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@alumni.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

• **Software and its engineering** → **Compilers**; *Software verification and validation*; • **Theory of computation** → *Program semantics.*

KEYWORDS

Static single assignment, Program transformation, SSA construction

ACM Reference Format:
Kenny Zhuo Ming Lu and Daniel Yu Hian Low. 2023. Constructing Structured SSA from FJ. In *Proceedings of the 25th ACM International Workshop on Formal Techniques for Java-like Programs (FTfJP '23), July 18, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3605156.3606457

```
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)

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. References to variables that are dependent on the control flow (e.g. an if-else statement) require 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 [8, 11, 12]. We note that from literatures, most of the SSA construction algorithms to date [6, 7, 10, 15, 17], 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, Figure 1 (b). Unstructured SSA form was originally designed for compiler optimization, it is not optimized for program verification. For example, given a structured program in FJ, one has to transform the source code 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, [8, 11, 12]. 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

FTfJP '23, July 18, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0246-4/23/07…$15.00
https://doi.org/10.1145/3605156.3606457

```
public int max() {
  int res = this.arr[0]; int i = 1;
  while (i < this.arr.length) {
    if (res < this.arr[i]) {
      res = this.arr[i];
    }
    i = i + 1;
  }
  return res;
}
```

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

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 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.

$$
\begin{array}{l}
(\ s_0 = 0;\ \ i_0 = 0; \\
\quad (i_1^1 = i_0; s_1^1 = s_0; assume\ \ (i_1^1 < N) \wedge s_1^1 <= i_1^1 * (i_1^1 - 1)/2; \\
\quad s_2^1 = s_1^1 + i_1^1; i_2^1 = i_1^1 + 1); assert\ \ false;\ )\ \ \square \\
(i_1 = i_0; s_1 = s_0; assume\ \ (i_1 >= N) \wedge s_1 <= i_1 * (i_1 - 1)/2;)
\end{array}
$$

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

In [1], Ancona 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 formalize the algorithm in declarative style, 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 (SSAFJ). In Section 4, we present the SSA construction algorithm and the formal results. In Section **??**, we discuss the implementation details. In Section 6 we discuss the related works and conclude in Section 7.

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

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

Where $\square$ denotes a hole in a program structure. $\mathbf{s}$ symbolically denotes a statement, and $\overline{\mathbf{s}}$ symbolically denotes a sequence of statements. The program contexts for the SSA form work in a similar way, except that instead of $\mathbf{s}$, we have $\mathbf{B}$ 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 locations where variable redefinitions occur, 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,

$$
\begin{array}{lllll}
I: & (\texttt{res}, & \square; \overline{\mathbf{s}}, & \square; \overline{\mathbf{B}}, & \texttt{res0}) \\
II: & (\texttt{i}, & \mathbf{s}; \square; \overline{\mathbf{s}}, & \square; \overline{\mathbf{B}}, & \texttt{i0})
\end{array}
$$

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 $\texttt{CTX}_d \leq \texttt{CTX}_r$. Intuitively speaking, the relation states that variables defined in $\texttt{CTX}_d$ might be redefined in $\texttt{CTX}_r$. Given that current SSAFJ context is $\texttt{CTX}_1 = \mathbf{B};\square;\overline{\mathbf{B}}$, we search in the variable mapping table to look for all the contexts CTX such that $\texttt{CTX} \leq \texttt{CTX}_1$. We found only one candidate $\texttt{CTX} = \square;\overline{\mathbf{B}}$. 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 join semi-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, ·)` and `res1 = phi(0:res0, ·)`. We extend the variable mapping table with two new entries as follows,

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

where $\texttt{CTX}_2 = \texttt{CTX}_1[\text{join } \{\blacksquare\}\text{while } \mathbf{E}\{\overline{\mathbf{B}}\}]$. 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 CTX[CTX′]. $\blacksquare$ is similar to $\square$ except that it cannot be substituted. To resolve the second operands of the phi assignments, we first convert the body of the while loop. We face an issue. The program context referring to the body of the while loop $\texttt{CTX}_1[\text{join } \{\overline{\phi}\}\text{while } \mathbf{E}\{\square\}]$ satisfies the following cyclical relation.

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

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 $\texttt{CTX}_2^{0,1} = \texttt{CTX}_1[\text{join } \{\blacksquare^{0,1}\}\text{while } \mathbf{E}\{\overline{\mathbf{B}}\}]$. The < relation is then adjusted to respect the bit b as follows

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

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 $\texttt{CTX}_2$, 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

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

  where
  - $\texttt{ctx}_3 = \mathbf{s};\mathbf{s};\text{while } \mathbf{e}\{\text{if } \mathbf{e}\ \{\square;\}\text{else}\{\overline{\mathbf{s}}\};\overline{\mathbf{s}}\};\overline{\mathbf{s}}$
  - $\texttt{CTX}_3 = \texttt{CTX}_1[\text{join } \{\overline{\phi}\}\text{while } \mathbf{E}\{\text{if } \mathbf{E}\ \{\square;\}\text{else}\{\overline{\mathbf{B}}\}\text{join}\{\overline{\phi}\};\overline{\mathbf{B}}\}]$
- 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 $\texttt{CTX}_3$, and the most recent definition from the else branch is $\texttt{CTX}_2$. 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

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

where

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

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 $\texttt{CTX}_2$.

`5: i2 = i1+ 1`

We add the following entry to the variable mapping table

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

where

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

```
public int max() {
0:int res0, res1, res2, res3;
   int i0, i1, i2;
   res0 = this.arr[0];
   i0 = 1;
1:join {res1 = phi(0:res0, 5:res2),
        i1 = phi(0:i0, 5:i2)}
  while (i1 < this.arr.length) {
  2:if (res1 < this.arr[i1]) {
    3: res3 = this.arr[i1];
    } else { 4: nop;
    } join {res2 = phi(3:res3, 4:res1)}
  5:i2 = i1 + 1;
  }
6:return res1;
}
```

Figure 3: Method `max` in SSA form

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

By looking up the variable mapping table with $\texttt{CTX}_2^0$ replaced by $\texttt{CTX}_2^1$, we find that the most recent redefinition of `i` is `i2` at $\texttt{CTX}_5$ and the most most recent redefinition of `res` is `res2` at $\texttt{CTX}_4$. We collect the renamed variables and declare them at the first block in 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 some minor difference in handling phi assignments. 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 $x = \texttt{phi}(\texttt{CTX}_1 : x_1, ..., \texttt{CTX}_n : x_n)$, the semantic rules (PhiRes) and (EmpPhiRes) select the variable x_i based on the matching of label $\texttt{CTX}_i$ and update the local environment with $(x, lenv(x_i))$. Given a set of pairs S, we write $S + (x, v)$ to denote $\{(y, v') \mid (y, v') \in S \wedge y \neq x\} \cup \{(x, v)\}$. $[]$ 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 $\texttt{CTX}[S]$ to refer to $\{\texttt{CTX}[\texttt{CTX}'] \mid \texttt{CTX}' \in S\}$. The auxiliary function $\mathbb{R}_{\leq \texttt{CTX}}(\beta, x)$ 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. $\beta, \texttt{CTX} \vdash e \Rightarrow E$ converts an expression, $\vdash md \Rightarrow MD$ converts a method declaration. $\beta \vdash \overline{vd} \Rightarrow \overline{VD}$ converts a sequence of variable declarations. $\texttt{ctx}, \beta, \texttt{CTX} \vdash \bar{s} \Rightarrow \overline{B}, \beta, \texttt{CTX}$ and $\texttt{ctx}, \beta, \texttt{CTX} \vdash s \Rightarrow B, \beta, \texttt{CTX}$ 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 $\mathbb{R}_{\leq \texttt{CTX}}(\beta, x)$ 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 $\texttt{if}\ e\ \{\overline{s_1}\}\texttt{else}\{\overline{s_2}\}$. 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

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

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

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

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

assignment by associating label $\texttt{CTX}_1$ with the most recent definition of x at $\texttt{CTX}_1$ and associating label $\texttt{CTX}_2$ with the most recent definition of x at $\texttt{CTX}_2$, where $\texttt{CTX}_1$ and $\texttt{CTX}_2$ the exiting contexts from the branches.

Case while $e\ \{\overline{s}\}$. 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 $\texttt{CTX}_0$. For each variable x in the domain of the incoming β_0, we construct a phi assignment of which the first operand has the label $\texttt{CTX}_0$ and the expression $\mathbb{R}_{\leq \texttt{CTX}_0}(\beta_0, x)$. 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 $\texttt{CTX}_2$, for each updated variable x in $dom3(\beta_2 - \beta_1)$, we *update* the second operand to the existing phi function, with $\texttt{CTX}_2$ as the label and $\mathbb{R}_{\leq \texttt{CTX}_2}(\beta_2, x)$ as the expression. Now we need to eliminate redundant phi assignments, which were 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 $\texttt{CTX}[\text{join } \{\blacksquare^0\}\text{while}\mathbf{E}\{\overline{\mathbf{B}}\}]$ in the resulting variable mapping table to 1.

4.1 Consistency

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.

Theorem 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.*

To verify this theorem, we first need to show that the variable mapping tables produced by the conversion algorithm are join semi-lattices.

Definition 1 (Program Context Join Semi-lattice). *Let* β *be a variable mapping and* CTX *be an SSAFJ program context. We say* $\beta \vdash_L \texttt{CTX}$ *iff* $\{\texttt{CTX}' | (x, \texttt{ctx}, \texttt{CTX}', x') \in \beta, \texttt{CTX}' \leq \texttt{CTX}\}$ *is a join semi-lattice.*

Lemma 4.2 (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* $\texttt{ctx}, \beta, \texttt{CTX} \vdash s \Rightarrow B, \beta', \texttt{CTX}'$. *Then* $\beta' \vdash_L \texttt{CTX}'$.

The second property required to verify the main consistency theorem is that the SSA conversion algorithm preserves the local declaration environment consistency.

Definition 2 (Consistent Local Environments). *Let lenv be an FJ local declaration environment and lenv′ be an SSAFJ local environment. Let* β *be a variable mapping,* CTX *be an SSAFJ program context. Then we say* $lenv \vdash_{(\texttt{CTX},\beta)} lenv'$ *iff for all* $x \in dom(lenv)$ *we have* $lenv(x) = lenv'(\mathbb{R}_{\leq \texttt{CTX}}(\beta, x))$.

Lemma 4.3 (Statement Conversion Consistency (simplified)). *Let* $\texttt{ctx}, \beta, \texttt{CTX} \vdash s \Rightarrow B, \beta', \texttt{CTX}'$. *Let lenv be an FJ local declaration environment and lenv′ be an SSAFJ local environment such that* $lenv \vdash_{(\texttt{CTX},\beta)} lenv'$. *Under the same class environment and memory environment, let lenv″, v, str be the resulted FJ local declaration environment, returned value and memory environment of evaluating s. let lenv‴, v′, str′ be the resulted SSAFJ local declaration environment, returned value and memory environment of evaluating B. Then* $lenv'' \vdash_{(\beta', \texttt{CTX}')} lenv'''$ *and* $v = v'$ *and* $str = str'$.

Note that we have not formalized the class environment and memory environment in the main text, they can be found in the Appendix.

4.2 Minimality

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

$$
\begin{array}{rcl}
[\![\cdot]\!] & :: & \texttt{Context} \rightarrow \texttt{CONTEXT} \\
[\![\Box]\!] & = & \Box \\
[\![\texttt{ctx};]\!] & = & [\![ctx]\!]; \\
[\![\texttt{ctx};\overline{\texttt{s}}]\!] & = & [\![\texttt{ctx}]\!];\overline{\texttt{B}} \\
[\![\texttt{s};\texttt{ctx}]\!] & = & \texttt{B};[\![\texttt{ctx}]\!] \\
[\![\texttt{if e }\{\texttt{ctx}\}\texttt{ else }\{\overline{\texttt{s}}\}]\!] & = & \texttt{if E }\{[\![\texttt{ctx}]\!]\}\texttt{ else }\{\overline{\texttt{B}}\}\texttt{ join }\{\overline{\phi}\} \\
[\![\texttt{if e }\{\overline{\texttt{s}}\}\texttt{ else }\{\texttt{ctx}\}]\!] & = & \texttt{if E }\{\overline{\texttt{B}}\}\texttt{ else }\{[\![\texttt{ctx}]\!]\}\texttt{ join }\{\overline{\phi}\} \\
[\![\texttt{while e }\{\texttt{ctx}\}]\!] & = & \texttt{join }\{\overline{\phi}\}\texttt{ while E }\{[\![\texttt{ctx}]\!]\}
\end{array}
$$

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

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

$Last(\Box;)$ $\quad Last(\texttt{if E }\{\overline{\texttt{B}}\}\texttt{else}\{\overline{\texttt{B}}\}\texttt{join}\{\blacksquare\};)$

$$\frac{Last(\texttt{CTX})}{Last(\texttt{B};\texttt{CTX})} \quad Last(\texttt{join}\{\blacksquare\}\texttt{while E }\{\overline{\texttt{B}}\};)$$

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

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

$$\texttt{join }\{\blacksquare^0\}\texttt{ while E }\{\overline{\texttt{B}}\} < \texttt{join }\{\overline{\phi}\}\texttt{ while E }\{\Box\}\ \text{(CtxWhile1)} \quad \frac{Last(\texttt{CTX})}{\texttt{join }\{\overline{\phi}\}\texttt{ while E }\{\texttt{CTX}\} < \texttt{join }\{\blacksquare^1\}\texttt{ while E }\{\overline{\texttt{B}}\}}\text{(CtxWhile2)}$$

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

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

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

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

$$\beta, \texttt{CTX} \vdash v \Rightarrow v\ \text{(ValEConv)}$$
$$\beta, \texttt{CTX} \vdash x \Rightarrow \mathbb{R}_{\leq \texttt{CTX}}(\beta, x)\text{(VarEConv)}$$
$$\beta, \texttt{CTX} \vdash this \Rightarrow this\ \text{(ThisEConv)}$$

$$\frac{\beta, \texttt{CTX} \vdash e_1 \Rightarrow E_1 \quad \beta, \texttt{CTX} \vdash e_2 \Rightarrow E_2}{\beta, \texttt{CTX} \vdash e_1\ op\ e_2 \Rightarrow E_1\ op\ E_2}\text{(OpEConv)} \quad \frac{\beta, \texttt{CTX} \vdash e \Rightarrow E}{\beta, \texttt{CTX} \vdash e.f \Rightarrow E.f}\text{(FieldEConv)}$$

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

$$\frac{\Box, [(x, \Box, \Box, x)], \Box \vdash \overline{s} \Rightarrow \overline{B}, \beta, \texttt{CTX} \quad \beta \vdash \overline{vd} \Rightarrow \overline{VD}}{\vdash t\ m\ (t'\ x)\ \{\overline{vd}; \overline{s}\} \Rightarrow t\ m\ (t'\ x)\{\overline{VD}; \overline{B}\}}$$

$\boxed{\beta \vdash \overline{vd} \Rightarrow \overline{VD}}$

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

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

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

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

$$\frac{\texttt{CTX}' = [\![\texttt{ctx}]\!] \quad \beta, \texttt{CTX} \vdash e \Rightarrow E \quad \beta' = \beta \cup \{(x, \texttt{ctx}, \texttt{CTX}', x_{\texttt{CTX}'})\}}{\texttt{ctx}, \beta, \texttt{CTX} \vdash x = e \Rightarrow \texttt{CTX}' : x_{\texttt{CTX}'} = E, \beta', \texttt{CTX}'}\text{(AsmStmtConv)} \quad \frac{\texttt{CTX}' = [\![\texttt{ctx}]\!] \quad \beta, \texttt{CTX} \vdash e \Rightarrow E \quad \beta, \texttt{CTX} \vdash e' \Rightarrow E'}{\texttt{ctx}, \beta, \texttt{CTX} \vdash e.f = e' \Rightarrow \texttt{CTX}' : E.f = E', \beta, \texttt{CTX}'}\text{(FdStmtConv)}$$

$$\frac{\texttt{CTX}' = [\![\texttt{ctx}]\!] \quad \beta, \texttt{CTX} \vdash e \Rightarrow E}{\texttt{ctx}, \beta, \texttt{CTX} \vdash \texttt{return } e \Rightarrow \texttt{CTX}' : \texttt{return } E, \beta, \texttt{undef}}\text{(RetStmtConv)} \quad \frac{\texttt{CTX} = [\![\texttt{ctx}]\!] \quad \beta, \texttt{CTX}' \vdash e \Rightarrow E \quad \beta, \texttt{CTX}' \vdash e' \Rightarrow E' \quad \beta' = \beta \cup \{(x, \texttt{ctx}, \texttt{CTX}, x_{\texttt{CTX}})\}}{\texttt{ctx}, \beta, \texttt{CTX}' \vdash e.m(e') \Rightarrow \texttt{CTX} : x_{\texttt{CTX}} = E.m(E'), \beta', \texttt{CTX}}\text{(MethdInvStmtConv)}$$

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

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

Figure 5: FJ to SSAFJ conversion

Theorem 4.4 (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.*

To validate this property, the main idea is to characterize the set of SSAFJ context pairs referenced in the RHS of phi assignments in the resulted SSAFJ program. Recall from [10], the *dominance frontier* of a program location `ctx` is the "earliest" program location `ctx′` where `ctx` loses its dominance. We introduce the notion of *dominance frontier handle* as the pair of immediate predecessors of a dominance frontier `ctx′`.

For instance, consider the phi assignment in line 12 in Figure 3, with numerical labels being replaced with the program contexts.

`res2 = phi(CTX3:res3, CTX'3:res2)`

where $\mathtt{CTX}'_3 = \mathtt{CTX}_1[\mathtt{join}\ \{\overline{\phi}\}\mathtt{while}\ \mathbf{E}\{\mathtt{if}\ \mathbf{E}\ \{\overline{\mathbf{B}}\}\mathtt{else}\{\square;\}\mathtt{join}\{\overline{\phi}\};\overline{\mathbf{B}}\}]$. The pair $\mathtt{ctx}_3$ and $\mathtt{ctx}'_3$ where $[\![\mathtt{ctx}_3]\!] = \mathtt{CTX}_3$ and $[\![\mathtt{ctx}'_3]\!] = \mathtt{CTX}'_3$ is a dominance frontier handle in the source program since their common successor $\mathtt{ctx}_5$ is a dominance frontier.

The intuition behind the validity of the minimality theorem is that all the program contexts referenced in the phi assignments are the (translated) dominance frontier handles. Hence the resulted SSAFJ is minimal.

4.3 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 IMPLEMENTATION

We implemented the structured SSA conversion algorithm in a library using Scala [14]. Our initial empirical results show that the number of phi assignments generated is equivalent the one produced by Cytron's approach. The current implementation has several restrictions. The first one restricts that all variables in a method must be declared and initialized at the beginning of the method body. To lift this restriction, we identify two possible approaches.

(1) to add an pre-processing step to lift variables to the outer scopes with renaming to avoid conflicts, or
(2) to add nested scopes to local declaration environment (for semantics) and variable mapping tables (for SSA conversion), hence for each scope we can prove that that Lemma 4.2 still holds.

The second restriction disallows a method to have multiple return statements. To lift this restriction, we envision that a pre-processing step is required, which merges different return statements by assigning the returned values into a new variable and return this variable.

6 RELATED WORKS

In [1], Ancona and Corradi formalized the syntax, semantics and validity of the structured SSA form for FJ, namely SSAFJ. They showed that converting FJ into SSAFJ enables a more precise type analysis to be conducted without unnecessary complexity, i.e. an untypable FJ program becomes typable in the same type analysis when it is converted to SSAFJ. Cytron et al's SSA algorithm [10] 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 [6], Braun et al proposed an SSA construction algorithm by traversing the AST backward and recursively. In a later work [9]. 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 [5]. The SSA produced by their algorithm is in a form of structured control flow graph. In his thesis [16], 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 [4], 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. In [3], Barnett and Leino presented a workflow to compute weakest-preconditions from unstructured programs. Their approach targets a different type of source programs. They removed redundant variable renamings by keeping track of a set of variable incarnation for each block. It is unknown whether their approach achieves minimality. We plan to conduct a followup study to compare both approaches and explore possibilities of reusing some of their techniques in our context. In [13], Lu applied a structured SSA to CPS translation to formalize a control flow obfuscation.

7 CONCLUSION

We proposed a novel algorithm which constructs structured SSA forms from FJ programs. 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. Detail proofs of the formal results and the GCL conversion algorithm can be found in the Appendix.

ACKNOWLEDGMENTS

We are grateful for the reviewers of FTfJP'23 giving their constructive feedback. We thank other anonymous reviewers for their comments and suggestion to the earlier draft of this paper.

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. https://doi.org/10.1007/3-540-46423-9_8

[3] Mike Barnett and K. Rustan M. Leino. 2005. Weakest-Precondition of Unstructured Programs. *SIGSOFT Softw. Eng. Notes* 31, 1 (sep 2005), 82–87. https://doi.org/10.1145/1108768.1108813

[4] 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. https://doi.org/10.1007/978-3-662-49498-1_3

[5] 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

[6] 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. https://doi.org/10.1007/978-3-642-37051-9_6

[7] 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. https://doi.org/10.1002/(SICI)1097-024X(19980710)28:8<859::AID-SPE188>3.0.CO;2-8

[8] David Brumley and Ivan Jager. 2011. *Efficient Directionless Weakest Preconditions.* Technical Report.

[9] 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* (Barcelona, Spain) *(CC 2016)*. Association for Computing Machinery, New York, NY, USA, 67–76. https://doi.org/10.1145/2892208.2892211

[10] 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* (Austin, Texas, USA) *(POPL '89)*. ACM, New York, NY, USA, 25–35. https://doi.org/10.1145/75277.75280

[11] 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* (London, United Kingdom) *(POPL '01)*. Association for Computing Machinery, New York, NY, USA, 193–205. https://doi.org/10.1145/360204.360220

[12] 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

[13] Kenny Zhuo Ming Lu. 2019. Control flow obfuscation via CPS transformation. In *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM@POPL 2019, Cascais, Portugal, January 14-15, 2019*, Manuel V. Hermenegildo and Atsushi Igarashi (Eds.). ACM, 54–60. https://doi.org/10.1145/3294032.3294083

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

[15] 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* (San Diego, California, USA) *(POPL '88)*. Association for Computing Machinery, New York, NY, USA, 12–27. https://doi.org/10.1145/73560.73562

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

[17] 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* (San Francisco, California, USA) *(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. We use Haskell's style list type `[I]` to denote a sequence of items `I`.

In Figure 7, we consider syntax and semantics of SSAFJ. For the semantics are expressed in denotational style. A global declaration environment maps class names to field declarations and method declaration. A local declaration environment maps local variable names to values. A memory store maps memory locations to values. The semantic function $\mathbb{E}_{\mathtt{fj}}[\![\cdot]\!]$ evaluates an FJ expression under a global enviromnent, a local environment and a store into a value with an updated memory store. The semantic function $\mathbb{S}_{\mathtt{fj}}[\![\cdot]\!]$ evaluates an FJ statement into a value, an updated local environment and an updated store. The details definition of these functions are omitted as they can be found in the literature. The semantic for SSAFJ works in a similar way except that in case of phi assignments, we need to resolve the LHS variable based on the given context (last visted label). Note that the rules (PhiRes) and (EmpPhiRes) in Figure 4 are encoded using a semantic function $\mathbb{F}_{\mathtt{ssa}}[\![\cdot]\!]$.

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.

Recall Lemma 4.2.

Let β be a variable mapping. Let $\mathtt{CTX}$ be an SSAFJ program context, such that $\beta \vdash_L \mathtt{CTX}$. Let s be an FJ statement, and $\mathtt{ctx}$ be an FJ program context such that $\mathtt{ctx}, \beta, \mathtt{CTX} \vdash s \Rightarrow B, \beta', \mathtt{CTX}'$. Then $\beta' \vdash_L \mathtt{CTX}'$.

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

Definition 3 (Consistent Global Environments). *Let* $genv \in \mathtt{GEnv}$ *and* $genv' \in \mathtt{GENV}$. *Then we say* $genv \vdash genv'$ *iff* $\forall (C, m) \in dom(genv) : \vdash genv(C, m) \Rightarrow genv'(C, m)$.

And Recall Definition 2.

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

The following lemma, the unabridged version of Lemma 4.3, 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.1 (Statement Conversion Consistency). *Let* $\mathtt{ctx}, \beta, \mathtt{CTX} \vdash s \Rightarrow B, \beta', \mathtt{CTX}'$. *Let* $genv \in \mathtt{GEnv}$ *and* $genv' \in \mathtt{GENV}$ *and* $genv \vdash genv'$. *Let* $lenv \in \mathtt{LEnv}$ *and* $lenv' \in \mathtt{LEnv}$ *and* $lenv \vdash_{(\mathtt{CTX},\beta)} lenv'$. *Let* str *be a memory store. Then* $\mathbb{S}_{\mathtt{fj}}[\![s]\!]\ genv\ lenv\ str = (v, lenv'', str'')$ *implies* $\mathbb{B}_{\mathtt{ssa}}[\![B]\!]\ genv'\ lenv'\ str = (v', lenv''', str''', l)$ *such that* $v = v'$ *and* $lenv'' \vdash_{(\beta',\mathtt{CTX}')} lenv'''$ *and* $str'' = str'''$.

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

$$
\text{(Context)}\quad \texttt{ctx} ::= \square \mid \texttt{ctx};\mid \texttt{ctx}; \overline{\mathbf{s}} \mid \mathbf{s}; \texttt{ctx} \mid \text{if } e\ \{\texttt{ctx}\} \text{ else } \{\overline{\mathbf{s}}\} \mid \text{if } \mathbf{e}\ \{\overline{\mathbf{s}}\} \text{ else } \{\texttt{ctx}\} \mid \text{while } e\ \{\texttt{ctx}\}
$$

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

Figure 6: Syntax and Semantics of FJ (excerpt)

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

$$
\begin{array}{rrcl}
\text{(CONTEXT)} & \texttt{CTX} & ::= & \square \mid \texttt{CTX};\mid \texttt{CTX}; \overline{\mathbf{B}} \mid \mathbf{B}; \texttt{CTX} \mid \text{if } \mathbf{E}\ \{\texttt{CTX}\} \text{ else } \{\overline{\mathbf{B}}\} \text{ join } \{\overline{\phi}\} \mid \text{if } \mathbf{E}\ \{\overline{\mathbf{B}}\} \text{ else } \{\texttt{CTX}\} \text{ join } \{\overline{\phi}\} \\
 & & & \mid \text{if } \mathbf{E}\ \{\overline{\mathbf{B}}\} \text{ else } \{\overline{\mathbf{B}}\} \text{ join}\{\blacksquare\} \mid \text{join } \{\blacksquare^b\} \text{ while } \mathbf{E}\ \{\overline{\mathbf{B}}\} \mid \text{join } \{\overline{\phi}\} \text{ while } \mathbf{E}\ \{\texttt{CTX}\}
\end{array}
$$

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

Figure 7: Syntax and Semantics of SSAFJ (excerpt)

Proof sketch. We apply the result of Lemma 4.2. We then prove the above lemma by induction over the derivation of $\texttt{ctx}, \beta, \texttt{CTX} \vdash s \Rightarrow B, \beta, \texttt{CTX}$. □

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 [10]. 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 $\texttt{ctx}[S]$ to denote $\{\texttt{ctx}[\texttt{ctx}'] \mid \texttt{ctx}' \in S\}$. Let P denote a set of pairs of program contexts, we write $\texttt{ctx}[P]$ to denote $\{(\texttt{ctx}[\texttt{ctx}_1], \texttt{ctx}[\texttt{ctx}_2]) \mid (\texttt{ctx}_1, \texttt{ctx}_2) \in P\}$.

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

$$
\begin{array}{l}
\mathbb{G}[\![\cdot]\!] :: \texttt{Statement} \to \{(\texttt{Context}, \texttt{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]\!]\} \\
\quad \text{in } \{(\square, \text{if } \mathbf{e} \{\square\} \text{else} \{\mathbf{s}\}), (\square, \text{if } \mathbf{e} \{\mathbf{s}\} \text{else} \{\square\})\} \cup \\
\quad (\text{if } \mathbf{e} \{\square\} \text{else} \{\overline{\mathbf{s}}\})[G_1] \cup (\text{if } \mathbf{e} \{\overline{\mathbf{s}}\} \text{else} \{\square\})[G_2] \\
\mathbb{G}[\![\text{while } e \ \{s\}]\!] = \text{let } \{G = \mathbb{G}[\![s]\!]; O = \mathbb{X}[\![s]\!]\} \\
\quad \text{in } \{(\square, \text{while } \mathbf{e} \ \{\square\})\} \cup (\text{while } \mathbf{e} \ \{\square\})[G] \\
\quad \cup \{((\text{while } \mathbf{e} \ \{\square\})[\texttt{ctx}], \square) \mid \texttt{ctx} \in O\} \\
\\
\mathbb{G}[\![\cdot]\!] :: [\texttt{Statement}] \to \{(\texttt{Context}, \texttt{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]\!]\} \\
\quad \text{in } \{(\square, (\square; \overline{\mathbf{s}}))\} \cup (\square; \overline{\mathbf{s}})[G_1] \cup \\
\quad \mathbf{s}; \square)[G_2] \cup \{(((\square; \overline{\mathbf{s}})[\texttt{ctx}], (\mathbf{s}; \square)) \mid \texttt{ctx} \in O_1\} \\
\\
\mathbb{X}[\![\cdot]\!] :: \texttt{Statement} \to \{\texttt{Context}\} \\
\mathbb{X}[\![a]\!] = \{\square\} \\
\mathbb{X}[\![\text{return } e]\!] = \{\} \\
\mathbb{X}[\![\text{if } e \{s_1\} \text{else} \{s_2\}]\!] = \{(\text{if } \mathbf{e} \{\square\} \text{else} \{\overline{\mathbf{s}}\})[\texttt{ctx}] \mid \texttt{ctx} \in \mathbb{X}[\![s_1]\!]\} \cup \\
\quad \{(\text{if } \mathbf{e} \{\overline{\mathbf{s}}\} \text{else} \{\square\})[\texttt{ctx}] \mid \texttt{ctx} \in \mathbb{X}[\![s_2]\!]\} \\
\mathbb{X}[\![\text{while } e \ \{s\}]\!] = \{\square\} \\
\\
\mathbb{X}[\![\cdot]\!] :: [\texttt{Statement}] \to \{\texttt{Context}\} \\
\mathbb{X}[\![s;]\!] = \mathbb{X}[\![s]\!] \\
\mathbb{X}[\![s_1; \overline{s_2}]\!] = \{\mathbf{s}; \texttt{ctx} \mid \texttt{ctx} \in \mathbb{X}[\![\overline{s_2}]\!]\}
\end{array}
$$

Figure 8: Control flow graph construction

Definition 4 (Path). *Let s be an FJ statement, G be a control flow graph such that $G = \mathbb{G}[\![s]\!]$. Then $p = [\texttt{ctx}_1, ..., \texttt{ctx}_n]$ 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 $\texttt{ctx}_1$ and $\texttt{ctx}_2$ be FJ program contexts. Then we say $\texttt{ctx}_1$ dominates $\texttt{ctx}_2$ iff for all path $p = [\square, ..., \texttt{ctx}_2]$ we have $\texttt{ctx}_1 \in p$.*

We write $\texttt{ctx} \gtrdot \texttt{ctx}'$ to denote $\texttt{ctx}$ dominating $\texttt{ctx}'$ and $\texttt{ctx} > > \texttt{ctx}'$ to denote $\texttt{ctx}$ strictly dominating $\texttt{ctx}'$.

Next we consider the following definition of dominance frontier adopted from Cytron's work [10].

Definition 6 (Dominance Frontier). *Let s be an FJ program statement and $\texttt{ctx}$ be an FJ program context. We define the dominance frontier of $\texttt{ctx}$ as follows.*

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

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

Lemma A.2 (DF is unique). *Let s be an FJ statement, $\texttt{ctx}_0$ be an FJ program context. Then for all $\texttt{ctx} \in DF(\texttt{ctx}_0 \,|\, s)$ and $\texttt{ctx}' \in DF(\texttt{ctx}_0 \,|\, s)$, we have $\texttt{ctx} = \texttt{ctx}'$.*

To prove the above lemma we need a supporting lemma

Lemma A.3 (Box Dominates all). *Let $\texttt{ctx}$ be an FJ program context. Then $\square$ dominates $\texttt{ctx}$.*

Proof. It follows from Definition 5. □

Now we can prove Lemma A.2.

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

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

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

- Case a: $DF(\texttt{ctx}_0 \,|\, a) = \{\}$. 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

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

  By applying the definition of control flow graph, we have

$$\begin{array}{l}
\mathbb{G}[\![\text{if } e \ \{\overline{s_1}\} \text{ else } \{\overline{s_2}\}]\!] = \text{let } \{G_1 = \mathbb{G}[\![\overline{s_1}]\!]; G_2 = \mathbb{G}[\![\overline{s_2}]\!]\} \\
\text{in } \{(\square, \text{if } \mathbf{e} \{\square\} \text{else} \{\overline{\mathbf{s}}\}), (\square, \text{if } \mathbf{e} \{\overline{\mathbf{s}}\} \text{else} \{\square\})\} \cup \quad (1) \\
(\text{if } \mathbf{e} \ \{\square\} \text{ else } \{\overline{\mathbf{s}}\})[G_1] \cup \quad (2) \\
(\text{if } \mathbf{e} \ \{\overline{\mathbf{s}}\} \text{ else } \{\square\})[G_2] \quad (3)
\end{array}$$

  By Lemma A.3, we can ignore edges coming from (1). Note that $\texttt{ctx}_0$ 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 $\{\overline{s}\}$: Applying Definition 6, we have

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

  By applying the definition of control flow graph, we have

$$\begin{array}{l}
\mathbb{G}[\![\text{while } e \ \{\overline{s}\}]\!] = \text{let } \{G = \mathbb{G}[\![\overline{s}]\!]; O = \mathbb{X}[\![\overline{s}]\!]\} \\
\text{in } \{(\square, \text{while } \mathbf{e} \ \{\square\})\} \cup \quad (4) \\
(\text{while } \mathbf{e} \ \{\square\})[G] \cup \quad (5) \\
\{((\text{while } \mathbf{e} \ \{\square\})[\texttt{ctx}], \square) \mid \texttt{ctx} \in O\} \quad (6)
\end{array}$$

  By Lemma A.3, 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 $\square$. We have verified this case.
- Case $s_1; \overline{s_2}$: Applying Definition 6, we have

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

  By applying the definition of control flow graph, we have

$$\begin{array}{l}
\mathbb{G}[\![s_1; \overline{s_2}]\!] = \text{let } \{G_1 = \mathbb{G}[\![s_1]\!]; G_2 = \mathbb{G}[\![\overline{s_2}]\!]; O = \mathbb{X}[\![s_1]\!]\} \\
\text{in } \{(\square, (\square; \overline{\mathbf{s}}))\} \cup \quad (7) \\
(\square; \overline{\mathbf{s}})[G_1] \cup \quad (8) \\
(\mathbf{s}; \square)[G_2] \cup \quad (9) \\
\{(\square; \overline{\mathbf{s}})[\texttt{ctx}], (\mathbf{s}; \square) \mid \texttt{ctx} \in O\} \quad (10)
\end{array}$$

  By Lemma A.3, 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 $(\mathbf{s}; \square)$. We have verified this case.

□

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 $\texttt{ctx}$ be an FJ program context. We define the dominance frontier handle of $\texttt{ctx}$ as follows.*

$$\begin{aligned}&DF_h(\texttt{ctx}\,|\,s) = \{<\texttt{ctx}_1, \texttt{ctx}_1'>\mid \exists(\texttt{ctx}_1, \texttt{ctx}_2) \in \mathbb{G}[\![s]\!]\\ &\wedge\exists(\texttt{ctx}_1', \texttt{ctx}_2) \in \mathbb{G}[\![s]\!] : \texttt{ctx} \gtrdot \texttt{ctx}_1 \wedge \neg(\texttt{ctx} \gg \texttt{ctx}_1')\}\\ &\cup\{<\texttt{ctx}_1, \texttt{ctx}_1'>\mid \texttt{ctx}_1 \in \mathbb{X}[\![s]\!] \wedge \texttt{ctx}_1' \in \mathbb{X}[\![s]\!]\\ &\wedge\texttt{ctx} \gtrdot \texttt{ctx}_1 \wedge \neg(\texttt{ctx} \gg \texttt{ctx}_1')\}\end{aligned}$$

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

We consider the follow definition of iterative dominance frontier adopted from Cytron's work [10].

Definition 8 (Iterative Dominance Frontier).

$$\begin{aligned}DF(\texttt{ctx}\,|\,s)^0 &= DF(\texttt{ctx}\,|\,s)\\ DF(\texttt{ctx}\,|\,s)^{n+1} &= \text{let } F = DF(\texttt{ctx}\,|\,s)^n\\ &\quad\text{in } DF(F\,|\,s) \cup DF(\texttt{ctx}\,|\,s)^n\end{aligned}$$

where $DF(F\,|\,s) = \bigcup_{\texttt{ctx}\in F} DF(\texttt{ctx}\,|\,s)$

The iterative dominance frontier captures the dominance frontier of the input context $\texttt{ctx}$ and the cascaded dominance frontiers. We define the handle of iterative dominance frontiers.

Definition 9 (Iterative Dominance Frontier Handle).

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

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

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

Lemma A.4 (DF-DFH correspondence). *Let s be an FJ statement such that $\mathbb{X}[\![s]\!] = \{\}$ and $\texttt{ctx}_0$ be an FJ program context referring to some location in s. Then*

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

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

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

- Case 0. We would like to show $\texttt{ctx}_2 \in DF(\texttt{ctx}_0\,|\,s)$ implies $\exists\ <\texttt{ctx}_1, \texttt{ctx}_1'>\in DF_h(\texttt{ctx}_0\,|\,s)$ such that $(\texttt{ctx}_1, \texttt{ctx}_2) \in \mathbb{G}[\![s]\!]$ and $(\texttt{ctx}_1', \texttt{ctx}_2) \in \mathbb{G}[\![s]\!]$.
  By Definition 6, we have $\exists(\texttt{ctx}_1'', \texttt{ctx}_2) \in \mathbb{G}[\![s]\!]$ such that $\texttt{ctx}_0 \gtrdot \texttt{ctx}_1''$ and $\neg(\texttt{ctx}_0 \gg \texttt{ctx}_2)$. It follows that there exists a path $p = \square, ..., \texttt{ctx}_2$ such that $\texttt{ctx}_0 \in p$. By Lemma A.3, $\texttt{ctx}_0 \neq \square$ and by definition 6, $\texttt{ctx}_2 \neq \square$. Thus $p = \square, ..., \texttt{ctx}_1''', \texttt{ctx}_2$. Let $p = p', \texttt{ctx}_2$. It follows that $\texttt{ctx}_0 \notin p'$, therefore $\neg(\texttt{ctx}_0 \gg \texttt{ctx}_1''')$. We conclude that we found $<\texttt{ctx}_1'', \texttt{ctx}_1'''>\in DF_h(\texttt{ctx}_0\,|\,s)$ and $(\texttt{ctx}_1'', \texttt{ctx}_2) \in \mathbb{G}[\![s]\!]$ and $(\texttt{ctx}_1''', \texttt{ctx}_2) \in \mathbb{G}[\![s]\!]$. We have verified this case.
- Case $n+1$. We would like to show that $\texttt{ctx}_2 \in DF(\texttt{ctx}_0\,|\,s)^{n+1}$ implies that $\exists\ <\texttt{ctx}_1, \texttt{ctx}_1'>\in DF_h(\texttt{ctx}_0\,|\,s)^{n+1}$ such that $(\texttt{ctx}_1, \texttt{ctx}_2) \in \mathbb{G}[\![s]\!]$ and $(\texttt{ctx}_1', \texttt{ctx}_2) \in \mathbb{G}[\![s]\!]$.
  By Definition 8, we have

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

  By Definition 9, we have

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

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

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

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

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

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

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

  such that $\texttt{ctx}_2 \in DF(ctx_2'\,|\,s)$ and $\texttt{ctx}_2 \notin DF(\texttt{ctx}_0\,|\,s)^n$.
  By applying I.H. to (3), we must be able find $<\texttt{ctx}_1'', \texttt{ctx}_1'''>\in DF_h(\texttt{ctx}_0\,|\,s)^n$ such that $(\texttt{ctx}_1'', \texttt{ctx}_2') \in \mathbb{G}[\![s]\!]$ and $(\texttt{ctx}_1''', \texttt{ctx}_2') \in \mathbb{G}[\![s]\!]$. By Lemma A.2, the $\texttt{ctx}_2''$ in (2) must be the same as $\texttt{ctx}_2'$.
  From (3), we can apply the same steps used in Case 0 to show that $\exists\ <\texttt{ctx}_1, \texttt{ctx}_1'>\in DF_h(\texttt{ctx}_2'\,|\,s)$ such that $(\texttt{ctx}_1, \texttt{ctx}_2) \in \mathbb{G}[\![s]\!]$ and $(\texttt{ctx}_1', \texttt{ctx}_2) \in \mathbb{G}[\![s]\!]$. We have verified this case.

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

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

By Definition 7, we have

$$DF_h(\texttt{ctx}_0 \mid s) = \{<\texttt{ctx}_1, \texttt{ctx}_1'> \mid (\texttt{ctx}_1, \texttt{ctx}_2) \in \mathbb{G}[\![s]\!] \wedge (\texttt{ctx}_1', \texttt{ctx}_2) \in \mathbb{G}[\![s]\!] \wedge \texttt{ctx}_0 \gtrdot \texttt{ctx}_1 \wedge \neg(\texttt{ctx}_0 \gg \texttt{ctx}_1')\} \quad (4)$$
$$\cup$$
$$\{<\texttt{ctx}_1, \texttt{ctx}_1'> \mid \texttt{ctx}_1 \in \mathbb{X}[\![s]\!] \wedge \texttt{ctx}_1' \in \mathbb{X}[\![s]\!] \wedge \texttt{ctx}_0 \gtrdot \texttt{ctx}_1 \wedge \neg(\texttt{ctx}_0 \gg \texttt{ctx}_1')\} \quad (5)$$

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

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

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

By Definition 8, we have

$$DF(\texttt{ctx}_0 \mid s)^{n+1} = \text{let } F = DF(\texttt{ctx}_0 \mid s)^n \text{ in } DF(F \mid s) \cup DF(\texttt{ctx}_0 \mid s)^n$$

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

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

where $H = \{\texttt{ctx}_2'' \mid (\texttt{ctx}_1, \texttt{ctx}_2'') \in \mathbb{G}[\![s]\!] \wedge (\texttt{ctx}_1', \texttt{ctx}_2'') \in \mathbb{G}[\![s]\!] \wedge <ctx_1, \texttt{ctx}_1'> \in DF_h(\texttt{ctx}_0 \mid s)^n$
We would like to find $\texttt{ctx}_2 \in DF(F \mid s) \cup DF(\texttt{ctx}_0 \mid s)^n$.
Recall that

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

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

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

and

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

By I.H. to (8), we must be able to find $\texttt{ctx}_2' \in DF(\texttt{ctx}_0 \mid s)^n$ such that $(\texttt{ctx}_1'', \texttt{ctx}_2') \in \mathbb{G}[\![s]\!]$ and $(\texttt{ctx}_1''', \texttt{ctx}_2') \in \mathbb{G}[\![s]\!]$.
By Lemma A.2, we conclude that $\texttt{ctx}_2' = \texttt{ctx}_2''$.
Now from (9), we can apply the same set of steps as in Case 0 to show that $\exists \texttt{ctx}_2 \in DF(\texttt{ctx}_2'' \mid s)$ such that $(\texttt{ctx}_1, \texttt{ctx}_2) \in \mathbb{G}[\![s]\!]$ and $(\texttt{ctx}_1', \texttt{ctx}_2) \in \mathbb{G}[\![s]\!]$. 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. $\texttt{phiparam}\ v\ B\ \texttt{CTX}\ \beta$ 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. $\texttt{CTX}_1$. This is because the contexts extracted by 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 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.5 (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* $\square, \{\}, \square \vdash s \Rightarrow B, \beta, \texttt{CTX}$ *Let v be a variable such that $(v, \texttt{ctx}', \texttt{CTX}', V) \in \beta$ and* $\texttt{CTX}'$ *is not a join context. Then* $(\texttt{phiparam}\ v\ B\ \texttt{CTX}'\ \beta) \subseteq [\![fst(DF_h(\texttt{ctx}' \mid s)^n)]\!]$.

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 $\texttt{CTX}, \texttt{CTX}'$ be SSAFJ program contexts such that $\texttt{ctx}, \beta, \texttt{CTX} \vdash s \Rightarrow B, \beta', \texttt{CTX}'$.. Let v be a variable such that $(v, \texttt{ctx}'', \texttt{CTX}'', V) \in (\beta' - \beta)_{|\texttt{ctx}}$ where $\texttt{CTX}''$ is not a join program context. Then $(\texttt{phiparam}\ v\ B\ \texttt{CTX}''\ (\beta' - \beta)_{|\texttt{ctx}}) \subseteq [\![fst(DF_h(\texttt{ctx}'' \mid s)^n)]\!]$.

We prove the lemma by induction over s.

- Case a. $\texttt{phiparam}\ v\ A\ \texttt{CTX}'\ \beta = \{\}$ trivial.
- Case return e. $\texttt{phiparam}\ v\ (\text{return } E)\ \texttt{CTX}'\ \beta = \{\}$ trivial.

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

Figure 9: Phi Parameter Extraction

- Case if $e\ \{\overline{s_1}\}$ else$\{\overline{s_2}\}$. Recall that

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

And the definition of `phiparam` for the if-else statement with some renaming

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

We consider the following sub-cases.

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

$$
\begin{array}{c}
(\texttt{phiparam}\ v\ \overline{B_1}\ \texttt{CTX}'''\ (\beta_1 - \beta_0)_{|\text{if}\mathbf{e}\{\square\}\text{else}\{\overline{\mathbf{s}}\}}) \\
\subseteq \\
[\![fst(DF_h(\texttt{ctx}''' | \overline{s_1}))^n)]\!]
\end{array}
$$

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

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

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 $\mathtt{CTX}_1$ is not a join context, it follows that $\mathbb{X}[\![\overline{s_1}]\!] = \{\mathtt{ctx}_1\}$ such that

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

We consider two possibilities.

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

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

Note that $\mathtt{ctx}_2$ is not dominated by $\mathtt{ctx}''$. We verified this case.

∗ Sub-subcase $DF_h(\mathtt{ctx}''' \mid \overline{s_1}) \neq \{\}$, By Definition 7,

$$\begin{aligned}&DF_h(\mathtt{ctx}''' \mid \overline{s_1})^0 =\\ &DF_h(\mathtt{ctx}''' \mid \overline{s_1}) =\\ &\{<\mathtt{ctx}_1'', \mathtt{ctx}_1'''>\mid (\mathtt{ctx}_1'', \mathtt{ctx}_2) \in \mathbb{G}[\![\overline{s_1}]\!]\\ &\wedge(\mathtt{ctx}_1''', \mathtt{ctx}_2) \in \mathbb{G}[\![\overline{s_1}]\!]\\ &\wedge \mathtt{ctx}''' \gtrdot \mathtt{ctx}_1'' \wedge \neg(\mathtt{ctx}''' \gg \mathtt{ctx}_1''')\} \quad (3)\\ &\cup\\ &\{<\mathtt{ctx}_1'', \mathtt{ctx}_1'''>\mid \mathtt{ctx}_1'' \in \mathbb{X}[\![\overline{s_1}]\!] \wedge \mathtt{ctx}_2'' \in \mathbb{X}[\![\overline{s_2}]\!]\\ &\wedge \mathtt{ctx}''' \gtrdot \mathtt{ctx}_1'' \wedge \neg(\mathtt{ctx}''' \gg \mathtt{ctx}_1''')\} \quad (4)\end{aligned}$$

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

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

Note that $\mathtt{ctx}_2$ is not dominated by $\mathtt{ctx}''$. We verified this case.

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

• Case while $e\ \{\overline{s}\}$. Recall that

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

And the definition of phiparam for while statement with some renaming

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

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

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

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

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

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

$$(\text{while } \mathbf{e}\ \{\square\})[DF_h(\mathtt{ctx}''' \mid \overline{s})^n] \subseteq DF_h(\mathtt{ctx}'' \mid \text{while } e\ \{\overline{s}\})^n$$

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 $\mathtt{CTX}_2$ is not a join context, it follows that $\mathbb{X}[\![\overline{s}]\!] = \{\mathtt{ctx}_2\}$ such that

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

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

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

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

And

$$\begin{array}{l}\texttt{phiparam}\ v\ (B;\overline{B})\ \texttt{CTX}'\ (\beta_2-\beta) = \\ \quad \texttt{case CTX}'\texttt{of} \\ \quad\quad (\square;\overline{\textbf{B}})[\texttt{CTX}''] \\ \quad\quad \mid (v,\texttt{ctx}'',\texttt{CTX}'',V') \in (\beta_2-\beta)_{|\square;\overline{\textbf{B}}} \rightarrow \quad --(6.1) \\ \quad\quad\ \texttt{phiparam}\ v\ B\ \texttt{CTX}''\ (\beta_2-\beta)_{|\square;\overline{\textbf{B}}} \\ \quad\quad (\textbf{B};\square)[\texttt{CTX}''] \\ \quad\quad \mid (v,\texttt{ctx}'',\texttt{CTX}'',V') \in (\beta_2-\beta)_{|\textbf{B};\square} \rightarrow \quad --(6.2) \\ \quad\quad\ \texttt{phiparam}\ v\ \overline{B}\ \texttt{CTX}''\ (\beta_2-\beta)_{|\textbf{B};\square}\end{array}$$

We consider the following sub-cases

– Subcase (6.1). Note that

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

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

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

where $(\square;\overline{\textbf{s}})[\texttt{ctx}''] = \texttt{ctx}'$. By Definition 7 and Definition 9 we find that

$$(\square;\overline{\textbf{B}})[DF_h(\texttt{ctx}''\mid s)^n] \subseteq DF_h(\texttt{ctx}'\mid s;\overline{s})^n$$

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

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

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

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

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

$$(\textbf{B};\square)[DF_h(\texttt{ctx}''\mid\overline{s})^n] \subseteq DF_h(\texttt{ctx}'\mid s;\overline{s})^n$$

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

□

B APPLICATION

B.1 Unrolling Structured SSA into GCL

In Figure 10, we report syntax of GCL statement adopted from [11]. 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 $\texttt{gcl}[\![\cdot]\!]$ converts a SSAFJ statement block into a GCL statement recursively. Function $\texttt{expand}[\![\cdot]\!]$ unrolls the loop with the fixed number of times. Applying $\texttt{gcl}[\![\cdot]\!]$ 1 to the running example in Figure 1 (c), we have

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

$$\begin{array}{l}\texttt{gcl}[\![\cdot]\!] :: \texttt{STATEMENT} \rightarrow \mathit{Int} \rightarrow \texttt{GCL} \\ \texttt{gcl}[\![\texttt{if}\ E\{\overline{S_1}\}\texttt{else}\{\overline{S_2}\}\ \texttt{join}\{\overline{\phi}\}]\!]\ n = \\ \quad (\mathit{assume}\ E;\texttt{gcl}[\![S_1]\!]\ n;\texttt{fsta}[\![\overline{\phi}]\!])\square(\mathit{assume}\ \neg E;\texttt{gcl}[\![S_2]\!]\ n; \\ \quad \texttt{snda}[\![\overline{\phi}]\!]) \\ \texttt{gcl}[\![\texttt{join}\{\overline{\phi}\}\ \texttt{while}\ E\ I\{\overline{S}\}]\!]\ n = \\ \quad \texttt{let}\ \overline{\mathcal{S}} = \texttt{gcl}[\![\overline{S}]\!]\ n\ \texttt{in}\ \texttt{expand}[\![\overline{\phi}]\!]\ E\ I\ \overline{\mathcal{S}}\ n \\ \texttt{gcl}[\![x = E]\!]\ n = x = E \\ \texttt{gcl}[\![\texttt{return}\ E]\!]\ n = \mathit{assume}\ \mathit{true}\end{array}$$

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

$$\begin{array}{l}\texttt{fsta}[\![\cdot]\!] :: [\texttt{Phi}] \rightarrow [\texttt{GCL}] \\ \texttt{fsta}[\![\{\}]\!] = \mathit{nop} \\ \texttt{fsta}[\![x = \mathit{phi}(x_1,x_2);\overline{\phi}]\!] = x = x_1;\texttt{fsta}[\![\overline{\phi}]\!]\end{array}$$

$$\begin{array}{l}\texttt{snda}[\![\cdot]\!] :: [\texttt{Phi}] \rightarrow [\texttt{GCL}] \\ \texttt{snda}[\![\{\}]\!] = \mathit{nop} \\ \texttt{snda}[\![x = \mathit{phi}(x_1,x_2);\overline{\phi}]\!] = x = x_2;\texttt{snda}[\![\overline{\phi}]\!]\end{array}$$

$$\begin{array}{l}\texttt{subst}_1[\![\cdot]\!] :: [\texttt{Phi}] \rightarrow \mathit{Int} \rightarrow \{(\texttt{Var},\texttt{Var})\} \\ \texttt{subst}_1[\![\{\}]\!]\ n = [] \\ \texttt{subst}_1[\![x = \mathit{phi}(x_1,x_2);\overline{\phi}]\!] = [x^n/x, x_2^n/x_2] \cup \texttt{subst}_1[\![\overline{\phi}]\!]\ n\end{array}$$

$$\begin{array}{l}\texttt{subst}_2[\![\cdot]\!] :: [\texttt{Phi}] \rightarrow \{(\texttt{Var},\texttt{Var})\} \rightarrow \{(\texttt{Var},\texttt{Var})\} \\ \texttt{subst}_2[\![\{\}]\!]\ \theta_1 = [] \\ \texttt{subst}_2[\![x = \mathit{phi}(x_1,x_2);\overline{\phi}]\!]\ \theta_1 = [\theta_1(x_2)/x_1] \cup \texttt{subst}_2[\![\overline{\phi}]\!]\ \theta_1\end{array}$$

Figure 10: Unrolling Structured SSA into GCL

$$\begin{array}{l}s_0 = 0;\ \ i_0 = 0; \\ \theta_1(i_1 = i_0; s_1 = s_0; \mathit{assume}\ \ (i_1 < N) \wedge s_1 <= i_1 * (i_1-1)/2; \\ s_2 = s_1 + i_1; i_2 = i_1 + 1); \mathit{assert}\ \ \mathit{false}; \\ \square\ \ i_1 = i_0; s_1 = s_0; \mathit{assume}\ \ (i_1 >= N) \wedge s_1 <= i_1 * (i_1-1)/2;\end{array}$$

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 $\texttt{expand}[\![\cdot]\!]$ 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).

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

where

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

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* $\texttt{gcl}[\![S]\!]\ n = \mathcal{S}$. *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 $\texttt{flatten}[\![\cdot]\!]$ 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 $\texttt{flatten}[\![\cdot]\!]$ 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 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 nnsuccs finds the set of nearest non-nested non-join successor of CTX. Function nnsucc is partial and expects the 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 $\texttt{flatten}[\![\cdot]\!]$ 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).

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

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

$$
\begin{array}{l}
\texttt{merge} :: [\texttt{Phi}] \to [\texttt{Phi}] \to [\texttt{Phi}] \\
\texttt{merge}\ \overline{\phi_1}\ \overline{\phi_2} = \\
\quad \text{let } \vartheta_1 = [(\texttt{CTX}_1 : x_1, ..., \texttt{CTX}_n : x_n)/\texttt{CTX}' : x \mid \\
\qquad x = \mathit{phi}(\texttt{CTX}_1 : x_1, ..., \texttt{CTX}_n : x_n) \in \overline{\phi_1}) \wedge (\texttt{CTX}' : x) \in \mathit{params}(\overline{\phi_2}) \\
\quad \text{in } \vartheta_1(\overline{\phi_2})
\end{array}
$$

$$
\begin{array}{l}
\texttt{succs} :: \{(\texttt{CONTEXT}, \texttt{CONTEXT})\} \to \texttt{CONTEXT} \to \{\texttt{CONTEXT}\} \\
\texttt{succs}\ G\ \texttt{CTX} = \text{let } \mathit{cs}_\square = \{\texttt{CTX}' \mid (\texttt{CTX}, \texttt{CTX}') \in G \wedge \neg \mathit{isJoin}(\texttt{CTX}')\} \\
\qquad \mathit{cs}_\blacksquare = \{\texttt{CTX}' \mid (\texttt{CTX}, \texttt{CTX}') \in G \wedge \mathit{isJoin}(\texttt{CTX}')\} \\
\qquad \text{in } \mathit{cs}_\square \cup (\text{succs } G\ \mathit{cs}_\blacksquare)
\end{array}
$$

$$
\begin{array}{l}
\texttt{succs} :: \{(\texttt{CONTEXT}, \texttt{CONTEXT})\} \to \{\texttt{CONTEXT}\} \to \{\texttt{CONTEXT}\} \\
\texttt{succs}\ G\ \mathit{cs} = \{\texttt{CTX}' \mid \texttt{CTX} \in \mathit{cs} \wedge \texttt{CTX}' \in \text{succs } G\ \texttt{CTX}\}
\end{array}
$$

$$
\begin{array}{l}
\texttt{nnsuccs} :: \{(\texttt{CONTEXT}, \texttt{CONTEXT})\} \to \texttt{CONTEXT} \to \{\texttt{CONTEXT}\} \\
\texttt{nnsuccs}\ G\ \texttt{CTX} = \\
\quad \{\texttt{CTX}' \mid \texttt{CTX}' \in \text{succs } G\ \texttt{CTX} \wedge \neg(\exists \texttt{CTX}''.\texttt{CTX}[\texttt{CTX}''] = \texttt{CTX}')\}
\end{array}
$$

$$
\begin{array}{l}
\texttt{nnsucc} :: \{(\texttt{CONTEXT}, \texttt{CONTEXT})\} \to \texttt{CONTEXT} \to \texttt{CONTEXT} \\
\texttt{nnsucc}\ G\ \texttt{CTX} = \text{case nnsuccs } G\ \texttt{CTX of} \{\texttt{CTX}'\} \to \texttt{CTX}'
\end{array}
$$

Figure 11: Flattening SSAFJ into Unstructured SSA

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* $\texttt{flatten}[\![\overline{B}]\!]\ [\,]\ G$ *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 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 $\mathtt{flatten}[\![\overline{B}]\!]\ \overline{\phi}\ G = (\overline{U}, \overline{\phi'})$. Let $it = \mathtt{buildIT}\ \overline{U}$. 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 $\mathsf{CTX}'_u \overset{G}{\preceq} \mathsf{CTX}'_s$.

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

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

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

  and

$$\begin{array}{l}\overline{\mathbb{B}}_{\mathsf{ssa}}[\![\mathsf{CTX} : \mathsf{join}\ \{\overline{\phi'}\}\mathsf{while}\ E\ \{\overline{B}\}; \overline{B'}]\!]\ \mathsf{CTX}_s\ genv\ lenv_s\ str \longrightarrow \\ \mathsf{case}\ \mathbb{B}_{\mathsf{ssa}}[\![\mathsf{CTX} : \mathsf{join}\ \{\overline{\phi'}\}\mathsf{while}\ E\ \{\overline{B}\}]\!]\ \mathsf{CTX}_s\ genv\ lenv_s\ str\mathsf{of} \\ \quad (\_, lenv'_s, str', \mathsf{CTX}'_s) \rightarrow \overline{\mathbb{B}}_{\mathsf{ssa}}[\![\overline{B'}]\!]\ \mathsf{CTX}'_s\ genv\ lenv'_s\ str' \quad --(2)\end{array}$$

  where

$$\begin{array}{l}\mathbb{B}_{\mathsf{ssa}}[\![\mathsf{CTX} : \{\mathsf{join}\ \{\overline{\phi}\}\ \mathsf{while}\ E\ \{\overline{B}\}\}]\!]\ \mathsf{CTX}_s\ genv\ lenv_s\ str \\ \longrightarrow \\ \quad \mathsf{let}\ lenv^1_s\ =\ \mathbb{F}_{\mathsf{ssa}}[\![\overline{\phi}]\!]\ \mathsf{CTX}_s\ lenv_s \\ \quad \mathsf{in\ case}\ \mathbb{E}_{\mathsf{ssa}}[\![E]\!]\ genv\ lenv^1_s\ str\ \mathsf{of} \\ \quad (false, str^2_s) \rightarrow (\mathsf{null}, lenv^2_s, str^2_s, \mathsf{CTX}[\mathsf{join}\{\blacksquare\}\mathsf{while}\ \mathbf{E}\ \{\overline{\mathbf{B}}\}]) \\ \quad (true, str^2_s) \rightarrow \\ \quad \mathsf{case}\ \overline{\mathbb{B}}_{\mathsf{ssa}}[\![\overline{B}]\!]\ \mathsf{CTX}[\mathsf{join}\{\blacksquare\}\mathsf{while}\ \mathbf{E}\ \{\overline{\mathbf{B}}\}]\ genv\ lenv^1_s\ str^2_s\mathsf{of} \\ \quad\quad \{(v^3_s, lenv^2_s, str^3_s, \mathsf{CTX}^2_s) \rightarrow \\ \quad\quad \mathbb{B}_{\mathsf{ssa}}[\![\mathsf{CTX} : \{\ \mathsf{join}\ \{\overline{\phi}\}\ \mathsf{while}\ E\ \{\overline{B}\}\}]\!]\ \mathsf{CTX}^2_s\ genv\ lenv^2_s\ str^3_s\}\end{array}$$

  Recall

$$\begin{array}{l}\mathbb{U}[\![\mathsf{CTX} : \mathtt{merge}\ \overline{\phi}\ (\mathtt{merge}\ \overline{\phi''}\ \overline{\phi'}); \\ \quad \mathsf{if}\ \neg(E)\mathsf{goto}\ (\mathsf{nnsucc}\ G\ \mathsf{CTX}); \overline{U}; \\ \quad \mathsf{goto}\ \mathsf{CTX}; \overline{U'}]\!]\ \mathsf{CTX}_u\ genv\ lenv_u\ str\ it \\ \longrightarrow \\ \mathbb{U}[\![\mathsf{if}\ \neg(E)\mathsf{goto}\ (\mathsf{nnsucc}\ G\ \mathsf{CTX}); \overline{U}; \\ \mathsf{goto}\ \mathsf{CTX}; \overline{U'}]\!]\ \mathsf{CTX}\ genv\ lenv^1_u\ str\ it \quad --(3)\end{array}$$

  where $lenv^1_u = \mathbb{F}_{\mathsf{ssa}}[\![\mathtt{merge}\ \overline{\phi}\ (\mathtt{merge}\ \overline{\phi''}\ \overline{\phi'})]\!]\ \mathsf{CTX}_u\ lenv_u$
  We consider the possible value of $lenv^1_u$. 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 $\mathsf{CTX}_u \overset{G}{\preceq} \mathsf{CTX}_s$. There are two sub-cases:
  – Subcase 1: CTX_u and CTX_s are from the preceding block.

$$\begin{array}{lcl} lenv^1_u & = & \mathbb{F}_{\mathsf{ssa}}[\![\mathtt{merge}\ \overline{\phi}\ (\mathtt{merge}\ \overline{\phi''}\ \overline{\phi'})]\!]\ \mathsf{CTX}_u\ lenv_u \\ & & \text{drop } \overline{\phi''} \text{ since both contexts} \\ & & \text{are from the preceding block.} \\ & = & \mathbb{F}_{\mathsf{ssa}}[\![\mathtt{merge}\ \overline{\phi}\ \overline{\phi'}]\!]\ \mathsf{CTX}_u\ lenv_u \\ & & \text{By Lemma C.1.} \\ & & \text{Note that } \mathsf{CTX}_u \in params(\overline{\phi}), \mathsf{CTX}_s \in params(\overline{\phi'}). \\ & = & \mathbb{F}_{\mathsf{ssa}}[\![\overline{\phi'}]\!]\ \mathsf{CTX}_s\ (\mathbb{F}_{\mathsf{ssa}}[\![\overline{\phi}]\!]\ \mathsf{CTX}_u\ lenv_u) \\ & & \text{From the premise.} \\ & = & \mathbb{F}_{\mathsf{ssa}}[\![\overline{\phi'}]\!]\ \mathsf{CTX}_s\ lenv_s \\ & = & lenv^1_s \end{array}$$

$$
\begin{array}{l}
\mathbb{G}[\![\cdot]\!] :: \texttt{STATEMENT} \rightarrow \{(\texttt{CONTEXT}, \texttt{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 } \mathbf{E}\ \{\square\}\text{else}\{\mathbf{B}\}\text{join}\{\overline{\phi}\}), (\square, \text{if } \mathbf{E}\ \{\mathbf{B}\}\text{else}\{\square\})\text{join}\{\overline{\phi}\}\} \cup \\
\quad (\text{if } \mathbf{E}\{\square\}\text{else}\{\overline{\mathbf{B}}\}\text{join}\{\overline{\phi}\})\,[G_1] \cup (\text{if } \mathbf{E}\{\overline{\mathbf{B}}\}\text{else}\{\square\}\text{join}\{\overline{\phi}\})\,[G_2] \cup \\
\quad \{(\text{if } \mathbf{E}\{\square\}\text{else}\{\overline{\mathbf{B}}\}\text{join}\{\overline{\phi}\}\,[\texttt{CTX}], \text{if } \mathbf{E}\{\mathbf{B}\}\text{else}\{\overline{\mathbf{B}}\}\text{join}\{\blacksquare\}) \mid \texttt{CTX} \in \mathbb{X}[\![\overline{B_1}]\!]\} \cup \\
\quad \{(\text{if } \mathbf{E}\{\overline{\mathbf{B}}\}\text{else}\{\square\}\text{join}\{\overline{\phi}\}\,[\texttt{CTX}], \text{if } \mathbf{E}\{\mathbf{B}\}\text{else}\{\overline{\mathbf{B}}\}\text{join}\{\blacksquare\}) \mid \texttt{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 } \mathbf{E}\ \{\overline{\mathbf{B}}\}), (\text{join } \{\blacksquare\}\text{while } \mathbf{E}\ \{\overline{\mathbf{B}}\}, \text{join } \{\overline{\phi}\}\text{while } \mathbf{E}\ \{\square\})\} \cup \\
\quad (\text{join } \{\overline{\phi}\}\text{while } \mathbf{E}\ \{\square\})\,[G] \cup \{((\text{join } \{\overline{\phi}\}\text{while } \mathbf{E}\ \{\square\})\,[\texttt{CTX}], \text{join } \{\blacksquare\}\text{while } \mathbf{E}\ \{\overline{\mathbf{B}}\}) \mid \texttt{CTX} \in O\} \\
\mathbb{G}[\![\cdot]\!] :: \texttt{BLOCK} \rightarrow \{(\texttt{CONTEXT}, \texttt{CONTEXT})\} \\
\mathbb{G}[\![l : \{S\}]\!] = \mathbb{G}[\![S]\!] \\
\\
\mathbb{G}[\![\cdot]\!] :: [\texttt{BLOCK}] \rightarrow \{(\texttt{CONTEXT}, \texttt{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}})\,[\texttt{CTX}], (\mathbf{B}; \square)) \mid \texttt{CTX} \in O_1\} \\
\\
\mathbb{X}[\![\cdot]\!] :: \texttt{STATEMENT} \rightarrow \{\texttt{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 } \mathbf{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 } \mathbf{E}\{\overline{\mathbf{B}}\}\} \\
\\
\mathbb{X}[\![\cdot]\!] :: \texttt{BLOCK} \rightarrow \{\texttt{CONTEXT}\} \\
\mathbb{X}[\![l : \{S\}]\!] = \mathbb{X}[\![S]\!] \\
\\
\mathbb{X}[\![\cdot]\!] :: [\texttt{BLOCK}] \rightarrow \{\texttt{CONTEXT}\} \\
\mathbb{X}[\![B;]\!] = \mathbb{X}[\![B]\!] \\
\mathbb{X}[\![B_1; \overline{B_2}]\!] = \{\mathbf{B}; \texttt{CTX} \mid \texttt{CTX} \in \mathbb{X}[\![\overline{B_2}]\!]\}
\end{array}
$$

Figure 12: CFG construction of SSAFJ

$$
\begin{array}{l}
\mathbb{U}[\![\cdot]\!] :: [\texttt{UnSSA}] \rightarrow \texttt{CONTEXT} \rightarrow \texttt{GEnv} \rightarrow \texttt{LEnv} \rightarrow \texttt{Store} \rightarrow \{(\texttt{CONTEXT}, [\texttt{UnSSA}])\} \rightarrow (\texttt{Value}, \texttt{LEnv}, \texttt{Store}, \texttt{CONTEXT}) \\
\mathbb{U}[\![\text{return } E;]\!]\ \texttt{CTX}\ \mathit{genv}\ \mathit{lenv}\ \mathit{str}\ \mathit{it} = \\
\quad \text{let } (V, \mathit{str}') = \mathbb{E}_{\texttt{ssa}}[\![E]\!]\ \mathit{genv}\ \mathit{lenv}\ \mathit{str} \\
\quad \text{in } (\text{null}, \mathit{lenv}, \mathit{str}', \texttt{CTX}) \\
\mathbb{U}[\![x = E; \overline{U}]\!]\ \texttt{CTX}\ \mathit{genv}\ \mathit{lenv}\ \mathit{str}\ \mathit{it} = \\
\quad \text{let } (V, \mathit{str}') = \mathbb{E}_{\texttt{ssa}}[\![E]\!]\ \mathit{genv}\ \mathit{lenv}\ \mathit{str} \\
\quad \text{in} \mathbb{U}[\![\overline{U}]\!]\ \texttt{CTX}\ \mathit{genv}\ \mathit{lenv} \cup (x, V)\ \mathit{str}'\ \mathit{it}, \texttt{CTX}) \\
\mathbb{U}[\![\text{if } E \text{ goto } \texttt{CTX}'; \overline{U}]\!]\ \texttt{CTX}\ \mathit{genv}\ \mathit{lenv}\ \mathit{str}\ \mathit{it} = \\
\quad \text{case } \mathbb{E}_{\texttt{ssa}}[\![E]\!]\ \mathit{genv}\ \mathit{lenv}\ \mathit{str} \text{ of} \\
\quad \{(\mathit{true}, \mathit{str}') \rightarrow \mathbb{U}[\![\mathit{it}[\texttt{CTX}']]\!]\ \texttt{CTX}\ \mathit{genv}\ \mathit{lenv}\ \mathit{str}'\ \mathit{it} \\
\quad ; (\mathit{false}, \mathit{str}') \rightarrow \mathbb{U}[\![\overline{U}]\!]\ \texttt{CTX}\ \mathit{genv}\ \mathit{lenv}\ \mathit{str}'\ \mathit{it}\} \\
\mathbb{U}[\![\text{goto } \texttt{CTX}'; \overline{U}]\!]\ \texttt{CTX}\ \mathit{genv}\ \mathit{lenv}\ \mathit{str}\ \mathit{it} = \mathbb{U}[\![\mathit{it}[\texttt{CTX}']]\!]\ \texttt{CTX}\ \mathit{genv}\ \mathit{lenv}\ \mathit{str}\ \mathit{it} \\
\mathbb{U}[\![\texttt{CTX}' : \overline{\phi}; \overline{U}]\!]\ \texttt{CTX}\ \mathit{genv}\ \mathit{lenv}\ \mathit{str}\ \mathit{it} = \mathbb{U}[\![\overline{U}]\!]\ \texttt{CTX}'\ \mathit{genv}\ (\mathbb{F}_{\texttt{ssa}}[\![\overline{\phi}]\!]\ \texttt{CTX}\ \mathit{lenv})\ \mathit{str}\ \mathit{it} \\
\\
\texttt{buildIT} :: [\texttt{UnSSA}] \rightarrow \{(\texttt{CONTEXT}, [\texttt{UnSSA}])\} \\
\texttt{buildIT}\ [] = \{\} \\
\texttt{buildIT}\ (U; \overline{U}) = \texttt{buildIT}\ \overline{U} \\
\texttt{buildIT}\ (\texttt{CTX} : U; \overline{U}) = \text{let } \mathit{it} = \texttt{buildIT}\ (U; \overline{U}) \text{ in } \mathit{it} \cup \{(\texttt{CTX}, (U; \overline{U}))\}
\end{array}
$$

Figure 13: Dynamic Semantics of unstructured SSA

– Subcase 2: $\texttt{CTX}_u$ and $\texttt{CTX}_s$ are from the while loop body.

$$
\begin{array}{lcl}
lenv_u^1 & = & \mathbb{F}_{\texttt{ssa}}[\![\texttt{merge}\ \overline{\phi}\ (\texttt{merge}\ \overline{\phi''}\ \overline{\phi'})]\!]\ \texttt{CTX}_u\ lenv_u \\
 & & \text{drop } \overline{\phi} \text{ since both contexts} \\
 & & \text{are from the preceding block.} \\
 & = & \mathbb{F}_{\texttt{ssa}}[\![\texttt{merge}\ \overline{\phi''}\ \overline{\phi'}]\!]\ \texttt{CTX}_u\ lenv_u \\
 & & \text{By Lemma C.1.} \\
 & & \text{Note that } \texttt{CTX}_u \in params(\overline{\phi''}), \texttt{CTX}_s \in params(\overline{\phi'}). \\
 & = & \mathbb{F}_{\texttt{ssa}}[\![\overline{\phi'}]\!]\ \texttt{CTX}_s\ (\mathbb{F}_{\texttt{ssa}}[\![\overline{\phi''}]\!]\ \texttt{CTX}_u\ lenv_u) \\
 & & \text{By I.H.} \\
 & = & \mathbb{F}_{\texttt{ssa}}[\![\overline{\phi'}]\!]\ \texttt{CTX}_s\ lenv_s \\
 & = & lenv_s^1
\end{array}
$$

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 $lenv_u^1 = lenv_s^1$ (4). Continuing from (3),

$$
\begin{array}{l}
\mathbb{U}[\![\text{if } \neg(E) \text{ goto } (\text{nnsucc } G\ \texttt{CTX}); \overline{U};\ \\
\text{goto } \texttt{CTX}; \overline{U'}]\!]\ \texttt{CTX}\ genv\ lenv_u^1\ str\ it \\
\longrightarrow \\
\text{case } \mathbb{E}_{\texttt{ssa}}[\![E]\!]\ genv\ lenv_u^1\ str \text{ of} \quad --(5) \\
\quad (false, str_u^2) \rightarrow \mathbb{U}[\![it[\text{nnsucc } G\ \texttt{CTX}]]\!]\ \texttt{CTX}\ genv\ lenv_u^1\ str_u^2 \\
\quad (true, str_u^2) \rightarrow \mathbb{U}[\![\overline{U}; \text{goto } \texttt{CTX}; \overline{U'}]\!]\ \texttt{CTX}\ genv\ lenv_u^1\ str_u^2
\end{array}
$$

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

- Subcase 1: $\mathbb{E}_{\texttt{ssa}}[\![E]\!]\ genv\ lenv_u^1\ str$ evaluates to $(false, str_u^2)$ and $\mathbb{E}_{\texttt{ssa}}[\![E]\!]\ genv\ lenv_s^1\ str$ evaluates to $(false, str_s^2)$ and $str_u^2 = str_s^2$. Continuing from (2), we have

$$\overline{\mathbb{B}}_{\texttt{ssa}}[\![\overline{B'}]\!]\ \texttt{CTX}[\text{join } \{\blacksquare\} \text{ while } \mathbf{E}\ \{\overline{\mathbf{B}}\}]\ genv\ lenv_s'\ str_s' \quad --(6)$$

  Continuing from (5), we have

$$\mathbb{U}[\![\overline{U'}]\!]\ \texttt{CTX}\ genv\ lenv_u^1\ str_u^2\ it \quad --(7)$$

  where $lenv_s' = lenv_s^1 = lenv_u^1$ and $str_s' = str_s^2 = str_u^2$. Note that CTX is the context containing the while loop. $\texttt{CTX} \overset{G}{\preceq} \texttt{CTX}[\text{join } \{\blacksquare\} \text{ while } \mathbf{E}\ \{\overline{\mathbf{B}}\}]$. From (1), we find that $lenv_s' = \mathbb{F}_{\texttt{ssa}}[\![[]]\!]\ \texttt{CTX}\ lenv_u^1$. We apply I.H to (6) and (7) to conclude this sub case.
- Subcase 2: $\mathbb{E}_{\texttt{ssa}}[\![E]\!]\ genv\ lenv_u^1\ str$ evaluates to $(true, str_u^2)$ and $\mathbb{E}_{\texttt{ssa}}[\![E]\!]\ genv\ lenv_s^1\ str$ evaluates to $(true, str_s^2)$ and $str_u^2 = str_s^2$. Continuing from (2), we have

$$
\begin{array}{l}
\text{case } \overline{\mathbb{B}}_{\texttt{ssa}}[\![\overline{B}]\!]\ \texttt{CTX}[\text{join } \{\blacksquare\} \text{ while } \mathbf{E}\ \{\overline{\mathbf{B}}\}]\ genv\ lenv_s^1\ str_s^2 \text{ of} \\
\quad (v_s^3, lenv_s^2, str_s^3, \texttt{CTX}_s^3) \rightarrow \\
\quad \overline{\mathbb{B}}_{\texttt{ssa}}[\![\texttt{CTX} : \{\text{join}\{\overline{\phi'}\}\} \text{while } E\ \{\overline{B}\}\}; \overline{B'}]\!] \\
\qquad \texttt{CTX}_s^3\ genv\ lenv_s^2\ str_s^3 \qquad --(8)
\end{array}
$$

  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 $\overline{\mathbb{B}}_{\texttt{ssa}}[\![\cdot]\!]$. Continuing from (5), we have

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

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

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

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

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

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

and

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

where

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

Recall that

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

We consider two sub-cases:

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

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

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

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

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

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

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

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

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

□

C.3 Time Complexity of Flattening Algorithm

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

Received 2023-05-26; accepted 2023-06-23