt{join } \{\blacksquare\} \texttt{ while } \mathbf{E} \{\overline{\mathbf{B}}\}]$. From (1), we find that $\textit{lenv}_s' = \mathbb{F}_{\textsf{ssa}}[\![[]]\!] \; CTX \; \textit{lenv}_u^1$. We apply I.H to (6) and (7) to conclude this sub case.

- Subcase 2: $\mathbb{E}_{\textsf{ssa}}[\![E]\!] \; \textit{genv lenv}_u^1 \; \textit{str}$ evaluates to $(\textit{true}, \textit{str}_u^2)$ and $\mathbb{E}_{\textsf{ssa}}[\![E]\!] \; \textit{genv lenv}_s^1 \; \textit{str}$ evaluates to $(\textit{true}, \textit{str}_s^2)$ and $\textit{str}_u^2 = \textit{str}_s^2$. Continuing from (2), we have

$$
\begin{aligned}
&\texttt{case } \overline{\mathbb{B}}_{\textsf{ssa}}[\![\overline{B}]\!] \; CTX[\texttt{join } \{\blacksquare\} \texttt{ while } \mathbf{E} \{\overline{\mathbf{B}}\}] \; \textit{genv lenv}_s^1 \; \textit{str}_s^2 \texttt{ of} \\
&(v_s^3, \textit{lenv}_s^2, \textit{str}_s^3, CTX_s^3) \to \\
&\overline{\mathbb{B}}_{\textsf{ssa}}[\![CTX : \{\texttt{join}\{\overline{\phi'}\}\}\texttt{while } E \{\overline{B}\}\}; \overline{B'}]\!] \\
&\quad CTX_s^3 \; \textit{genv lenv}_s^2 \; \textit{str}_s^3 \qquad --(8)
\end{aligned}
$$

Note that in the above we "unroll" the evaluation of following block $\overline{B'}$ back into the right hand side of the case pattern clause, by applying the inverse of $\overline{\mathbb{B}}_{\textsf{ssa}}[\![\cdot]\!]$. Continuing from (5), we have

$\mathbb{U}[\![\overline{U}; \text{goto CTX}; \overline{U'}]\!]$ CTX $genv\ lenv_u^1\ str_u^2\ it$
$\longrightarrow$
let $(v_u^3, lenv_u^2, str_u^3, \text{CTX}_u^3) = \mathbb{U}[\![\overline{U}]\!]$ CTX $genv\ lenv_u^1\ str_u^2\ it$
in$\mathbb{U}[\![\text{goto CTX}; \overline{U}]\!]$ CTX$_u^3\ genv\ lenv_u^2\ str_u^3\ it$
let $(v_u^3, lenv_u^3, str_u^3, \text{CTX}_u^3) = \mathbb{U}[\![\overline{U}]\!]$ CTX $genv\ lenv_u^1\ str_u^2\ it$
in$\mathbb{U}[\![\text{CTX} : \text{merge}\ \overline{\phi}\ (\text{merge}\ \overline{\phi''}\ \overline{\phi'});$
   if $\neg(E)$goto (nnsucc G CTX); $\overline{U}$; goto CTX; $\overline{U'}]\!]$
      CTX$_u^3\ genv\ lenv_u^2\ str_u^3\ it$ $--(9)$

By Definition 12 and construction of the CFG, we have
CTX $\overset{G}{\preceq}$ CTX[join $\{\blacksquare\}$ while $E\ \{\overline{B}\}]$. From (4) and $str_u^2 = str_s^2$
we apply I.H. to verify that

$$lenv_s^2 = \mathbb{F}_{\text{ssa}}[\![\overline{phi''}]\!]\ \text{CTX}_u^3$$

and CTX$_u^3\ \overset{G}{\preceq}$ CTX$_s^3$ and $str_u^3 = str_s^3$.
We apply I.H. to (8) and (9) then we verify this sub case.

Case: CTX : {if $E\ \{\overline{B_1}\}$ else$\{\overline{B_2}\}$join $\{\overline{\phi'}\}\}; \overline{B_3}$. Recall that

flatten$[\![\text{CTX} : \{\text{if}\ E\ \{\overline{B_1}\}\text{else}\{\overline{B_2}\}\ \text{join}\{\overline{\phi'}\}\}; \overline{B_3}]\!]\ \overline{\phi}\ G =$
   let $\{(\overline{U_1}, \overline{\phi_1}) = \text{flatten}[\![\overline{B_1}]\!]\ []\ G;$
      $(\overline{U_2}, \overline{\phi_2}) = \text{flatten}[\![\overline{B_2}]\!]\ []\ G;$ $--(10)$
      CTX$_2 = \text{CTX}[\text{if}\ E\ \{\overline{B}\}\text{else}\{\square\}\ \text{join}\{\overline{\phi}\}];$
      $\{\text{CTX}_3\} = (\text{succs}\ G\ \mathbb{X}[\![\overline{B_1}]\!]) \cap (\text{succs}\ G\ \mathbb{X}[\![\overline{B_2}]\!]);$
      $\overline{\phi_3} = \text{merge}\ \overline{\phi_1}\ (\text{merge}\ \overline{\phi_2}\ \overline{\phi'});$
      $(\overline{U_4}, \overline{\phi_4}) = \text{flatten}[\![\overline{B_3}]\!]\ \overline{\phi_3}\ G\}$
   in (CTX : $\overline{\phi}$; if $\neg(E)$ goto CTX$_2$; $\overline{U_1}$; goto CTX$_3$; $\overline{U_2}$; $\overline{U_4}, \overline{\phi_4}$)

and

$\overline{\mathbb{B}}_{\text{ssa}}[\![\text{CTX} : \{\text{if}\ E\ \{\overline{B_1}\}\ \text{else}\ \{\overline{B_2}\}\ \text{join}\{\overline{\phi}\}\}; \overline{B_3}]\!]$
   CTX$_s\ genv\ lenv_s\ str \rightarrow$
case $\mathbb{B}_{\text{ssa}}[\![\text{CTX} : \{\text{if}\ E\ \{\overline{B_1}\}\text{else}\{\overline{B_2}\}\text{join}\{\overline{\phi}\}\}]\!]$
   CTX$_s\ genv\ lenv_s\ str'$of
   $(\_, lenv_s', str_s', \text{CTX}_s') \rightarrow \overline{\mathbb{B}}_{\text{ssa}}[\![\overline{B_3}]\!]$ CTX$_s'\ genv\ lenv_s'\ str'$ $--(11)$

where

$\mathbb{B}_{\text{ssa}}[\![\text{CTX} : \{\text{if}\ E\ \{\overline{B_1}\}\ \text{else}\ \{\overline{B_2}\}\ \text{join}\{\overline{\phi}\}\}]\!]$ CTX $genv\ lenv_s\ str \longrightarrow$
case $\mathbb{E}_{\text{ssa}}[\![E]\!]\ genv\ lenv_s\ str_s^1$ of
   $(true, str_s^1) \rightarrow$ case $\overline{\mathbb{B}}_{\text{ssa}}[\![\overline{B_1}]\!]$ CTX $genv\ lenv_s\ str_s^1$ of
      $(v_s^2, lenv_s^2, str_s^2, \text{CTX}_s^2) \rightarrow$
      $(v_s^2, \mathbb{F}_{\text{ssa}}[\![\overline{\phi'}]\!]$ CTX$_s^2\ lenvs_s^2, str_s^2,$
         CTX[if $E\ \{\overline{B}\}$ else$\{\overline{B}\}$join$\{\blacksquare\}\}])$ $--(12)$
   $(false, str_s^1) \rightarrow$ case $\overline{\mathbb{B}}_{\text{ssa}}[\![\overline{B_2}]\!]$ CTX $genv\ lenv_s\ str_s^1$ of
      $(v_s^2, lenv_s^2, str_s^2, \text{CTX}_s^2) \rightarrow$
      $(v_s^2, \mathbb{F}_{\text{ssa}}[\![\overline{\phi'}]\!]$ CTX$_s^2\ lenv_s^2, str_s^2,$
         CTX[if $E\ \{\overline{B}\}$ else$\{\overline{B}\}$join$\{\blacksquare\}\}])$ $--(13)$

Recall that
$\mathbb{U}[\![\text{CTX} : \overline{\phi}; \text{if}\ \neg(E)\ \text{goto CTX}_2; \overline{U_1}; \text{goto CTX}_3; \overline{U_2}; \overline{U_4}]\!]$
CTX$_u\ genv\ lenv_u\ str\ it$
$\longrightarrow$
$\mathbb{U}[\![\text{if}\ \neg(E)\ \text{goto CTX}_2; \overline{U_1}; \text{goto CTX}_3; \overline{U_2}; \overline{U_4}]\!]$
CTX $genv\ (\mathbb{F}_{\text{ssa}}[\![\overline{\phi}]\!]\ \text{CTX}_u\ lenv_u)\ str\ it$
$\longrightarrow$ (from the premise$lenv_s = \mathbb{F}_{\text{ssa}}[\![\overline{\phi}]\!]\ \text{CTX}_u\ lenv_u$)
$\mathbb{U}[\![\text{if}\ \neg(E)\ \text{goto CTX}_2; \overline{U_1}; \text{goto CTX}_3; \overline{U_2}; \overline{U_4}]\!]$ CTX $genv\ lenv_s\ str\ it$
$\longrightarrow$
case $\mathbb{E}_{\text{ssa}}[\![E]\!]\ genv\ lenv_s\ str$ of
   $\{(false, str_u^1) \rightarrow \mathbb{U}[\![it[\text{CTX}_2]]\!]$ CTX $genv\ lenv_s\ str_u^1$
   ; $(true, str_u^1) \rightarrow \mathbb{U}[\![\overline{U_1}; \text{goto CTX}_3; \overline{U_2}; \overline{U_4}]\!]$ CTX $genv\ lenv_s\ str_u^1\}$
$\longrightarrow$
case $\mathbb{E}_{\text{ssa}}[\![E]\!]\ genv\ lenv_s\ str$ of
   $\{(false, str_u^1) \rightarrow \mathbb{U}[\![\overline{U_2}; \overline{U_4}]\!]$ CTX $genv\ lenv_s\ str_u^1$ $--(14)$
   ; $(true, str_u^1) \rightarrow \mathbb{U}[\![\overline{U_1}; \text{goto CTX}_3; \overline{U_2}; \overline{U_4}]\!]$ CTX $genv\ lenv_s\ str_u^1\}$
      $--(15)$

We consider two sub-cases:
- Subcase 1: $\mathbb{E}_{\text{ssa}}[\![E]\!]\ genv\ lenv_s\ str$ evaluates to $(false, str_u^1)$.
  Note that $str_u^1 = str_s^1$. (11) + (13) evaluates to

  case $\mathbb{B}_{\text{ssa}}[\![\overline{B_2}]\!]$ CTX $genv\ lenv_s\ str_s^1$ of
     $(v_s^2, lenv_s^2, str_3^2, \text{CTX}_s^2) \rightarrow$
        $\overline{\mathbb{B}}_{\text{ssa}}[\![\overline{B_3}]\!]$ CTX[if $E\ \{\overline{B}\}$else$\{\overline{B}\}$join $\{\blacksquare\}]$
     $genv\ (\mathbb{F}_{\text{ssa}}[\![\overline{\phi'}]\!]$ CTX$_s^2\ lenv_s^2)\ str_s^2$ $--(16)$

  Let $lenv_s' = \mathbb{F}_{\text{ssa}}[\![\overline{\phi'}]\!]$ CTX$_s^2\ lenv_s^2$. Next we consider the un-
  structured SSA. (14) evaluates to

  $\mathbb{U}[\![\overline{U_2}; \overline{U_4}]\!]$ CTX $genv\ lenv_s\ str_u^1 \longrightarrow$
  case $\mathbb{U}[\![\overline{U_2}]\!]$ CTX $genv\ lenv_s\ str_u^1$of
     $(v_u^2, lenv_u^2, str_u^2, \text{CTX}_u^2) \rightarrow \mathbb{U}[\![\overline{U_4}]\!]$ CTX$_u^2\ genv\ lenv_u^2\ str_u^2$ $--(17)$

  By I.H. We have $lenv_s^2 = \mathbb{F}_{\text{ssa}}[\![\overline{\phi_2}]\!]$ CTX$_u^2\ lenv_u^2$ (18), and
  CTX$_u^2\ \overset{G}{\preceq}$ CTX$_s^2$ and $str_u^2 = str_s^2$. Note that CTX$_s^2$ must be in
  the set $\mathbb{X}[\![\overline{B_2}]\!]$, thus CTX$_s^2\ \overset{G}{\preceq}$ CTX[if $E\ \{\overline{B}\}$else$\{\overline{B}\}$join$\{\blacksquare\}]$. It
  follows that

  $$\text{CTX}_u^2\ \overset{G}{\preceq}\ \text{CTX}[\text{if}\ E\ \{\overline{B}\}\text{else}\{\overline{B}\}\text{join}\{\blacksquare\}]$$

  In addition, Since we are considering the else branch. $\overline{\phi_3} =$
  merge $\overline{\phi_1}$ (merge $\overline{\phi_2}\ \overline{\phi'}$) = merge $\overline{\phi_2}\ \overline{\phi'}$. By Lemma C.1 it
  follows from (18) that

  $$lenv_s' = \mathbb{F}_{\text{ssa}}[\![\overline{\phi_3}]\!]\ \text{CTX}_u^2\ lenv_u^2$$

  We apply I.H. to (16) and (17) to verify this subcase.
- Subcase 2: $\mathbb{E}_{\text{ssa}}[\![E]\!]\ genv\ lenv_s\ str$ evaluates to $(true, str_u^1)$.
  It can be proven similar to the previous sub case.

  $\square$

C.3 Time Complexity of Flattening Algorithm

Note that the flattening algorithm has the worst-case time complex-
ity $O(n^3)$ where n is the size of the program, as each call to merge
is $O(n^2)$.