A Formal Study on Backward Compatible Dynamic Software Updates

Jun Shen
Arizona State University
jun.shen.1@asu.edu

Rida A. Bazzi
Arizona State University
bazzi@asu.edu

Abstract

We study the dynamic software update problem for programs interacting with an environment that is not necessarily updated. We argue that such updates should be backward compatible. We propose a general definition of backward compatibility and cases of backward compatible program update. Based on our detailed study of real world program evolution, we propose classes of backward compatible update for interactive programs, which are included at an average of 32% of all studied program changes. The definitions of update classes are parameterized by our novel framework of program equivalence, which generalizes existing results on program equivalence to non-terminating executions. Our study of backward compatible updates is based on a typed extension of *W* language.

Categories and Subject Descriptors D.3.1 [*Formal Definitions and Theory*]: Semantics, Syntax; D.2.4 [*Software/Program Verification*]: Correctness Proof, Formal Methods; F.3.2 [*Semantics of Programming Languages*]: Operational Semantics, Program Analysis; D.3.3 [*Language Constructs and Features*]: Input/output, Procedures, functions, and subroutines

General Terms Theory

Keywords dynamic software update, backward compatibility, program equivalence, proof rule, operational semantic

Contents

1. Introduction

Dynamic software update (DSU) allows programs to be updated in the middle of their execution by mapping a state of an old version of the program to that of a newer version. The ability to update programs without having to restart them is useful for high-availability applications that cannot afford the downtime incurred by offline updates [16]. DSU has been an active area of research[5, 16, 22, 25] with much of the published work emphasizing the *update mechanism* that implements a *state mapping* which maps the execution state of an old version of the program to that of a new version. DSU *safety* has not yet been successfully studied. Existing studies on DSU safety are lacking in one way or another: high-level studies are concerned with change management for system components [9, 19] and lower-level studies typically require significant programmer annotations [15, 24, 32] or have a restricted class of applications to which they apply (e.g., controller systems [28]).

In this paper, we consider the safety of DSU when applied to possibly non-terminating programs interacting with an environment that is not necessarily updated. For such updates, the new version of the program must be able to interact with the old environment, which means that it should be, in some sense, *backward compatible* with the old version. A strict definition of backward compatibility would require the new version to exhibit the same I/O behavior as the old version; in other words the two programs are observationally equivalent. It should be immediately clear that a more nuanced definition is needed because observational-equivalence does not allow changes which one would want to allow as backward compatible such as bug fixes, new functionalities, or usability improvement (e.g., improved user messages). Allowing for such differences would be needed in any practical definition of backward compatibility. One contribution of this work is a general definition of backward compatibility, a classification of common backward compatible program behavior changes, as well as classes of program change from real world program evolution.

Determining backward compatibility, which allows for differences between two program versions, is requiring one to solve the *semantic equivalence* problem which has been extensively studied [6, 13, 17, 18, 20, 21, 23, 31]. Unfortunately, existing results turned out to be lacking in one or more aspects which rules out retrofitting them for our setting. In fact, existing work on program equivalence typically guarantees equivalence at the end of an execution. Such equivalence is not adequate for our purposes because it does not allow us to express that a point in the middle of a loop execution of one program *corresponds* (in a well defined sense) to a point in the middle of a loop execution of another program. The ability to express such correspondences is desirable for dynamic software update. Besides, existing formulations of the program equivalence problem either do not use formal semantics [7, 17, 18], only apply to terminating programs [6, 20], severely restrict the programming model [13, 18, 31], or rely on some form of model checking [21, 23] (which is not appropriate for non-terminating programs with infinite states). Our goal for program equivalence is to establish compile-time conditions ensuring that two programs have the same I/O behavior in *all* executions. In particular, if one program enters an infinite loop and does not produce a certain output, the other program should not produce that output either. This is different from much of the literature on program equivalence which only guarantees same behavior in terminating executions.

The closest work that aims to establish program equivalence for nonterminating programs is that of Godlin and Strichman [13] who give sufficient conditions for semantic equivalence for a language that includes recursive functions, but does not allow loops (loops are extracted as recursive functions). That and the fact that equivalence is enforced on corresponding functions severely limits the applicability of the work to general transformations affecting loops such as loop-invariant code motion, loop fission/fusion. So, as a major component of our formal treatment of backward compatible updates, we set out to develop sufficient conditions for semantic equivalence for programs in a typed extension of the *W* languages [11] with small-step operational semantics. The syntax of language is extended with arrays and enumeration types and the semantics take into consideration the execution environment to allow various classes of updates.

In summary, the paper makes the following contributions:

1. We formally define backward compatibility and identify cases of backward compatible program behavior for typical program update motivation.
2. We identify and formally define classes of program changes that result in backward compatible program update based on empirical study of real world program evolution.
3. We give a formal treatment of the semantic equivalence for nonterminating imperative programs.

The rest of the paper is organized as follows. Section 2 proposes the general backward compatibility and cases of backward compatible new program behavior. Then we describe real world update classes that result in backward compatible update in Section 3. Section 4 formally defines our extension of the **W** language to study backward compatible updates. Section 4.3 shows terms, notations and definitions (e.g., execution) heavily used in the technical result. The technical results on semantic equivalence are presented in Section 5. We propose our formal treatment of real world update classes in Section 6. A more detailed comparison to related work is given in Section 7 . Section 8 concludes the paper.

2. Backward compatibility

2.1 Programs and Specifications

Programs are designed to satisfy specifications. Specification can be explicitly provided or implicitly defined by the behavior of a program. Programs interact with their environment by receiving inputs and producing outputs. In this section we introduce enough of a computing model to describe the input/output behavior of programs; In the next section we introduce a specific programming language to reason about specific software updates.

An execution of a program consists of a sequence of steps from a finite set of steps, $\mathcal{S} = \mathcal{S}_{in} \cup \mathcal{S}_{internal} \cup \mathcal{S}_{out} \cup \{halt\}$. A step of a program can either be an input step in which input is received, an internal step in which the state of the program is modified, an output step in which output is produced, or a halt.

We make a distinction between internal state of a program and external state (e.g., application settings) of the local environment in which the program executes. Such external state can include the state of a file system that program can access; we include both as part of the program state. The state of a program is an element of a set $\mathcal{M} \times \mathcal{I}$, where the set $\mathcal{M} = \mathcal{M}_{int} \times \mathcal{M}_{ext}$, $\mathcal{M}_{int} = \prod_{k=0}^{n_{int}} V_k$ is a cartesian product of n_{int} sets of values, one for each internal memory location, and $\mathcal{M}_{ext} = \prod_{k=0}^{n_{ext}} V_k$ is a cartesian product of n_{ext} sets of values, one for each external location. The input value last received is an element of the set $\mathcal{I}$ of input values.

A program executes in an *execution environment*. An execution environment (M_{ext_0}, I) specifies an initial value for the external program state M_{ext_0} and a possibly infinite sequence of input values I. The input sequence is assumed to be produced by *users* that we do not model explicitly.

A step of a program P is a mapping that specifies the next program state and the next step to execute. For an internal step $s_{internal} \in \mathcal{S}_{internal}$, the mapping is $s_{internal} : \mathcal{M} \times \mathcal{I} \mapsto$

$\mathcal{S} \times \mathcal{M} \times \{\bot\}$, which specifies the next step and how the state is modified. The internal steps clear input in the state if any. For an output step $s_{out} \in \mathcal{S}_{out}$, the mapping $s_{out} : \mathcal{M} \mapsto \mathcal{S} \times \mathcal{O}$ which specifies the next step to execute and the output value produced. $\mathcal{O}$ is the set of output values produced by the program. An input step $s_{in} \in \mathcal{S}_{in}$ is simply an element of $\mathcal{S} \times \mathcal{I}$ and specifies the next step to execute and the input obtained from the environment. (We simply write $s_{in}()$ to denote the next step and the input received.) Because the input value is received by the program, we do not restrict the next step to execute. We allow the input value to be ignored by the program by two consecutive input steps. When the step is $halt$, there is no further action as if $halt$ were mapped to itself.

Definition 1. (Program) *A program P is a tuple $(\mathcal{S}, \mathcal{M}, M_{int_0}, s_0, \mathcal{I}, \mathcal{O})$, where $\mathcal{S}$ is the set of steps as defined above, $\mathcal{M}$ is the set of program states, M_{int_0} is the initial internal state, s_0 is the initial step, and $\mathcal{I}$ and $\mathcal{O}$ are disjoint sets of input and output values.*

We do not include the initial external state M_{ext_0} in the program definition; we include it in the execution environment of P.

Definition 2. (Execution) *An execution of a program $P = (\mathcal{S}, \mathcal{M}, M_{int_0}, s_0, \mathcal{I}, \mathcal{O})$ in execution environment (M_{ext_0}, I), where I is a possibly infinite sequence of input values from $\mathcal{I}$, is a sequence of configurations C from the infinite set $\{(M, s, i, I_r, IO)\}$. A configuration c has the form $c = (M, s, i, I_r, IO)$, where M is a state, s is a step, i is the last input received, I_r is a sequence of remaining input values and IO is the input/output sequence produced so far. The kth configuration c_k in an execution is obtained from the $(k-1)$th configuration $c_{k-1} = (M, s, i, I_r, IO)$ where $s \neq halt$ in one of the following cases:*

1. *The first configuration c_0 is of the form $(M_0, s_0, \bot, I, \varnothing)$, where $M_0 = (M_{int_0}, M_{ext_0})$;*
2. *$s \in \mathcal{S}_{internal} : c_k = (M', s', \bot, I_r, IO)$, where $(s', M', \bot) = s(M, i)$;*
3. *$s \in \mathcal{S}_{in}$ and the remaining inputs I_r is not empty: $c_k = (M, s', head(I_r), tail(I_r), IO{\cdot}head(I_r))$ where $(s', head(I_r)) = s(I_r)$;*
4. *$s \in \mathcal{S}_{in}$ and the remaining inputs I_r is empty: $c_k = c_{k-1}$;*
5. *$s \in \mathcal{S}_{out} : c_k = (M, s', i, I_r, IO \cdot o')$, where $(s', o') = s(M)$;*

In the definition, $head(I)$ denotes the head (leftmost) element in the sequence I and $tail(I)$ denotes the remaining sequence without the head. The input value in i is either consumed by the next internal step or updated by another input from the next input step. Execution is stuck if an input step is attempted in state in which there are no remaining inputs. In what follows, we include the execution environment in the execution and we abuse notation to say (M_{ext_0}, I, C) is an execution of a program P.

Specifications We consider specifications that define the input/output behavior of programs. Specifications are not concerned with how fast an output is produced or about the internal state of the program.

Definition 3. (Specification) *Given a set $\mathcal{M}_{ext}$ of external states, a set $seq(\mathcal{I})$ of input sequences, and a set $seq(\mathcal{I} \cup \mathcal{O})$ of I/O sequences, specification Σ is a predicate: $\mathcal{M}_{ext} \times seq(\mathcal{I}) \times seq(\mathcal{I} \cup \mathcal{O}) \times \mapsto \{true, false\}$.*

We define the I/O sequence of a sequence of configurations C to be a sequence $IO(C)$ of values from $\mathcal{I} \cup \mathcal{O}$ such that every finite prefix of $IO(C)$ is the IO sequence of some configuration $c \in C$ and every I/O sequence of a configuration $c \in C$ is a finite prefix of $IO(C)$.

An execution (M_{ext_0}, I, C) of program P satisfies a specification Σ if $\Sigma(M_{ext_0}, I, IO(C)) = true$. A specification distinguishes executions into those that satisfy the specification and those that do not.

A specification defines the external behavior of a program that is observed by a user. The input sequence and I/O sequence are obviously part of external behavior. We also include $\mathcal{M}_{ext}$ in specification domain because a user can have information about the external state. For example, a user who has data stored in the file system considers the program's refusal to access the stored data a violation of the service specification; this is not the case if the user has no stored data.

2.2 Hybrid executions and state mapping

DSU is a process of updating software while it is running. This results in a hybrid execution in which part of the execution is that of the old program and part of the execution is for the new program.

State mapping is a function δ mapping an internal state and a non-halt step of one program P to an internal state and a step of another program P', $\delta : \mathcal{M}^P_{int} \times (\mathcal{S}^P \setminus \{halt\}) \mapsto \mathcal{M}^{P'}_{int} \times \mathcal{S}^{P'}$. The external state is not mapped because the environment is not necessarily updated. In addition, we cannot change input and output that already occurred and that I/O must be part of the hybrid execution.

Definition 4. (Hybrid execution) *A hybrid execution $(M_{ext_0}, I, C_P; C_{P'})$, produced by DSU using state mapping δ from program P to program P', is an execution (M_{ext_0}, I, C_P) of P concatenated with an execution $(M'_{ext}, I'_r, C_{P'})$ of P' where the first configuration $c_{P'} = ((M'_{int}, M'_{ext}), s', i', I'_r, IO')$ in $C_{P'}$ is obtained by applying the state mapping to the last configuration $c_P = ((M_{int}, M_{ext}), s(\neq halt), i, I_r, IO)$ in C_P as follows:*

- *$(M'_{int}, s') = \delta(M_{int}, s)$;*
- *$(i' = i) \wedge (I'_r = I_r) \wedge (IO' = IO) \wedge (M_{ext} \subseteq M'_{ext})$.*

2.3 Backward compatibility

In this paper, we consider updates in which the environment is not necessarily updated. It follows that in order for the hybrid execution to be meaningful, the new program should provide functionality expected by both old and new users of the system.

In practice, specifications are not explicitly available. Instead, the program is its own specification. This means that the specification that the program satisfies can only be inferred by the external behavior of the program. Bug fixes create a dilemma for dynamic software updates. When a program has a bug, its external behavior does not captures its *implicit specification* and the update will change the behavior of the program. In what follows, we first discuss what flexibility we can be afforded for a backward compatible update and then we give formal definitions of backward compatibility and state our assumptions for allowing bug fixes.

We consider a hybrid execution starting from a program $P = (\mathcal{S}, \mathcal{M}_{int} \times \mathcal{M}_{ext}, M_{int_0}, s_0, \mathcal{I}, \mathcal{O})$ and being updated to a program $P' = (\mathcal{S}', \mathcal{M}'_{int} \times \mathcal{M}'_{ext}, M'_{int_0}, s'_0, \mathcal{I}', \mathcal{O}')$. We examine how the two programs should be related for a meaningful hybrid execution.

1. (Inputs) Input set $\mathcal{I}'$ of P' should be a superset of that $\mathcal{I}$ of P to allow for *old users* to interact with P' after the update. It is possible to allow for new input values in $\mathcal{I}'$ to accommodate new functionality under the assumption that old users do not generate new input values. Such new input values should be expected to produce erroneous output by old users as they are not part of P's specification.
2. (Outputs) Output produced by P' should be identical to output produced by P if all the input in an execution comes from the input set of P. This is needed to ensure that interactions between old users and the program P' can make sense from the

perspective of old users. This is true in the case that the update does not involve a bug fix, but what should be done if the update indeed involves a bug fix and the output produced by the old program was not correct to start with? As far as syntax, a bug fix should not introduce new output values. As far as semantics, we should allow the bug fix to change what output is produced for a given input. We discuss this further under the bug fix heading. In summary, if we ignore bug fixes, the new program should behave as the old program when provided with input meant for the old program.

3. (Bugfix) Handling bug fixes is problematic. If the produced output already violates the fix, then there is no way for the hybrid execution to satisfy the *implicit semantics* of the program or the semantics of the new program. Some bug fixes can be handled. For example, a bug that causes a program to crash for some input can be fixed to allow the program to continue executing. Applying the fix to a program that has not encountered the bug should not be problematic. Another case is when the program should terminate for some input sequence, but the old program does not terminate. A bug fix that allows the program to terminate should not present a semantic difficulty for old users.

   In general, we assume that there are *valid* executions and *invalid* executions of the old program. I/O sequences produced in invalid executions are not in specification of the program. We assume that an invalid execution will lead to an *error* configuration not explicitly handled by the program developers. We do not expect the state mapping to change an error configuration into an non-error configuration just as static updating does not fix occurred errors. Besides, we do not attempt to determine if a particular configuration is an error configuration. Such determination is not possible in general and very hard in practice. We simply assume that the configuration at the time of the update is not an error configuration. (which is equivalent to assuming the existence of an oracle $\mathcal{J}_P$ to determine if a particular configuration is erroneous, $\mathcal{J}_P(C_P)$ = true if the configuration C_P is not erroneous).

4. (New functionality) New functionality is usually accompanied by new inputs/outputs and the expansion of external state. We assume that new functionality is independent of existing functionality in the sense that programs P and P' produce the same I/O sequence when receiving inputs in $\mathcal{I}$ only. We therefore assume all new inputs $\mathcal{I}' \setminus \mathcal{I}$ are introduced by new functionality.

   Every external state of P is part of some external state of program P' because of the definition of the specification of P. We only consider expansion of the external state of P for new functionalities in P' where the expansion of external state is independent of values in existing external state. One of the motivating examples is to add application settings for new program feature.

In light of the discussion above we give the following definition of backward compatibility in the absence of bug fixes.

Definition 5. (Backward compatible hybrid executions) *Let $P = (\mathcal{S}, \mathcal{M}_{int} \times \mathcal{M}_{ext}, M_{int_0}, s_0, \mathcal{I}, \mathcal{O})$ be a program satisfying a specification Σ. We say that a hybrid execution $(M_{ext}, I, C_P; C'_P)$ from P to a program $P' = (\mathcal{S}', \mathcal{M}'_{int} \times \mathcal{M}'_{ext}, M'_{int_0}, s'_0, \mathcal{I}', \mathcal{O}')$ is backward compatible with implicit specification of P if all of the following hold:*

- *The last configuration in C_P is not an error configuration, $C_P =$ "$C'; (M, s', i, I_r, IO)$" : $\mathcal{J}_P(C_P)$=true.*
- *The hybrid execution satisfies the specification Σ of P, $\Sigma(M_{ext}, I, IO(C_P; C'_P)) = true$;*
- *Inputs/outputs/external states of P are a subset of those of P' : $\mathcal{I} \subseteq \mathcal{I}', \mathcal{O} \subseteq \mathcal{O}'$ and $\mathcal{M}_{ext} \subseteq \mathcal{M}'_{ext}$;*

If there is bug fix between programs P' and P, we need to adapt Definition 5 to allow for some executions on input sequences from $\mathcal{I}$ to violate the specification of P. Above we identified two cases in which bug fixes are safe (replacing a response with no response or replacing a no response with a correct response without introducing new output values). We omit the definition.

We have the backward compatible updates by extending the definition of a backward compatible hybrid execution to all possible hybrid executions.

Definition 6. (Backward compatible updates) *We say an updated program P' is backward compatible with a program P in configuration C if there is hybrid execution, from configuration C of P to P' that is backward compatible with specification of P.*

2.4 Backward compatible program behavior changes

With the formal definition of backward compatibility, it is desirable to check what behavior changes of an updated program help ensure a safe update. Backward compatibility is essentially a relation between I/O sequences produced by an old program and those produced by an updated program. We summarized typical possibilities of the relation into six cases in Figure 1 by considering consequence of major update motivation (i.e., new functionality, bug fix and program perfective/preventive needs [1]). According to David Parnas [29], a program is updated to adapt to changing needs. In other words, program changes are to produce more or less or different output according to changing needs. These changes are captured by case 2, 3, 4, 5 and 6 in Figure 1. We also capture output-preserving changes which are most likely motivated by the program developer's own needs (e.g., software maintainability), which is case 1 in Figure 1.

Furthermore, we find that an update is *backward compatible* if in every execution the new program behavior is one of the six cases in Figure 1. Cases 1 and 2 are obviously backward compatible because an old client is guaranteed to get old responses. Cases 3, 4, 5, and 6 are not obviously backward compatible. Unlike case 1 and 2, case 3, 4 and 5 are backward compatible under specific assumptions on program semantics while case 6 is different. Case 3 is backward compatible because we assume the change is either adding new functionality, or fixing a bug in which the old program hanged or crashed. Similarly, case 4 is backward compatible. Case 5 is backward compatible because different I/O interaction could express the same application semantics. For example, a greeting message could be changed from "hi" to "hello". Case 6 is backward compatible in that the new program makes implicit specification of the program explicit by enforcing restrictions on program state and therefore eliminating undesired I/O sequence.

The six cases in Fig. 1 have covered the changes of output, including more or less or different output. There exists more specific cases of backward compatible program behavior changes under various specific assumptions. However, these more specific cases could be attributed to one of the six cases as far as the changes of output are concerned. In conclusion, it is not possible to go much beyond the six cases of backward compatibility in Fig. 1.

3. Real world backward compatible update classes: brief description

We have studied evolution of three real world programs (i.e., vsftpd, sshd and icecast) to identify real world changes that are backward compatible. We chose these three programs because the programs are widely used in practice [2, 3] and are widely studied in the DSU community [26, 27]. We have studied several years of re-

Case	Formal new program behavior
1	the old behavior including external state extension: $\Sigma_P \subseteq \Sigma_{P'}$, or $\Sigma_{P'} = \{(M_{ext'}, \text{oneseq}(\mathcal{I}), \text{oneseq}(\mathcal{I} \cup \mathcal{O})) \rightarrow \text{val}$ $\mid \exists (M_{ext}, \text{oneseq}(\mathcal{I}), \text{oneseq}(\mathcal{I} \cup \mathcal{O})) \rightarrow \text{val in } \Sigma_P$ and $M_{ext} \subseteq M_{ext'}$ $\}$ where $\mathcal{I} = \mathcal{I}'$, $\mathcal{O} = \mathcal{O}'$ and $\mathcal{M}_{ext} \subseteq \mathcal{M}_{ext'}$
2	the old behavior for old input and consuming inputs that are only from new clients: $\Sigma_P \subseteq \Sigma_{P'} \wedge$ $\Sigma_{P'} \setminus \Sigma_P = \{(M_{ext}, \text{oneseq}(\mathcal{I}'), \text{oneseq}(\mathcal{I}' \cup \mathcal{O}')) \rightarrow \text{true}$ $\mid \text{oneseq}(\mathcal{I}' \cup \mathcal{O}')$ includes at least one input in $(\mathcal{I}' \setminus \mathcal{I})\} \neq \emptyset$ where $\mathcal{I} \subset \mathcal{I}'$, $\mathcal{O} \subseteq \mathcal{O}'$ and $\mathcal{M}_{ext} = \mathcal{M}_{ext'}$
3	producing more output while the old program terminates: $\Sigma_{P'} \setminus \Sigma_P = \{(M_{ext}, \text{oneseq}(\mathcal{I}), \text{oneseq}(\mathcal{I} \cup \mathcal{O})) \mapsto \text{false}$ $\mid (M_{ext}, \text{oneseq}(\mathcal{I}), \text{oneseq}(\mathcal{I} \cup \mathcal{O})) \in \Delta_f \neq \emptyset\}$ $\cup \{(M_{ext}, \text{oneseq}(\mathcal{I}), \text{oneseq}(\mathcal{I} \cup \mathcal{O}) \,\mathbf{.}\, \text{oneseq'}(\mathcal{I} \cup \mathcal{O})) \mapsto \text{true}$ $\mid (M_{ext}, \text{oneseq}(\mathcal{I}), \text{oneseq}(\mathcal{I} \cup \mathcal{O}) \,\mathbf{.}\, \text{oneseq'}(\mathcal{I} \cup \mathcal{O})) \in \Delta_t \neq \emptyset\}$ $\Sigma_P \setminus \Sigma_{P'} = \{(M_{ext}, \text{oneseq}(\mathcal{I}), \text{oneseq}(\mathcal{I} \cup \mathcal{O})) \mapsto \text{false}$ $\mid (M_{ext}, \text{oneseq}(\mathcal{I}), \text{oneseq}(\mathcal{I} \cup \mathcal{O})) \in \Delta_t \neq \emptyset\}$ $\cup \{(M_{ext}, \text{oneseq}(\mathcal{I}), \text{oneseq}(\mathcal{I} \cup \mathcal{O}) \,\mathbf{.}\, \text{oneseq'}(\mathcal{I} \cup \mathcal{O})) \mapsto \text{true}$ $\mid (M_{ext}, \text{oneseq}(\mathcal{I}), \text{oneseq}(\mathcal{I} \cup \mathcal{O}) \,\mathbf{.}\, \text{oneseq'}(\mathcal{I} \cup \mathcal{O})) \in \Delta_f \neq \emptyset\}$ where $\mathcal{I} = \mathcal{I}'$, $\mathcal{O} = \mathcal{O}'$ and $\mathcal{M}_{ext} = \mathcal{M}_{ext'}$
4	termination while the old program produces erroneous output: $\Sigma_{P'} \setminus \Sigma_P = \{(M_{ext}, \text{oneseq}(\mathcal{I}), \text{oneseq}(\mathcal{I} \cup \mathcal{O})) \mapsto \text{true}$ $\mid (M_{ext}, \text{oneseq}(\mathcal{I}), \text{oneseq}(\mathcal{I} \cup \mathcal{O})) \in \Delta_t \neq \emptyset\}$ $\cup \{(M_{ext}, \text{oneseq}(\mathcal{I}), \text{oneseq}(\mathcal{I} \cup \mathcal{O}) \,\mathbf{.}\, \text{oneseq'}(\mathcal{I} \cup \mathcal{O})) \mapsto \text{false}$ $\mid (M_{ext}, \text{oneseq}(\mathcal{I}), \text{oneseq}(\mathcal{I} \cup \mathcal{O}) \,\mathbf{.}\, \text{oneseq'}(\mathcal{I} \cup \mathcal{O})) \in \Delta_f \neq \emptyset\}$ $\Sigma_P \setminus \Sigma_{P'} = \{(M_{ext}, \text{oneseq}(\mathcal{I}), \text{oneseq}(\mathcal{I} \cup \mathcal{O})) \mapsto \text{true}$ $\mid (M_{ext}, \text{oneseq}(\mathcal{I}), \text{oneseq}(\mathcal{I} \cup \mathcal{O})) \in \Delta_f \neq \emptyset\}$ $\cup \{(M_{ext}, \text{oneseq}(\mathcal{I}), \text{oneseq}(\mathcal{I} \cup \mathcal{O}) \,\mathbf{.}\, \text{oneseq'}(\mathcal{I} \cup \mathcal{O})) \mapsto \text{false}$ $\mid (M_{ext}, \text{oneseq}(\mathcal{I}), \text{oneseq}(\mathcal{I} \cup \mathcal{O}) \,\mathbf{.}\, \text{oneseq'}(\mathcal{I} \cup \mathcal{O})) \in \Delta_t \neq \emptyset\}$ where $\mathcal{I} = \mathcal{I}'$, $\mathcal{O} = \mathcal{O}'$ and $\mathcal{M}_{ext} = \mathcal{M}_{ext'}$
5	different output that is functionally equivalent to old output: $(\Sigma_P \neq \Sigma_{P'}) \wedge (\Sigma_P \equiv \Sigma_{P'})$ where $\mathcal{I} = \mathcal{I}'$, $(\mathcal{O} \neq \mathcal{O}') \wedge (\mathcal{O} \equiv \mathcal{O}')$ and $\mathcal{M}_{ext} = \mathcal{M}_{ext'}$
6	enforcing restrictions on program state: $\Sigma_{P'} \setminus \Sigma_P = \{(M_{ext}, \text{oneseq}(\mathcal{I}), \text{oneseq}(\mathcal{I} \cup \mathcal{O})) \mapsto \text{false}$ $\mid (M_{ext}, \text{oneseq}(\mathcal{I}), \text{oneseq}(\mathcal{I} \cup \mathcal{O})) \in \Delta_{arbi} \neq \emptyset\}$ $\Sigma_P \setminus \Sigma_{P'} = \{(M_{ext}, \text{oneseq}(\mathcal{I}), \text{oneseq}(\mathcal{I} \cup \mathcal{O})) \mapsto \text{true}$ $\mid (M_{ext}, \text{oneseq}(\mathcal{I}), \text{oneseq}(\mathcal{I} \cup \mathcal{O})) \in \Delta_{arbi} \neq \emptyset\}$ where $\mathcal{I} = \mathcal{I}'$, $\mathcal{O} = \mathcal{O}'$ and $\mathcal{M}_{ext} = \mathcal{M}_{ext'}$

Figure 1: Six cases of formalized general new program behavior

Software version	Update date	Total	Class
vsftpd 1.1.0 –1.1.1	2002-10-07	16	8
vsftpd 1.1.1 –1.1.2	2002-10-16	8	1
vsftpd 1.1.2 –1.1.3	2002-11-09	8	4
vsftpd 1.1.3 –1.2.0	2003-05-29	61	9
vsftpd 1.2.0 –1.2.1	2003-11-13	33	11
vsftpd 1.2.1 –1.2.2	2004-04-26	10	6
vsftpd 1.2.2 –2.0.0	2004-07-01	52	13
vsftpd 2.0.0 –2.0.1	2004-07-02	7	4
vsftpd 2.0.1 –2.0.2	2005-03-03	23	4
vsftpd 2.0.2 –2.0.3	2005-03-19	18	8
vsftpd 2.0.3 –2.0.4	2006-01-09	14	9
vsftpd 2.0.4 –2.0.5	2006-07-03	21	15
vsftpd 2.0.5 –2.0.6	2008-02-13	20	9
vsftpd 2.0.6 –2.0.7	2008-07-30	16	8
vsftpd 2.0.7 –2.1.0	2009-02-19	53	11
vsftpd 2.1.0 –2.1.2	2009-05-29	21	9
vsftpd 2.1.2 –2.2.0	2009-08-13	34	14
Software version	Update date	Total	Class
vsftpd 2.2.0 –2.2.2	2009-10-19	21	5
vsftpd 2.2.2 –2.3.0	2010-08-06	13	3
vsftpd 2.3.0 –2.3.2	2010-08-19	5	0
vsftpd 2.3.2 –2.3.4	2011-03-12	7	0
vsftpd 2.3.4 –2.3.5	2011-12-19	14	6
vsftpd 2.3.5 –3.0.0	2012-04-10	23	4
vsftpd 3.0.0 –3.0.2	2012-09-19	40	2
sshd 3.5p1 –3.6p1	2003-03-31	95	34
sshd 3.6p1 –3.6.1p1	2003-04-01	13	12
sshd 3.6.1p1 –3.6.1p2	2003-04-29	16	12
sshd 4.5p1 –4.6p1	2007-03-07	48	13
sshd 6.6p1 –6.7p1	2014-10-06	283	51
icecast 0.8.0 –0.8.1	2004-08-04	4	3
icecast 0.8.1 –0.8.2	2004-08-04	2	0
icecast 2.3.0 –2.3.1	2005-11-30	47	10
icecast 2.3.1 –2.3.2	2008-06-02	250	28
icecast 2.4.0 –2.4.1	2014-11-19	178	154

Figure 2: Statistics of classified real world software update

leases of vsftpd and consecutive updates of sshd and icecast. This is because vsftpd is more widely studied by the DSU community [25–27].

Our study of real world program evolution is carried out as follows. We examined every changed function manually to classify updates. For every individual change, we first identified the motivation of the change, then the assumptions under which the change could be considered backward compatible. If the assumption under which the change is considered backward compatible is reasonable, we recorded the change into one particular update class. Finally we summarized common update classes observed in the evolution of studied programs.

Fig. 2 shows the statistics from our study of real world program evolution where "total" refers to the number of all updated functions, "class" refers to the number of updated functions with at least one classified update pattern. In summary, 32% of all updated functions include at least one classified program update; the unclassified updates are mostly bug fix that are related to specific program logic. We summarized seven most common real world update classes from all the studied updates in Fig. 3 and we believe that these update classes are also widespread in other program evolution. Each of the six real world update classes falls in one of the five cases of backward compatibility in Fig. 1. We present informal descriptions of all update classes including required assumptions for the two programs to produce same or equivalent output sequence which guarantees backward compatible DSU.

3.1 Observational equivalence: the old behavior

In case 1 in Fig. 1, two programs are backward compatible because the new program keeps all old behaviors ("observational equivalence"). In our study, we differentiate two types of "observational equivalence" based on if assumptions are required.

Program equivalence We consider several types of program changes that are allowed by "observational equivalence" without user assumptions. These changes include: loop fission or fusion, statement reordering or duplication, and extra statements unrelated to output(e.g., logging related changes). We incorporate these changes in our framework of program equivalence which ensures two programs produce the same output regardless of whether the programs terminate or not. The details of the formal treatment is in Section 5.

Specializing new configuration variables Another update class of "observational equivalence" is "specializing new configuration variables", which is backward compatible under user assumptions.

Update class (Case)	Required assumptions for backward compatible update
program equivalence (1)	none
new config. variables (1)	no redefinitions of new config variables after initialization
enum type extension (2)	no inputs from old clients match the extended enum labels
var. type weakening (3)	no intentional use of value type mismatch and array out of bound
exit on error (4)	correct error check before exit
improved prompt msgs (5)	changing prompt messages for more effective communication
missing var. init. (6)	no intentional use of undefined variables

Figure 3: Required assumptions for real world backward compatible update classes

```
1:                        1':  If $(b)$ then
2:                        2':      output $a * 2$
3:                        3':  else
4:  output $a + 2$        4':      output $a + 2$
    old                        new
```

Figure 4: Specializing new configuration variables

```
1:  enum $id\ \{o_1\}$            1':  enum $id\ \{o_1, o_2\}$
2:  a : enum $id$                 2':  a : enum $id$
3:  If $(a == o_1)$ then          3':  If $(a == o_1)$ then
4:      output $2 + c$            4':      output $2 + c$
5:                                5':  If $(a == o_2)$ then
6:                                6':      output $3 + c$
    old                                new
```

Figure 5: Enumeration type extension

In this update class, new configuration variables are introduced to generalize functionality. For example, in Fig. 15, a new configuration variable b is used to introduce new code. The two statement sequences in Fig. 15 are equivalent when the new variable b is specialized to 0. In general, if all new code is introduced in a way that is similar to that in Fig. 15 where there is a valuation of new configuration variables under which new code is not executed, and new configuration variables are not redefined after initialization, then the new program and the old program produce the same output sequence. The point is that new functionality is not introduced abruptly in interaction with an old client. Instead new functionality could be enabled for a new client when old clients are not a concern.

3.2 Enum. type extension: old behavior for old input and allowing new input

Enumeration types allow developers to list similar items. New code is usually accompanied with the introduction of new enumeration labels. Fig. 16 shows an example of the update. The new enum label o_2 gives a new option for matching the value of the variable a, which introduces the new code "**output** $3 + c$". To show enumeration type extensions to be backward compatible, we assume that values of enum variables, used in the If-predicate introducing the new code, are only from inputs that cannot be translated to new enum labels. This is case 2 of the backward compatibility.

```
1:                       1':  If $(1/(a - 5))$ then
2:                       2':      skip
3:  output $a$           3':  output $a$

    old                       new
```

Figure 6: Exit-on-error

3.3 Variable type weakening: more output when the old program terminates

In program updates, variable types are changed either to allow for larger ranges (weakening) or smaller ranges to save space (strengthening). For example, an integer variable might be changed to become a long variable to avoid integer overflow or a long variable might be changed to an integer variable because the larger range of long is not needed. Type weakening also includes adding a new enumeration value and increasing array size. The kinds of strengthening or weakening that should be allowed are application dependent and would need to be defined by the user in general. The type weakening considered is either changes from type int to long or increase of array size. These updates fix integer overflow or array index out of bound respectively, the case 3 of backward compatibility. Implicitly, we assume that there is no intentional use of integer overflow and array out of bound as program semantics.

3.4 Exit on errors: stopping execution while the old program produces more output

One kind of bug fix, which we call *exit on error*, causes a program to exit in observation of errors that depend on application semantic. Fig. 17 shows an example of exit-on-error update. In the example, the fixed bugs refer to the program semantic error that $a = 5$. Instead of using an "exit" statement, we rely on the crash from expression evaluations to model the "exit". When errors do not occur, the two programs in Fig. 17 produce the same output sequence. This is case 4 of backward compatibility. Naturally, we assume that all error checks are correct.

3.5 Improved prompt messages: functionally equivalent outputs

In practice, outputs could be classified into prompt outputs and actual outputs. Prompt outputs are those asking clients for inputs, which are constants hardcoded in output statements. Actual outputs are dynamic messages produced by evaluation of non-constant expressions in execution. If the differences between two programs are only the prompt messages that a client receives, we consider that the two programs are equivalent. The prompt messages are the replaceable part of program semantics. We observe cases of improving prompt messages in program evolution for effective communication. The changes of prompt outputs do not matter only for human clients. This is case 5 of backward compatibility.

3.6 Missing variable initialization: enforcing restrictions on program states

Another kind of bug fix, which we call *missing variable initialization*, includes initializations for variables whose arbitrary initial values can affect the output sequence in the old program. Fig. 18 shows an example of missing variable initialization. The initialization $b := 2$ ensures the value used in "**output** $b + c$" not to be undefined. Despite of initialization statements, the two programs are same. In general, initializations of variables only affect rare buggy executions of the old program, where undefined variables affect the output sequence. This update class is case 6 of backward compatibility and we assume that there is no intentional use of undefined variable in the program. When there are no uses of variables with

```
1:                          1':  $b := 2$
2:  If $(a > 0)$ then       2':  If $(a > 0)$ then
3:      $b := c + 1$        3':      $b := c + 1$
4:  output $b + c$          4':  output $b + c$
    old                          new
```

Figure 7: Missing initialization

Identifier id	Constant n		Label l
Enum Items	el	::=	$l \mid el_1, el_2$
Enumeration	EN	::=	$\varnothing \mid \text{enum}\, id\, \{el\} \mid EN_1, EN_2$
Prompt Msg	msg	::=	$l : n \mid msg_1, msg_2$
Prompts	$Pmpt$	::=	$\varnothing \mid \{msg\}$
Base type	τ	::=	$\text{Int} \mid \text{Long} \mid \text{pmpt} \mid \text{enum}\, id$
Variables	V	::=	$\varnothing \mid \tau\, id \mid \tau\, id[n] \mid V_1, V_2$
Left value	$lval$	::=	$id \mid id_1[id_2] \mid id[n]$
Expression	e	::=	$id == l \mid lval \mid \text{other}$
Statement	s	::=	$lval := e \mid \text{input}\, id \mid \text{output}\, e \mid \text{skip}$ $\mid \text{while}\, (e)\, \{S\}$
Stmt Seq.	S	::=	$s_1; ...; s_k$ for $k \geq 1$
Program	P	::=	$Pmpt; EN; V; S_{entry}$

Figure 8: Abstract syntax

Values	v	$\in$	$\mathbb{Z}_L \cup \mathbb{L}$	integer values in type long and enum/prompt labels
I/O values	v_{io}	$\in$	$\mathbb{Z}_L$	
Inputs	v_i	::=	$\underline{v}_{io}$	tagged input values
Eval. values	v_{err}	::=	$v \mid \text{error}$	values and the runtime error
Param. types	$\tau_\top$	::=	$\tau \mid \text{array}(\tau, n)$	
Loop Labels	$loop_{lbl}$	$\in$	$\mathbb{N}$	

Figure 9: Values, types and domains

Crash flag	$\mathfrak{f}$	::=	$0 \mid 1$	
Overflow flag	$\mathfrak{of}$	::=	$0 \mid 1$	
Type Env.	Γ	::=	$\varnothing \mid id : \tau_\top \mid id : \{l_1, ..., l_k\} \mid \Gamma_1, \Gamma_2$	
Loop counter	$loop_c$	::=	$(loop_{lbl} \mapsto (n \mid \perp))$	
Value store	σ	::=	$id \mapsto (v \mid \perp)$	values of scalar variables
		$\mid$	$id \mapsto (n \mapsto (v \mid \perp))$	values of array elements
		$\mid$	$id_I \mapsto v_{io}^*$	input sequence
		$\mid$	$id_{IO} \mapsto (v_i \mid v_o)^*$	I/O sequence
State	m	::=	$(\mathfrak{f}, \mathfrak{of}, \Gamma, loop_c, \sigma)$	

Figure 10: Elements of an execution state

undefined variables in executions of the old program, the two programs produce the same output sequence.

4. Formal programming language

We present the formal programming language based on which we prove our semantic equivalence results and describe categories of backward compatible changes. We first explain the language syntax, then the language semantics.

4.1 Syntax of the formal language

The language syntax is in Figure. 8. We use id to range over the set of identifiers, n to range over integers, l to range over labels. We assume unique identifiers across all syntactic categories, unique labels across all enumeration types and the prompt type. We have base type Int and Long for integer values. The integers defined in type Int are also defined in type Long. Every label defined in the prompt type is related with an integer constant as the actual value used in output statement. We differentiate type Long and Int to define the bug fix of type relaxation from Int to Long to prevent overflow in calculation (e.g., a + b can cause an error with Int but not with Long). The type Int is necessary reflecting the concern of space and time efficiency in practical computation. We also have user-defined enumeration type, prompt type and array type.

We explicitly have "$id == l$" and $lval$ as expressions for convenience of the definition of specific updates. To make our programming language general and to separate the concern of expression evaluation, we parameterize the language by "other" expressions which are unspecified.

We have explicit input and output statement because we model the program behavior as the I/O sequence which is the observational behavior of a program. The I/O statement makes it convenient for the argument of program behavior correspondence. In this paper, every I/O value is an integer value which is a common I/O representation [14]. A Statement sequence is defined as $s_1; ...; s_k$ where $k > 0$ for the convenience of syntax-direct definition from both ends of the sequence.

A program is composed of a possibly empty prompt type $Pmpt$, a possibly empty sequence of enumeration types EN, a possibly empty sequence of global variables V and a sequence of entry statements S_{entry}. Finally, we have a standard type system based on our syntax.

4.2 Small-step operational semantics of the formal language

Figure 9 shows semantic categories of our language. We consider values to be either labels $\mathbb{L}$ or integer numbers $\mathbb{Z}_L$ defined in type Long. The integer numbers defined $\mathbb{Z}_I$ of type Int are a proper subset of those in type Long, $\mathbb{Z}_I \subset \mathbb{Z}_L$. We use the notation $\mathbb{Z}_{L+}$ for the positive integers defined in type Long. We use the notation udf$[\![\tau]\!]$ for an undefined value of type τ. Unlike the "undef" in Clight [8], we need to parameterize the undefined value with a type τ because we do not have an underlining memory model that can interpret any block content according to a type. An individual value in I/O sequence is an integer number with tag differentiating inputs and outputs, our tags for inputs and outputs are standard notations [14]. The value from expression evaluation is a pair. One of the pair is either a value v or "error" for runtime errors(e.g., division by zero); the other is the overflow flag (i.e., 0 for no overflow).

We use notation $\tau_\top$ for all types that are defined in syntax, including array types.

Every loop statement in a program is with a unique label $loop_{lbl}$ of a natural number in order to differentiate their executions.

The composition of an execution state is in Figure 10.

1. The crash flag $\mathfrak{f}$ is initially zero and is set to one whenever an exception occurs. Once the crash flag is set, it is not cleared. We only consider unrecoverable crashes. The crash flag is used to make sure that updates do not occur in error states.

2. The overflow flag $\mathfrak{of}$ is initially zero and is set to one whenever an integer overflow in expression evaluation occurs. Overflow flag is sticky in the sense that once it is set, the flag is not cleared. According to [12], integer overflows are common in mature programs.

3. Γ is the type environment mapping enumeration type identifers and variable identifiers to their types. Type environment is necessary for checking array index out of bound or checking value mismatch in execution of input/assignment statement.

4. Loop counters $loop_c$ are to record the number of iterations for one instance of a loop statement. The loop counters $loop_c$ is

$\boxed{(S,m) \to (S',m')}$ $\qquad \dfrac{(r,m) \to (r',m')}{(\mathbb{E}[r],m) \to (\mathbb{E}[r'],m')}$

Eval. Context $\mathbb{E} ::= \_ \mid id[\mathbb{E}] \mid \mathbb{E} == l$
$\mid id := \mathbb{E} \mid id[\mathbb{E}] := e \mid id[v] := \mathbb{E} \mid \text{output } \mathbb{E}$
$\mid \text{while } (\mathbb{E})\{S\} \mid \text{If } (\mathbb{E}) \text{ then } \{S_t\} \text{ else } \{S_f\} \mid \mathbb{E}; S$

Figure 11: Contextual semantic rule

$\boxed{(r,m) \to (r',m')}$

$\mathcal{E} : \text{other} \to \sigma \to (v_{\text{err}} \times \{0,1\})$
$\text{Err} : \text{other} \to \{id\} \quad \text{(unspecified)}$

$$\text{Var}\ \dfrac{\mathfrak{f}=0 \quad \sigma(id)=v}{(id, m(\mathfrak{f},\sigma)) \to (v,m)}$$

$$\text{Arr-1}\ \dfrac{\mathfrak{f}=0 \quad \sigma(id,v_1)=v_2}{(id[v_1], m(\mathfrak{f},\sigma)) \to (v_2,m)}$$

$$\text{Arr-2}\ \dfrac{\mathfrak{f}=0 \quad (\Gamma \vdash id : \text{array}(\tau,n)) \wedge \neg(1 \le v_1 \le n)}{(id[v_1], m(\mathfrak{f},\Gamma)) \to (id[v_1], m(1/\mathfrak{f}))}$$

$$\text{Eq-T}\ \dfrac{\mathfrak{f}=0}{(l == l, m(\mathfrak{f})) \to (1,m)}$$

$$\text{Eq-F}\ \dfrac{\mathfrak{f}=0 \quad l_1 \ne l_2}{(l_1 == l_2, m(\mathfrak{f})) \to (0,m)}$$

$$\text{EEval}\ \dfrac{\mathfrak{f}=0 \quad e = \text{other}}{(e, m(\mathfrak{f},\sigma)) \to (\mathcal{E}[\![e]\!]\sigma, m)}$$

$$\text{ECrash}\ \dfrac{\mathfrak{f}=0}{((\text{error}, v_{\mathfrak{of}}), m(\mathfrak{f})) \to (0, m(1/\mathfrak{f}))}$$

$$\text{EOflow-1}\ \dfrac{\mathfrak{f}=0 \quad \mathfrak{of}=0}{((v, v_{\mathfrak{of}}), m(\mathfrak{f},\mathfrak{of})) \to (v, m(v_{\mathfrak{of}}/\mathfrak{of}))}$$

$$\text{EOflow-2}\ \dfrac{\mathfrak{f}=0 \quad \mathfrak{of}=1}{((v, v_{\mathfrak{of}}), m(\mathfrak{f},\mathfrak{of})) \to (v,m)}$$

Figure 12: SOS rules for expressions

not necessary for program executions but are needed for our reasoning of the execution of loops. When a counter entry for loop label n is not defined in loop counters $loop_c$, we write $loop_c(n) = \bot$. Otherwise, we write $loop_c(n) \ne \bot$.

5. The value store σ is a valuation for scalar variables, array elements, the input sequence variable, and the I/O sequence variable.

Execution state m is a composition of elements discussed above. In our SOS rules, we only show components of a state m when necessary (e.g., $m(\Gamma, \sigma)$).

Figure 11 shows typical contextual rule and Figure 12, 13 and 14 show all SOS rules.

Figure 12 shows rules for expression evaluation. We use the expression meaning function $\mathcal{E} : \text{other} \to \sigma \to (v_{\text{err}} \times \{0,1\})$ to evaluate "other" expressions. In evaluation of expression "other" against a value store σ, the expression meaning function $\mathcal{E}$ returns a pair $(v_{\text{err}}, \mathfrak{of})$ where the value v_{err} is either a value v or an "error", $\mathfrak{of}$ is a flag indicating if there is integer overflow in the evaluation (e.g., 1 if there is overflow). The meaning function $\mathcal{E}$ interprets "other" expressions deterministically. In addition, there is a function Use : other $\to \{id\}$ maps an "other" expression to a set of variables used in the expression; there is a function Err : other $\to \{id\}$ maps an expression to a set of variables whose values decide if the evaluation of expression leads to crash. We assume function Use and Err available. The value returned by the expression meaning function only depends on the values of variables in the use set of the expression and the error evaluation only depends on the variables in the error set.

$\boxed{(r,m) \to (r',m')}$

$$\text{As-Scl}\ \dfrac{\mathfrak{f}=0 \quad \sigma(id) \ne \bot}{(id := v, m(\mathfrak{f},\sigma)) \to (\text{skip}, m(\sigma[v/id]))}$$

$$\text{As-Arr}\ \dfrac{\mathfrak{f}=0 \quad \sigma(id,v_1) \ne \bot}{(id[v_1] := v_2, m(\mathfrak{f},\sigma)) \to (\text{skip}, m(\sigma[v_2/(id,v_1)]))}$$

$$\text{As-Err1}\ \dfrac{\mathfrak{f}=0 \quad (\Gamma \vdash id : \text{array}(\tau,n)) \wedge \neg(1 \le v_1 \le n)}{(id[v_1] := v_2, m(\mathfrak{f},\Gamma)) \to (id[v_1] := v_2, m(1/\mathfrak{f}))}$$

$$\text{As-Err2}\ \dfrac{\begin{array}{c}\mathfrak{f}=0 \qquad \sigma(id) \ne \bot \\ (\Gamma \vdash id : \text{Int}) \wedge (v \in (\mathbb{Z}_L \setminus \mathbb{Z}_I))\end{array}}{(id := v, m(\mathfrak{f},\Gamma,\sigma)) \to (id := v, m(1/\mathfrak{f}))}$$

$$\text{As-Err3}\ \dfrac{\begin{array}{c}\mathfrak{f}=0 \qquad \sigma(id,v_1) \ne \bot \\ (\Gamma \vdash id : \text{array}(\text{Int},n)) \wedge (v_2 \in (\mathbb{Z}_L \setminus \mathbb{Z}_I))\end{array}}{(id[v_1] := v_2, m(\mathfrak{f},\Gamma,\sigma)) \to (id[v_1] := v_2, m(1/\mathfrak{f}))}$$

$$\text{If-T}\ \dfrac{\mathfrak{f}=0 \quad (v \in \mathbb{Z}_L) \wedge (v \ne 0)}{(\text{If } (v) \text{ then } \{S_t\} \text{ else } \{S_f\}, m(\mathfrak{f})) \to (S_t, m)}$$

$$\text{If-F}\ \dfrac{\mathfrak{f}=0}{(\text{If } (0) \text{ then } \{S_t\} \text{ else } \{S_f\}, m(\mathfrak{f})) \to (S_f, m)}$$

$$\text{Wh-T}\ \dfrac{\mathfrak{f}=0 \quad (v \in \mathbb{Z}_L) \wedge (v \ne 0) \quad loop_c(n) = k}{\begin{array}{l}(\text{while}_{\langle n \rangle}\ (v)\ \{S\}, m(\mathfrak{f}, loop_c)) \to \\ (S; \text{while}_{\langle n \rangle}\ (e)\ \{S\}, m(loop_c[(k+1)/n])\end{array}}$$

$$\text{Wh-F}\ \dfrac{\mathfrak{f}=0 \quad loop_c(n) \ne \bot}{(\text{while}_{\langle n \rangle}\ (0)\ \{S\}, m(\mathfrak{f}, loop_c)) \to (\text{skip}, m(loop_c[0/n]))}$$

$$\text{Seq}\ \dfrac{\mathfrak{f}=0}{(\text{skip}; S, m(\mathfrak{f})) \to (S,m)} \qquad \text{Crash}\ \dfrac{\mathfrak{f}=1}{(s, m(\mathfrak{f})) \to (s,m)}$$

Figure 13: SOS rules for Assignment, If, and While statements

As to integer overflow, there are two ways of handling overflow in practice one is to wrap around overflow using twos-complement representation (e.g., the gcc option -fwrapv); the other is to generates traps for overflow (e.g., the gcc option -ftrapv). We adopt a combination of the two handling of overflow: the meaning function $\mathcal{E}$ wraps the overflow in some representation (e.g., two-complement) and notifies the overflow in return value. Rule EOflow-1 and EOflow-2 update the sticky overflow flag. The evaluation of $lval$ or $id == l$ is shown by respective rules in Figure 12.

Figure 13 shows SOS rules for assignment, If, while statements, statement sequence, and crash, which are almost standard. There are four particular crash in execution of assignment statements. One is array out of bound for array access for l-value (e.g., rule As-Err1); the second is assigning a value defined in type Long but not type Int to an Int-typed variable (e.g., rule As-Err2); the third is value mismatch in input statement; the last is expression evaluation exception. As to loop statement, if the predicate expression evaluates to a nonzero integer, corresponding loop counter value increments by one; otherwise, the loop counter value is reset to zero. We

$$\boxed{(r, m) \to (r', m')}$$

$$\text{In-1}\ \frac{\mathfrak{f} = 0 \quad \sigma(id) \neq \bot \quad \text{hd}(\sigma(id_I)) = v_{io} \quad \Gamma \vdash id : \text{Long}}{\begin{array}{l}(\text{input}\ id, m(\mathfrak{f}, \Gamma, \sigma)) \to \\ (\text{skip}, m(\sigma[v_{io}/id][\text{tl}(\sigma(id_I))/id_I][\text{“}\sigma(id_{IO}) \cdot \underline{v}_{io}\text{”}/id_{IO}])\end{array}}$$

$$\text{In-2}\ \frac{\begin{array}{cc}\mathfrak{f} = 0 & \sigma(id) \neq \bot \\ \text{hd}(\sigma(id_I)) = v_{io} & (\Gamma \vdash id : \text{Int}) \wedge (v_{io} \in \mathbb{Z}_I)\end{array}}{\begin{array}{l}(\text{input}\ id, m(\mathfrak{f}, \Gamma, \sigma)) \to (\text{skip}, \\ m(\sigma[v_{io}/id][\text{tl}(\sigma(id_I))/id_I][\text{“}\sigma(id_{IO}) \cdot \underline{v}_{io}\text{”}/id_{IO}])\end{array}}$$

$$\text{In-3}\ \frac{\begin{array}{cc}\mathfrak{f} = 0 & \sigma(id) \neq \bot \\ \text{hd}(\sigma(id_I)) = v_{io} & (\Gamma \vdash id : \text{Int}) \wedge (v_{io} \notin \mathbb{Z}_I)\end{array}}{(\text{input}\ id, m(\mathfrak{f}, \Gamma, \sigma)) \to (\text{input}\ id, m(1/\mathfrak{f}))}$$

$$\text{In-4}\ \frac{\begin{array}{c}\mathfrak{f} = 0 \quad \sigma(id) \neq \bot \quad \text{hd}(\sigma(id_I)) = v_{io} \\ (\Gamma \vdash id : \text{enum}\ id') \wedge (\Gamma \vdash id' : \{l_1, ..., l_k\}) \wedge (1 \leq v_{io} \leq k)\end{array}}{\begin{array}{l}(\text{input}\ id, m(\mathfrak{f}, \Gamma, \sigma)) \to (\text{skip}, \\ m(\sigma[l_{v_{io}}/id, \text{tl}(\sigma(id_I))/id_I][\text{“}\sigma(id_{IO}) \cdot \underline{v}_{io}\text{”}/id_{IO}])\end{array}}$$

$$\text{In-5}\ \frac{\begin{array}{c}\mathfrak{f} = 0 \quad \sigma(id) \neq \bot \quad \text{hd}(\sigma(id_I)) = v_{io} \\ (\Gamma \vdash id : \text{enum}\ id') \wedge (\Gamma \vdash id' : \{l_1, ..., l_k\}) \wedge \neg(1 \leq v_{io} \leq k)\end{array}}{(\text{input}\ id, m(\mathfrak{f}, \Gamma, \sigma)) \to (\text{input}\ id, m(1/\mathfrak{f}))}$$

$$\text{In-6}\ \frac{\mathfrak{f} = 0 \quad \sigma(id_I) = \varnothing}{(\text{input}\ id, m(\mathfrak{f}, \sigma)) \to (\text{input}\ id, m(1/\mathfrak{f}))}$$

$$\text{Out-1}\ \frac{\mathfrak{f} = 0 \quad v \in \mathbb{Z}_L}{(\text{output}\ v, m(\mathfrak{f}, \sigma)) \to (\text{skip}, m(\sigma[\text{“}\sigma(id_{IO}) \cdot \overline{v}\text{”}/id_{IO}]))}$$

$$\text{Out-2}\ \frac{\mathfrak{f} = 0 \quad \Gamma \vdash id : \{l_1, ..., l_k\} \wedge v = l_i \in \{l_1, ..., l_k\}}{(\text{output}\ v, m(\mathfrak{f}, \Gamma, \sigma)) \to (\text{skip}, m(\sigma[\text{“}\sigma(id_{IO}) \cdot \overline{i}\text{”}/id_{IO}]))}$$

$$\text{Out-3}\ \frac{\begin{array}{c}\mathfrak{f} = 0 \quad \Gamma \vdash pmpt : \{l_1 : n_1, ..., l_k : n_k\} \\ \text{“}l : n\text{”} \in \{l_1 : n_1, ..., l_k : n_k\}\end{array}}{(\text{output}\ l, m(\mathfrak{f}, \Gamma)) \to (\text{output}\ n, m)}$$

Figure 14: SOS rules for input/output statements

use rule Crash to treat crash as non-terminating execution, telling apart normally terminating executions and others.

Figure 14 shows rules for the execution of input/output statements. As to input, there are conversion from values of type Long to those of Int or enumeration types but not the prompt type. For an enumeration type, the Long-typed value is transformed to the label with index of that value if possible. There is crash when value conversion is impossible. Besides, there is crash when executing input statement with empty input sequence. We use standard list operation hd and tl for fetching the list head(leftmost element) or the list tail(the list by removing its head) respectively [30].

Last, we construct initial state in following steps: First, crash flag $\mathfrak{f}$, overflow flag $\mathfrak{of}$ are zero. Second, type environment is obtained after parsing of the program. Third, every loop counter value in loop$_c$ is initially zero. Fourth, every scalar variable or array element has an entry in value store with some initial value if specified. Last, there is initial input sequence and empty I/O sequence.

4.3 Preliminary terms and notations

We present terms, notations and definitions for program equivalence and backward compatible update classes.

We use Use(e) or Use(S) to denote used variables in an expression e or a statement sequence S; Def(S) denotes the set of defined variables in a statement sequence S. The full definitions of Use and Def are in appendix B.

We use symbol $\in$ for two different purposes: $x \in X$ denotes one variable to be in a set of variables, $s \in S$ denotes a statement to be in a statement sequence. We use the symbol $\subset$ to refer to proper subset relation.

We call an "If" statement or a "while" statement as a compound statement; all other statements are simple statements. We introduce terms referring to a part of a compound statement. Let $s =$ "If(e) then$\{S_t\}$ else$\{S_f\}$" be an "If" statement, we call e in s the predicate expression, S_t/S_f the true/false branch of s.

5. Program equivalence

We consider several types of program changes that are allowed by "observational equivalence" without user assumptions. These changes include: statement reordering or duplication, extra statements unrelated to output(e.g., logging related changes), loop fission or fusion, and extra statements unrelated to output. Our program equivalence ensures two programs produce the same output, which means two programs produce same I/O sequence till any output. The program equivalence is established upon two other kinds of equivalence, namely equivalent terminating computation of a variable and equivalent termination behavior.

We first define terminating and nonterminating execution. Then we present the framework of program equivalence in three steps in which every later step relies on prior ones. We first propose a proof rule ensuring two programs to compute a variable in the same way. We then suggest a condition ensuring two programs to either both terminate or both do not terminate. Finally we describe a condition ensuring two programs to produce the same output sequence. Our proof rule of program equivalence gives program point mapping as well as program state mapping. Though we express the program equivalence as a whole program relation, it is easy to apply the equivalence check for local changes using our framework under user's various assumptions for equivalence.

5.1 Definitions of execution

We define an execution to be a sequence of configurations which are pairs (S, m) where S is a statement sequence and m is a execution state shown in Figure 10. Let (S_1, m_1), (S_2, m_2) be two consecutive configurations in an execution, the later configuration (S_2, m_2) is obtained by applying one semantic rule w.r.t to the configuration (S_1, m_1), denoted $(S_1, m_1) \to (S_2, m_2)$, called one step (of execution). For our convenience, we use the notation $(S, m) \xrightarrow{k} (S', m')$ for k steps execution where $k > 0$. When we do not care the exact (finite) number of steps, we write the execution as $(S, m) \xrightarrow{*} (S', m')$. We express terminating executions, nonterminating executions including crash in Definition 7 and 8.

Definition 7. (Termination) *A statement sequence S normally terminates when started in a state m iff $(S, m) \xrightarrow{*}$ (skip, $m'(\mathfrak{f})$) where $\mathfrak{f} = 0$.*

Definition 8. (Nontermination) *A statement sequence S does not terminate when started in a state m iff, $\forall k > 0 : (S, m) \xrightarrow{k} (S_k, m_k)$ where $S_k \neq$ skip.*

5.2 Equivalent computation for terminating programs

We propose a proof rule under which two terminating programs are computing a variable in the same way. We start by giving the definition of equivalent computation for terminating programs right after this paragraph. Then we present the proof rule of equivalent computation in the same way. We prove that the proof rule ensures equivalent computation for terminating programs by induction on the program size of the two programs in the proof rule. We also list auxiliary lemmas required by the soundness proof for the proof rule for equivalent computation for terminating programs.

Definition 9. (Equivalent computation for terminating programs) *Two statement sequences S_1 and S_2 compute a variable x equivalently when started in states m_1 and m_2 respectively, written $(S_1, m_1) \equiv_x (S_2, m_2)$, iff $(S_1, m_1) \xrightarrow{*} (skip, m_1'(\sigma_{1'}))$ and $(S_2, m_2) \xrightarrow{*} (skip, m_2'(\sigma_{2'}))$ imply $\sigma_{1'}(x) = \sigma_{2'}(x)$.*

5.2.1 Proof rule for equivalent computation for terminating programs

We define a proof rule under which $(S_1, m_1) \equiv_x (S_2, m_2)$ holds for generally constructed initial states m_1 and m_2, written $S_1 \equiv_x^S S_2$. Our proof rule for equivalent computation for terminating programs allows updates including statement reordering or duplication, loop fission or fusion, additional statements unrelated to the computation and statements movement across if-branch.

Definition 12 includes the recursive proof rule of equivalent computing for terminating programs. The base case is the condition for two simple statements in Definition 11. Definition 10 of imported variables captures the variable def-use chain which is the essence of our equivalence. In Definition 10, the Def and Use refer to variables defined or used in a statement (sequence) or an expression similar to those in the optimization chapter in the dragon book [4]; S^i refers to i consecutive copies of a statement sequence S.

Definition 10. (Imported variables) *The imported variables in a sequence of statements S relative to variables X, written $Imp(S, X)$, are defined in one of the following cases:*

1. *$Def(S) \cap X = \emptyset$: $Imp(S, X) = X$;*
2. *S = "$id := e$" or "input id" or "output e" and $Def(S) \cap X \neq \emptyset$:*
   $Imp(S, X) = Use(S) \cup (X \setminus Def(S))$;
3. *S = "If (e) then $\{S_t\}$ else $\{S_f\}$" and $Def(S) \cap X \neq \emptyset$:*
   $Imp(S, X) = Use(e) \cup \bigcup_{y \in X} \big(Imp(S_t, \{y\}) \cup Imp(S_f, \{y\})\big)$;
4. *S = "while(e) $\{S'\}$" where $(Def(S') \cap X) \neq \emptyset$): $Imp(S, X) = \bigcup_{i \geq 0} Imp(S'^i, Use(e) \cup X)$;*
5. *For $k > 0$, $S = s_1; ...; s_{k+1}$:*
   $Imp(S, X) = Imp(s_1; ...; s_k, Imp(s_{k+1}, X))$

Definition 11. (Base cases of the proof rule for equivalent computation for terminating programs) *Two simple statements s_1 and s_2 satisfy the proof rule of equivalent computation of a variable x, written $s_1 \equiv_x^S s_2$, iff one of the following holds:*

1. *$s_1 = s_2$;*
2. *$s_1 \neq s_2$ and one of the following holds:*
   (a) *$s_1 =$ "input id_1", $s_2 =$ "input id_2", $x \notin \{id_1, id_2\}$;*
   (b) *Case* a) *does not hold and $x \notin Def(s_1) \cup Def(s_2)$;*

Definition 12. (Proof rule of equivalent computation for terminating programs) *Two statement sequences S_1 and S_2 satisfy the proof rule of equivalent computation of a variable x, written $S_1 \equiv_x^S S_2$, iff one of the following holds:*

1. *S_1 and S_2 are one statement and one of the following holds:*
   (a) *S_1 and S_2 are simple statement: $s_1 \equiv_x^S s_2$;*
   (b) *$S_1 =$ "If (e) then $\{S_1^t\}$ else $\{S_1^f\}$", $S_2 =$ "If (e) then $\{S_2^t\}$ else $\{S_2^f\}$" such that all of the following hold:*
      - *$x \in Def(S_1) \cap Def(S_2)$;*
      - *$(S_1^t \equiv_x^S S_2^t) \wedge (S_1^f \equiv_x^S S_2^f)$;*

   (c) *$S_1 =$ "while$_{\langle n_1 \rangle}(e)$ $\{S_1''\}$", $S_2 =$ "while$_{\langle n_2 \rangle}(e)$ $\{S_2''\}$" such that both of the following hold:*
      - *$x \in Def(S_1) \cap Def(S_2)$;*
      - *$\forall y \in Imp(S_1, \{x\}) \cup Imp(S_2, \{x\})$: $S_1'' \equiv_y^S S_2''$;*

   (d) *S_1 and S_2 do not define the variable x: $x \notin Def(S_1) \cup Def(S_2)$.*
2. *S_1 and S_2 are not both one statement and one of the following holds:*
   (a) *$S_1 = S_1'; s_1, S_2 = S_2'; s_2$ and last statements both define the variable x such that both of the following hold:*
      - *$\forall y \in Imp(s_1, \{x\}) \cup Imp(s_2, \{x\})$: $S_1' \equiv_y^S S_2'$;*
      - *$s_1 \equiv_x^S s_2$ where $x \in Def(s_1) \cap Def(s_2)$;*

   (b) *Last statement in S_1 or S_2 does not define the variable x: $\big(x \notin Def(s_2) \wedge (S_1 \equiv_x^S S_2')\big) \vee \big(x \notin Def(s_1) \wedge (S_1' \equiv_x^S S_2)\big)$;*
   (c) *$S_1 = S_1'; s_1, S_2 = S_2'; s_2$ and there are statements moving in/out of If statement: $s_1 =$ "If (e) then $\{S_1^t\}$ else $\{S_1^f\}$", $s_2 =$ "If (e) then $\{S_2^t\}$ else $\{S_2^f\}$" such that none of the above cases hold and all of the following hold:*
      - *$\forall y \in Use(e)$: $S_1' \equiv_y^S S_2'$;*
      - *$(S_1'; S_1^t \equiv_x^S S_2'; S_2^t) \wedge (S_1'; S_1^f \equiv_x^S S_2'; S_2^f)$;*

The generalization of definition $S_1 \equiv_x^S S_2$ to a set of variables is as follows.

Definition 13. *Two statement sequences S_1 and S_2 have equivalent computation of variables X, written $S_1 \equiv_X^S S_2$, iff $\forall x \in X$: $S_1 \equiv_x^S S_2$.*

5.2.2 Soundness of the proof rule for equivalent computation for terminating programs

We show that if two programs satisfy the proof rule of equivalent computation of a variable x (Definition 12) and their value stores in initial states agree on values of the imported variables relative to x, then the two programs compute the same value of x if they terminate. We start by proving the theorem for the base cases of terminating computation equivalently.

Theorem 1. *If s_1 and s_2 are simple statements that satisfy the proof rule for equivalent computation of x, $s_1 \equiv_x^S s_2$, and their initial states $m_1(\sigma_1)$ and $m_2(\sigma_2)$ agree on the values of the imported variables relative to x, $\forall y \in Imp(s_1, \{x\}) \cup Imp(s_2, \{x\})$: $\sigma_{s_1}(y) = \sigma_{s_2}(y)$, then s_1 and s_2 equivalently compute x when started in states m_1 and m_2 respectively, $(s_1, m_1) \equiv_x (s_2, m_2)$.*

Proof. The proof is a case analysis according to the cases in the definition of the proof rule for equivalent computation (i.e., Definition 11).

1. $s_1 = s_2$
   Since the two statements are identical, they have the same imported variables. By assumption, the imported variables of s_1 and s_2 have the same initial values, so it is enough to show that the value of x at the end of the computation only depends on the initial values of the imported variables.
   (a) $s_1 = s_2 =$ "skip". In this case, the states before and after the execution of skip are the same and Imp(skip, $\{x\}$) $= \{x\}$.
   (b) $s_1 = s_2 =$ "$lval := e$".
      i. $lval = x$.
         $s_1 = s_2 =$ "$x := e$". By the definition of imported variables, Imp($x := e, \{x\}$) $=$ Use(e). The execution of s_1 proceeds as follows.
         $(x := e, m(\sigma))$
         $\rightarrow(x := \mathcal{E}'[\![e]\!]\sigma, m(\sigma))$ by the EEval' rule
         $\rightarrow$(skip, $m(\sigma[\mathcal{E}'[\![e]\!]\sigma/x])$) by the Assign rule.
         The value of x after the full execution is $\sigma[(\mathcal{E}[\![e]\!]\sigma)/x](x)$ which only depend on the initial values of the imported

variables by the property of the expression meaning function.

ii. $lval \neq id$.
By the definition of imported variables, Imp$(s_1, \{x\}) =$ Imp$(s_2, \{x\}) = \{x\}$. It follows, by assumption, that $\sigma_1(x) = \sigma_2(x)$ and also s_1 terminate, $(s_1, m_1(\sigma_1)) \xrightarrow{*}$ (skip, $m'_1(\sigma'_1)$). Hence, $\sigma'_1(x) = \sigma_1(x)$ by Corollary E.2. Similarly, s_2 terminates, $(s_2, m_2(\sigma_2)) \xrightarrow{*}$ (skip, $m'_2(\sigma'_2)$) and $\sigma'_2(x) = \sigma_2(x)$. Therefore, $\sigma'_2(x) = \sigma_2(x) = \sigma_1(x) = \sigma'_1(x)$ and the theorem holds.

(c) $s_1 = s_2 =$ "input id".

i. $x \in$ Def(input id) $= \{id, id_I, id_{IO}\}$.
By the In rule, the execution of input id is the following.

$$\begin{aligned}&(\text{input } id, m(\sigma))\\ \rightarrow&(\text{skip}, m(\sigma[\text{tl}(\sigma(id_I))/id_I]\\ &[``\sigma(id_{IO}) \cdot \underline{\text{hd}(\sigma(id_I))}"/id_{IO}][\text{hd}(\sigma(id_I))/id])).\end{aligned}$$

The value of x after the execution of "input id" is one of the following:

A. tl$(\sigma(id_I))$ if $x = id_I$.
B. $\sigma_1(id_{IO}) \cdot \underline{\text{hd}(\sigma(id_I))}$ if $x = id_{IO}$.
C. hd$(\sigma(id_I))$ if $x = id$.

By the definition of imported variables, Imp(input $id, \{x\}$) $= \{id_{IO}, id_I\}$. So, in all cases, the value of x only depends on the initial values of the imported variables id_I and id_{IO}.

ii. $x \notin$ Def(input id) $= \{id, id_I, id_{IO}\}$.
By same argument in the subcase $id \neq x$ of case $s_1 = s_2 =$ "$id := e$", the theorem holds.

(d) $s_1 = s_2 =$ "output e".

i. $x = id_{IO}$
By the definition of imported variables, Imp(output $e, \{x\}$) $= \{id_{IO}\} \cup$ Use(e). The execution of s_1 proceeds as follows.

$$\begin{aligned}&(\text{output } e, m(\sigma))\\ \rightarrow&(\text{output } \mathcal{E}[\![e]\!]\sigma, m(\sigma))\\ \rightarrow&(\text{skip}, m(\sigma[``\sigma(id_{IO}) \cdot \mathcal{E}[\![\bar{e}]\!]\sigma"/id_{IO}])).\end{aligned}$$

The value of x after the execution is "$\sigma(id_{IO}) \cdot \mathcal{E}[\![\bar{e}]\!]\sigma$", which only depends on the initial value of the imported variables of the statement "output e" by the expression meaning function.

ii. $x \neq id_{IO}$
By same argument in the subcase $id \neq x$ of case $s_1 = s_2 =$ "$id := e$", the theorem holds.

2. $s_1 \neq s_2$

(a) $s_1 =$ "input id_1", $s_2 =$ "input id_2", $x \notin \{id_1, id_2\}$.

i. $x \in \{id_I, id_{IO}\}$.
By the definition of imported variables, Imp$(s_1, \{x\}) =$ Imp$(s_2, \{x\}) = \{id_{IO}, id_I\}$. It follows, by assumption, that $\sigma_1(y) = \sigma_2(y), \forall y \in \{id_{IO}, id_I\}$. The execution of s_1 proceeds as follows.

$$\begin{aligned}&(s_1, m_1)\\ =&(\text{input } id_1, m_1(\sigma_1))\\ \rightarrow&(\text{skip}, m_1(\sigma_1[\text{tl}(\sigma_1(id_I))/id_I]\\ &[``\sigma_1(id_{IO}) \cdot \underline{\text{hd}(\sigma_1(id_I))}"/id_{IO}][\text{hd}(\sigma_1(id_I))/id_1]))\end{aligned}$$

Let $\sigma'_1 = \sigma_1[\text{tl}(\vec{v})/id_I, ``\sigma_1(id_{IO}) \cdot \underline{\text{hd}(\vec{v})}"/id_{IO}, \text{hd}(\vec{v})/id_1]$. The value of x after the execution of s_1 is one of the following:

A. $\sigma'_1(x) = \text{tl}(\sigma_1(id_I))$ if $x = id_I$.
B. $\sigma'_1(x) = \sigma_1(id_{IO}) \cdot \underline{\text{hd}(\sigma_1(id_I))}$ if $x = id_{IO}$.

Similarly, $(s_2, m_2) \rightarrow$ (skip, $m_2(\sigma_2[\text{tl}(\sigma_2(id_I))/id_2]$ $[``\sigma_2(id_{IO}) \cdot \underline{\text{hd}(\sigma_2(id_I))}"/id_{IO}][\text{hd}(\sigma_2(id_I))/id_2])$).
Let $\sigma'_2 = \sigma_2[\text{tl}(\sigma_2(id_I))/id_I][``\sigma_2(id_{IO}) \cdot \underline{\text{hd}(\sigma_2(id_I))}"/id_{IO}]$ $[\text{hd}(\sigma_2(id_I))/id_2]$. Then the value of x after the execution of s_2 is one of the following:

A. $\sigma'_2(x) = \text{tl}(\sigma_2(id_I))$ if $x = id_I$
B. $\sigma'_2(x) = \sigma_2(id_{IO}) \cdot \underline{\text{hd}(\sigma_2(id_I))}$ if $x = id_{IO}$

Repeatedly, $\sigma_2(id_I) = \sigma_1(id_I)$ and $\sigma_2(id_{IO}) = \sigma_1(id_{IO})$. Therefore, the theorem holds.

ii. $x \notin \{id_I, id_{IO}\}$
Repeatedly, $x \notin \{id_1, id_2\}$. By same argument in the subcase $id \neq x$ of case $s_1 = s_2 =$ "$id := e$", the theorem holds.

(b) all the above cases do not hold and $x \notin$ Def$(s_1) \cup$ Def(s_2)
By same argument in the subcase $id \neq x$ of case $s_1 = s_2 =$ "$id := e$", the theorem holds.

□

Theorem 2. *If statement sequence S_1 and S_2 satisfy the proof rule of equivalent computation of a variable x, $S_1 \equiv^S_x S_2$, and their initial states $m_1(\sigma_1)$ and $m_2(\sigma_2)$ agree on the initial values of the imported variables relative to x, $\forall y \in$ Imp$(S_1, \{x\}) \cup$ Imp$(S_2, \{x\})$: $\sigma_1(y) = \sigma_2(y)$, then S_1 and S_2 equivalently compute the variable x when started in state m_1 and m_2 respectively, $(S_1, m_1) \equiv_x (S_2, m_2)$.*

Proof. By induction on size(S_1)+size(S_2), the sum of the program size of S_1 and S_2.
Base case.
$S_1 \equiv^S_x S_2$ where S_1 and S_2 are two simple statements. This theorem holds by theorem 1.

Induction step
The hypothesis IH is that Theorem 2 holds when size(S_1) + size$(S_2) = k \geq 2$.
Then we show that the Theorem holds when size(S_1)+size$(S_2) = k + 1$. The proof is a case analysis according to the cases in the definition of the proof rule of terminating computation of statement sequence. the two big categories enum

1. S_1 and S_2 are one statement such that one of the following holds:

(a) S_1 and S_2 are If statement that define the variable x:
$S_1 =$ "If (e) then $\{S^t_1\}$ else $\{S^f_1\}$", $S_2 =$ "If (e) then $\{S^t_2\}$ else $\{S^f_2\}$" such that all of the following hold:

- $x \in$ Def$(S_1) \cap$ Def(S_2);
- $S^t_1 \equiv^S_x S^t_2$;
- $S^f_1 \equiv^S_x S^f_2$;

We first show that the evaluations of the predicate expression of S_1 and S_2 produce the same value when started from state $m_1(\sigma_1)$ and $m_2(\sigma_2)$, w.l.o.g. say zero. Next, we show that S^f_1 started in the state m_1 and S^f_2 in the state m_2 equivalently compute the variable x.
In order to show that the evaluations of predicate expression of S_1 and S_2 produce same value when started from state $m_1(\sigma_1)$ and $m_2(\sigma_2)$, we show that the variables used in predicate expression of S_1 and S_2 are a subset of imported variables in S_1 and S_2 relative to x. This is true by the definition of imported variables, Use$(e) \subseteq$ Imp$(S_1, \{x\})$, Use$(e) \subseteq$ Imp$(S_2, \{x\})$. By assumption, the value stores σ_1 and σ_2 agree on the values of the variables used in predicate expression of S_1 and S_2, $\sigma_1(y) = \sigma_2(y), \forall y \in$ Use(e). By the property of

expression meaning function $\mathcal{E}$, the predicate expression of S_1 and S_2 evaluate to the same value when started in states $m_1(\sigma_1)$ and $m_2(\sigma_2)$, $\mathcal{E}[\![e]\!]\sigma_1 = \mathcal{E}[\![e]\!]\sigma_2$, w.l.o.g, $\mathcal{E}[\![e]\!]\sigma_1 = \mathcal{E}[\![e]\!]\sigma_2 = (0, v_{\mathfrak{of}})$. Then the execution of S_1 proceeds as follows.

$(S_1, m_1(\sigma_1))$
$=$ (If (e) then $\{S_1^t\}$ else $\{S_1^f\}, m_1(\sigma_1))$
$\rightarrow$(If $((0, v_{\mathfrak{of}}))$ then $\{S_1^t\}$ else $\{S_1^f\}, m_1(\sigma_1))$
by the EEval' rule
$\rightarrow$(If (0) then $\{S_1^t\}$ else $\{S_1^f\}, m_1(\sigma_1))$
by the E-Oflow1 or E-Oflow2 rule
$\rightarrow(S_1^f, m_1(\sigma_1))$ by the If-F rule.

Similarly, the execution from $(s_2, m_2(\sigma_2))$ gets to $(S_2^f, m_2(\sigma_2))$. By the hypothesis IH, we show that S_1^f and S_2^f compute the variable x equivalently when started in state $m_1(\sigma_1)$ and $m_2(\sigma_2)$ respectively. To do that, we show that all required conditions are satisfied for the application of hypothesis IH.

- size(S_1^f) + size$(S_2^f) < k$.
  Because size$(S_1) = 1+$size$(S_1^t)+$size(S_1^f), size$(S_2) = 1 +$ size(S_2^t) + size(S_2^f).
- the value stores σ_1 and σ_2 agree on the values of the imported variables in S_1^f and S_2^f relative to x, $\sigma_1(y) = \sigma_2(y), \forall y \in$ Imp$(S_1^f, \{x\}) \cup$ Imp$(S_2^f, \{x\})$.
  By the definition of imported variables,
  Imp$(S_1^f, \{x\}) \subseteq$ Imp$(S_1, \{x\})$, Imp$(S_2^f, \{x\}) \subseteq$ Imp$(S_2, \{x\})$.

By the hypothesis IH, S_1^f and S_2^f compute the variable x equivalently when started in state $m_1(\sigma_1)$ and $m_2(\sigma_2)$ respectively. Therefore, the theorem holds.

(b) S_1 and S_2 are while statement that define the variable x: $S_1 =$ "while$_{\langle n_1 \rangle}(e)$ $\{S_1''\}$", $S_2 =$ "while$_{\langle n_2 \rangle}(e)$ $\{S_2''\}$" such that both of the following hold:
- $x \in$ Def$(S_1) \cap$ Def(S_2);
- $S_1'' \equiv_y^S S_2''$ for $\forall y \in$ Imp$(S_1, \{x\}) \cup$ Imp$(S_2, \{x\})$;

By Lemma 5.2, we show S_1 and S_2 compute the variable x equivalently when started from state $m_1(m_c^1, \sigma_1)$ and $m_2(m_c^2, \sigma_2)$ respectively. The point is to show that all required conditions are satisfied for the application of lemma 5.2.
- loop counter value of S_1 and S_2 are zero.
  By our assumption, the loop counter value of S_1 and S_2 are initially zero.
- S_1 and S_2 have same imported variables relative to x, Imp$(S_1, \{x\}) =$ Imp $(S_2, \{x\}) =$ Imp(Δ).
  This is obtained by Lemma 5.3.
- the initial value store σ_1 and σ_2 agree on the values of the imported variables in S_1 and S_2 relative to x, $\sigma_1(y) = \sigma_2(y), \forall y \in$ Imp$(S_1, \{x\}) \cup$ Imp$(S_2, \{x\})$.
  By assumption, this holds.
- S_1'' and S_2'' compute the imported variables in S_1 and S_2 relative to x equivalently, $(S_1'', m_{S_1''}(\sigma_{S_1''})) \equiv_y (S_2'', m_{S_2''}(\sigma_{S_2''})), \forall y \in$ Imp(Δ) with value stores $\sigma_{S_1''}$ and $\sigma_{S_2''}$ agreeing on the values of the imported variables in S_1'' and S_2'' relative to Imp(Δ), $\sigma_{S_1''}(z) = \sigma_{S_2''}(z), \forall z \in$ Imp$(S_1''$, Imp$(\Delta)) \cup$ Imp$(S_2''$, Imp$(\Delta))$.
  By the definition of program size, the sum of the program size of S_1' and S_2' is less than k, size(S_1'') + size$(S_2'') < k$. By the hypothesis IH, S_1'' and S_2'' compute the imported variables in S_1 and S_2 relative to x equivalently when started in states $m_{S_1''}(\sigma_{S_1''})$ and $m_{S_2''}(\sigma_{S_2''})$ with value store $\sigma_{S_1''}$ and $\sigma_{S_2''}$ agreeing on the values of the imported variables in S_1'' and S_2'' relative to the variables Imp(Δ).

By Lemma 5.2, we show S_1 and S_2 compute the variable x equivalently when started from state $m_1(m_c^1, \sigma_1)$ and $m_2(m_c^2, \sigma_2)$ respectively. The theorem holds.

(c) S_1 and S_2 do not define the variable x: $x \notin$ Def$(S_1) \cup$ Def(S_2).
By the definition of imported variable, the imported variables in S_1 and S_2 relative to x are both x, Imp$(S_1, \{x\}) =$ Imp$(S_2, \{x\}) = \{x\}$. By assumption, the initial values σ_1 and σ_2 agree on the value of the variable x, $\sigma_1(x) = \sigma_2(x)$. In addition, by assumption, execution of S_1 and S_2 when started in state $m_1(\sigma_1)$ and $m_2(\sigma_2)$ terminate, $(S_1, m_1(\sigma_1)) \xrightarrow{*}$ (skip, $m_1'(\sigma_1'))$, $(S_2, m_2(\sigma_2)) \xrightarrow{*}$ (skip, $m_2'(\sigma_2'))$.
Finally, by Corollary E.2, the value of x is not changed in execution of S_1 and S_2, $\sigma_1'(x) = \sigma_1(x) = \sigma_2(x) = \sigma_2'(x)$.
The theorem holds.

2. S_1 and S_2 are not both one statement such that one of the following holds:

(a) Last statements both define the variable x such that all of the following hold:
- $S_1' \equiv_y^S S_2', \forall y \in$ Imp$(s_1, \{x\}) \cup$ Imp$(s_2, \{x\})$;
- $x \in$ Def$(s_1) \cap$ Def(s_2);
- $s_1 \equiv_x^S s_2$;

We show that S_1' and S_2' compute the imported variables in s_1 and s_2 relative to the variable x equivalently when started in state $m_1(\sigma_1)$ and $m_2(\sigma_2)$ respectively by the hypothesis IH. To do that, we show the required conditions are satisfied for applying the hypothesis IH.
- size(S_1') + size$(S_2') < k$.
  By the definition of program size, size$(s_1) \geq 1$, size$(s_2) \geq 1$. Hence, size(S_1') + size$(S_2') < k$.
- the executions from $(S_1, m_1(\sigma_1))$ and $(S_2, m_2(\sigma_2))$ terminate respectively,
  $(S_1', m_1(\sigma_1)) \xrightarrow{*}$ (skip, $m_1''(\sigma_1''))$, $(S_2', m_2(\sigma_2)) \xrightarrow{*}$ (skip, $m_2''(\sigma_2''))$.
  By assumption, the execution from $(S_1, m_1(\sigma_1))$ and $(S_2, m_2(\sigma_2))$ terminate, then the execution of S_1' and S_2' from state $m_1(\sigma_1)$ and $m_2(\sigma_2)$ terminate, $(S_1', m_1(\sigma_1)) \xrightarrow{*}$ (skip, $m_1''(\sigma_1''))$, $(S_2', m_2(\sigma_2)) \xrightarrow{*}$ (skip, $m_2''(\sigma_2''))$.
- the initial value stores agree on the values of the variables:
  Imp$(S_1'$, Imp$(s_1, \{x\})) \cup$ Imp$(S_2'$, Imp$(s_2, \{x\}))$.
  By Lemma 5.3, s_1 and s_2 have the same imported variables relative to x, Imp$(s_1, \{x\}) =$ Imp$(s_2, \{x\})$ = Imp(x). By the definition of imported variables, imported variables in S_1' relative to Imp(x) are same as the imported variables in S_1 relative to x, Imp$(S_1'$, Imp$(s_1, \{x\})) =$ Imp$(S_1, \{x\})$. Similarly, Imp$(S_2'$, Imp$(s_2, \{x\})) =$ Imp$(S_2, \{x\})$. Then, by assumption, the initial value stores agree on the values of the variables Imp$(S_1'$, Imp$(s_1, \{x\}))$ and $\forall y \in$ Imp$(S_1'$, Imp$(s_1, \{x\})) \cup$ Imp$(S_2'$, Imp$(s_2, \{x\}))$, Imp$(S_2'$, Imp$(s_2, \{x\}))$, $\sigma_1(y) = \sigma_2(y)$.

By the hypothesis IH, after the full execution of S_1' from state $m_1(\sigma_1)$ and the execution of S_2' from state $m_2(\sigma_2)$, the value stores agree on the values of the imported variables in s_1 and s_2 relative to x, $\sigma_1''(y) = \sigma_2''(y), \forall y \in$ Imp $(x) =$ Imp $(s_1, \{x\}) =$ Imp $(s_2, \{x\})$.
Then, we show s_1 and s_2 compute x equivalently. By Corollary E.1, s_1 and s_2 continue execution after the full execution of S_1' and S_2' respectively, $(S_1'; s_1, m_1(\sigma_1)) \xrightarrow{*} (s_1, m_1''(\sigma_1''))$, $(S_2'; s_2, m_2(\sigma_2)) \xrightarrow{*} (s_2, m_2''(\sigma_2''))$. When s_1 and s_2 are while statements, by our assumption of unique

loop labels, s_1 is not in S'_1. By Corollary E.4, the loop counter value of s_1 is not redefined in the execution of S'_1. Similarly, the loop counter value of s_2 is not redefined in the execution of S'_2. By the hypothesis IH again, after the full execution of s_1 and s_2, the value stores agree on the value of x, $(s_1, m''_1(\sigma''_1)) \xrightarrow{*} (\text{skip}, m'_1(\sigma'_1)), (s_2, m''_2(\sigma''_2)) \xrightarrow{*} (\text{skip}, m'_2(\sigma'_2))$ such that $\sigma'_1(x) = \sigma'_2(x)$. The theorem holds.

(b) One last statement does not define the variable x: W.l.o.g., $(x \notin \text{Def}(s_2)) \wedge (S_1 \equiv^S_x S'_2)$.
We show that S_1 and S'_2 compute the variable x equivalently when started from state $m_1(\sigma_1)$ and $m_2(\sigma_2)$ by the hypothesis IH. First, by the definition of program size, $\text{size}(s_2) \geq 1$. Hence, $\text{size}(S_1) + \text{size}(S'_2) \leq k$. Next, by the definition of imported variables, $\text{Imp}\ (S'_2, \{x\}) \subseteq \text{Imp}\ (S_2, \{x\})$. By assumption, $\sigma_1(y) = \sigma_2(y)$ for $\forall y \in \text{Imp}\ (S'_2, \{x\}) \cup \text{Imp}\ (S_1, \{x\})$. By the hypothesis IH, S_1 and S'_2 compute the variable x equivalently when started in state $m_1(\sigma_1)$ and $m_2(\sigma_2)$ respectively, $(S'_2, m_2(\sigma_2)) \xrightarrow{*} (\text{skip}, m''_2(\sigma''_2))$, $(S_1, m_1(\sigma_1)) \xrightarrow{*} (\text{skip}, m'_1(\sigma'_1))$ such that $\sigma'_1(x) = \sigma''_2(x)$.
Then, we show that S_1 and S_2 compute the variable x equivalently after the full execution of s_2. By Corollary E.1, s_2 continues execution immediately after the full execution of S'_2, $(S'_2; s_2, m_2) \xrightarrow{*} (s_2, m''_2)$. By assumption, the execution from $(S'_2; s_2, m_2)$ terminates, $(s_2, m''_2(\sigma''_2)) \xrightarrow{*} (\text{skip}, m'_2(\sigma'_2))$. By Corollary E.2, the value of x is not changed in the execution of s_2, $\sigma'_2(x) = \sigma''_2(x)$. Hence, $\sigma'_1(x) = \sigma'_2(x)$. The theorem holds.

(c) There are statements moving in/out of If statement: $s_1 =$ "If (e) then $\{S^t_1\}$ else $\{S^f_1\}$", $s_2 =$ "If (e) then $\{S^t_2\}$ else $\{S^f_2\}$" such that none of the above cases hold and all of the following hold:
- $S'_1 \equiv^S_y S'_2$ for $\forall y \in \text{Use}(e)$;
- $S'_1; S^t_1 \equiv^S_x S'_2; S^t_2$;
- $S'_1; S^f_1 \equiv^S_x S'_2; S^f_2$;
- $x \in \text{Def}(s_1) \cap \text{Def}(s_2)$;

Repeatedly $S_1 = S'_1; s_1, S_2 = S'_2; s_2$. We first show that, after the full execution of S'_1 and S'_2 started in state m_1 and m_2, the predicate expression of s_1 and s_2 evaluate to the same value, w.l.o.g, zero. Next we show that S_1 and $S'_1; S^f_1$ compute the variable x equivalently when (1) both started in state m_1 and (2) the predicate expression of s_1 evaluates to zero after the full execution of S'_1 started in state m_1, similarly S_2 and $S'_2; S^f_2$ compute the variable x equivalently when (1) both started in the state m_2 and (2) the predicate expression of s_2 evaluates to zero after the full execution of S'_1 when started in state m_2. Last we prove the theorem by showing that $S'_1; S^f_1$ started in state m_1 and $S'_2; S^f_2$ started in state m_2 compute the variable x equivalently.

In order to show that S'_1 and S'_2 compute the variables used in predicate expression of s_1 and s_2 equivalently by the hypothesis IH, we show that all required conditions are satisfied for the application of hypothesis IH.
- $\text{size}(S'_1) + \text{size}(S'_2) < k$.
  The sum of program size of S'_1 and S'_2 are less than k by the definition of program size for s_1 and s_2, $\text{size}(S'_1) + \text{size}(S'_2) < k$.
- the execution of S'_1 and S'_2 terminate, $(S'_1, m_1) \xrightarrow{*} (\text{skip}, m''_1(\sigma''_1))$, and $(S'_2, m_2) \xrightarrow{*} (\text{skip}, m''_2(\sigma''_2))$.
  By assumption, the execution of S_1 and S_2 from the state m_1 and m_2 respectively terminate, then the execution of S'_1 and S'_2 terminate when started in state m_1 and m_2 respectively.
- the initial value stores σ_1 and σ_2 agree on the values of the imported variables in S'_1 and S'_2 relative to the variables used in the predicate expression of s_1 and s_2.
  By Lemma 5.3, the imported variables in S'_1 and S'_2 relative to the variables used in predicate expression of s_1 and s_2 are same, $\text{Imp}(S'_1, \text{Use}(e)) = \text{Imp}(S'_2, \text{Use}(e)) = \text{Imp}(e)$. By the definition of imported variable, the imported variables in S'_1 relative to the variables used in predicate expression of s_1 are a subset of the imported variables in S_1 relative to x respectively, $\text{Imp}(S'_1, \text{Use}(e)) \subseteq \text{Imp}(S'_1, \text{Imp}(s_1, \{x\})) = \text{Imp}(S_1, \{x\})$. Similarly $\text{Imp}(S'_2, \text{Use}(e)) \subseteq \text{Imp}(S_2, \{x\})$. Then, by assumption, the initial value stores agree on the values of the imported variables in S'_1 and S'_2 relative to the variables used in the predicate expression of s_1 and s_2, $\sigma_1(y) = \sigma_2(y), \forall y \in \text{Imp}(e) = \text{Imp}(S'_1, \text{Use}(e)) = \text{Imp}(S'_2, \text{Use}(e))$.

By the hypothesis IH, after the full execution of S'_1 and S'_2, the value stores agree on the values of the variables used in the predicate expression of s_1 and s_2, $\sigma''_1(y) = \sigma''_2(y), \forall y \in \text{Use}(e)$. By Corollary E.1, s_1 and s_2 continue execution after the full execution of S'_1 and S'_2 respectively, $(S'_1; s_1, m_1) \xrightarrow{*} (s_1, m''_1(\sigma''_1))$, and $(S'_2; s_2, m_2) \xrightarrow{*} (s_2, m''_2(\sigma''_2))$.
By the property of expression meaning function $\mathcal{E}$, expression e evaluates to the same value w.r.t value stores σ''_1 and σ''_2, w.l.o.g., zero, $\mathcal{E}[\![e]\!]\sigma''_1 = \mathcal{E}[\![e]\!]\sigma''_2 = 0$. Then the execution of s_1 proceeds as follows.

$$\begin{aligned}
&(s_1, m''_1(\sigma''_1))\\
=\ &(\text{If } (e) \text{ then } \{S^t_1\} \text{ else } \{S^f_1\}, m''_1(\sigma''_1))\\
\rightarrow&(\text{If } (0) \text{ then } \{S^t_1\} \text{ else } \{S^f_1\}, m''_1(\sigma''_1)) \text{ by the EEval rule.}\\
\rightarrow&(S^f_1, m''_1(\sigma''_1)) \text{ by the If-F rule.}
\end{aligned}$$

Similarly, the execution from $(s_2, m''_2(\sigma''_2))$ gets to $(S^f_2, m''_2(\sigma''_2))$.

Then, we show that S_1 and $S'_1; S^f_1$ compute the variable x equivalently when both started from state $m_1(\sigma_1)$. The execution of $S'_1; S^f_1$ started from state m_1 also gets to configuration $(S^f_1, m''_1(\sigma''_1))$ because execution of $S_1 = S'_1; s_1$ and $S'_1; S^f_1$ share the common execution $(S'_1, m_1) \xrightarrow{*} (\text{skip}, m''_1(\sigma''_1))$. By Corollary E.1, S^f_1 continues execution after the full execution of S'_1, $(S'_1; S^f_1, m_1) \xrightarrow{*} (S^f_1, m''_1)$. Therefore, the execution of S_1 and $S'_1; S^f_1$ from state m_1 compute the variable x equivalently because both executions get to same intermediate configuration. Similarly, S_2 and $S'_2; S^f_2$ compute the variable x equivalently when both started from state $m_2(\sigma_2)$.
Lastly, we show that $S'_1; S^f_1$ and $S'_2; S^f_2$ compute the variable x equivalently when started in states $m_1(\sigma_1)$ and $m_2(\sigma_2)$ respectively by the hypothesis IH. To do that, we show that all required conditions are satisfied for the application of hypothesis IH.
- $\text{size}(S'_1; S^f_1) + \text{size}(S'_2; S^f_2) < k$.
  This is obtained by the definition of program size.
- execution of $S'_1; S^f_1$ and $S'_2; S^f_2$ terminate when started in state $m_1(\sigma_1)$ and $m_2(\sigma_2)$ respectively.
  This is obtained by above argument.
- $\sigma_1(y) = \sigma_2(y), \forall y \in \text{Imp}(S'_1; S^f_1, \{x\}) \cup \text{Imp}(S'_2; S^f_2, \{x\})$.
  We show that $\text{Imp}(S'_1; S^f_1, \{x\}) \subseteq \text{Imp}(S_1, \{x\})$ as follows.

  $\text{Imp}(S^f_1, \{x\})$

$\subseteq$ Imp$(s_1, \{x\})$ (1) by the definition of imported variables.

Imp$(S'_1; S^f_1, \{x\})$
= Imp$(S'_1, \text{Imp}(S^f_1, \{x\}))$ by Lemma C.1
$\subseteq$ Imp$(S'_1, \text{Imp}(s_1, \{x\}))$ by (1)
= Imp$(S_1, \{x\})$ by the definition of imported variables.

Similarly, Imp$(S'_2; S^f_2, \{x\}) \subseteq$ Imp$(S_2, \{x\})$. Then, by assumption, the initial value stores agree on the values of the imported variables in $S'_1; S^f_1$ and $S'_2; S^f_2$ relative to x.

Then, by the hypothesis IH, after the full execution of $S'_1; S^f_1$ and $S'_2; S^f_2$, the value stores agree on the value of x, $(S'_1; S^f_1, m_1) \xrightarrow{*}$ (skip, $m'_1(\sigma'_1)$), $(S'_2; S^f_2, m_2) \xrightarrow{*}$ (skip, $m'_2(\sigma'_2)$) such that $\sigma'_1(x) = \sigma'_2(x)$.
In conclusion, after execution of S_1 and S_2, the value stores agree on the value of x. Therefore, the theorem holds.

□

5.2.3 Supporting lemmas for the soundness proof of equivalent computation for terminating programs

The lemmas include the proof of two while statements computing a variable equivalently used in the proof of Theorem 2 and the property that two programs have same imported variables relative to a variable x if the two programs satisfy the proof rule of equivalent computation of the variable x. From the proof rule of terminating computation of a variable x equivalently, we have the two programs either both define x or both do not.

Lemma 5.1. *Let s_1 = "while$_{\langle n_1\rangle}(e)$ $\{S_1\}$" and s_2 = "while$_{\langle n_2\rangle}(e)$ $\{S_2\}$" be two while statements with the same set of imported variables relative to a variable x (defined in s_1 and s_2), Imp(x), and whose loop bodies S_1 and S_2 terminatingly compute the variables in Imp(x) equivalently when started in states that agree on the values of the variables imported by S_1 or S_2 relative to Imp(x):*

- *$x \in$ Def$(s_1) \cap$ Def(s_2);*
- *Imp$(s_1, \{x\}) =$ Imp$(s_2, \{x\}) =$ Imp(x);*
- *$\forall y \in$ Imp$(x), \forall m_{S_1}(\sigma_{S_1}), m_{S_2}(\sigma_{S_2})$:
$((\forall z \in \text{Imp}(S_1, \text{Imp}(x)) \cup \text{Imp}(S_2, \text{Imp}(x))\ :\ \sigma_{S_1}(z) = \sigma_{S_2}(z)) \Rightarrow (S_1, m_{S_1}(\sigma_{S_1})) \equiv_y (S_2, m_{S_2}(\sigma_{S_2})))$.*

If the executions of s_1 and s_2 terminate when started in states $m_1(loop^1_c, \sigma_1)$ and $m_2(loop^2_c, \sigma_2)$ in which s_1 and s_2 have not already executed (loop counter initially 0: $loop^1_c(n_1) = loop^2_c(n_2) = 0$), and whose value stores σ_1 and σ_2 agree on the values of the variables in Imp(x), $\forall y \in$ Imp(x), $\sigma_1(y) = \sigma_2(y)$, then, for any positive integer i, one of the following holds:

1. *The loop counters for s_1 and s_2 are always less than i:
$\forall m'_1, m'_2$ such that $(s_1, m_1) \xrightarrow{*} (S'_1, m'_1(loop^{1'}_c)$ and $(s_2, m_2) \xrightarrow{*} (S'_2, m'_2(loop^{2'}_c))$,
$loop^{1'}_c(n_1) < i$ and $loop^{2'}_c(n_2) < i$;*
2. *There are two configurations (s_1, m_{1_i}) and (s_2, m_{2_i}) reachable from (s_1, m_1) and (s_2, m_2), respectively, in which the loop counters of s_1 and s_2 are equal to i and value stores agree on the values of imported variables relative to x and, for every state in execution, $(s_1, m_1) \xrightarrow{*} (s_1, m_{1_i})$ or $(s_2, m_2) \xrightarrow{*} (s_2, m_{2_i})$ the loop counters for s_1 and s_2 are less than or equal to i respectively:
$\exists (s_1, m_{1_i}), (s_2, m_{2_i})\ :\ (s_1, m_1) \xrightarrow{*} (s_1, m_{1_i}(loop^{1_i}_c, \sigma_{1_i})) \wedge (s_2, m_2) \xrightarrow{*} (s_2, m_{2_i}(loop^{2_i}_c, \sigma_{2_i}))$ where*
   - *$loop^{1_i}_c(n_1) = loop^{2_i}_c(n_2) = i$; and*
   - *$\forall y \in$ Imp$(x)\ :\ \sigma_{1_i}(y) = \sigma_{2_i}(y)$ and*
   - *$\forall m'_1\ :\ (s_1, m_1) \xrightarrow{*} (S'_1, m'_1(loop^{1'}_c)) \xrightarrow{*} (s_1, m_{1_i}(loop^{1_i}_c, \sigma_{1_i}))$, $loop^{1'}_c(n_1) \le i$; and*
   - *$\forall m'_2\ :\ (s_2, m_2) \xrightarrow{*} (S'_2, m'_2(loop^{2'}_c)) \xrightarrow{*} (s_2, m_{2_i}(loop^{2_i}_c, \sigma_{2_i}))$, $loop^{2'}_c(n_2) \le i$;*

Proof. By induction on i.
Base case. $i = 1$.
By assumption, initial loop counters of s_1 and s_2 are of value zero. Initial value stores σ_1 and σ_2 agree on the values of the variables in Imp(x). Then we show one of the following cases hold:

1. The loop counters for s_1 and s_2 are always less than 1:
$\forall m'_1, m'_2$ such that $(s_1, m_1) \xrightarrow{*} (S'_1, m'_1(\text{loop}^{1'}_c))$ and $(s_2, m_2) \xrightarrow{*} (S'_2, m'_2(\text{loop}^{2'}_c))$, loop$^{1'}_c(n_1) < 1$ and loop$^{2'}_c(n_2) < 1$;
2. There are two configurations (s_1, m_{1_1}) and (s_2, m_{2_1}) reachable from (s_1, m_1) and (s_2, m_2), respectively, in which the loop counters of s_1 and s_2 are equal to 1 and value stores agree on the values of imported variables relative to x and, for every state in execution, $(s_1, m_1) \xrightarrow{*} (s_1, m_{1_1})$ or $(s_2, m_2) \xrightarrow{*} (s_2, m_{2_1})$ the loop counters for s_1 and s_2 are less than or equal to one respectively:
$\exists (s_1, m_{1_1}), (s_2, m_{2_1})\ :\ (s_1, m_1) \xrightarrow{*} (s_1, m_{1_1}(\text{loop}^{1_1}_c, \sigma_{1_1})) \wedge (s_2, m_2) \xrightarrow{*} (s_2, m_{2_1}(\text{loop}^{2_1}_c, \sigma_{2_1}))$ where
   - loop$^{1_1}_c(n_1) =$ loop$^{2_1}_c(n_2) = 1$; and
   - $\forall y \in$ Imp$(x)\ :\ \sigma_{1_1}(y) = \sigma_{2_1}(y)$; and
   - $\forall m'_1\ :\ (s_1, m_1) \xrightarrow{*} (S'_1, m'_1(\text{loop}^{1'}_c)) \xrightarrow{*} (s_1, m_{1_1}(\text{loop}^{1_1}_c, \sigma_{1_1}))$, loop$^{1'}_c(n_1) \le 1$; and
   - $\forall m'_2\ :\ (s_2, m_2) \xrightarrow{*} (S'_2, m'_2(\text{loop}^{2'}_c)) \xrightarrow{*} (s_2, m_{2_1}(\text{loop}^{2_1}_c, \sigma_{2_1}))$, loop$^{2'}_c(n_2) \le 1$.

We show evaluations of the predicate expression of s_1 and s_2 w.r.t value stores σ_1 and σ_2 produce same value. By the definition of imported variables, Imp$(s_1, \{x\}) = \bigcup_{j \ge 0}$ Imp$(S^j_1, \{x\} \cup$ Use$(e))$. By our notation of S^0, $S^0_1 =$ skip. By the definition of imported variables, Imp$(S^0_1, \{x\} \cup \text{Use}(e)) = \{x\} \cup$ Use(e). Then Use$(e) \subseteq$ Imp(x). By assumption, value stores σ_1 and σ_2 agree on the values of the variables in Use(e). By Lemma D.1, the predicate expression e of s_1 and s_2 evaluates to same value v w.r.t value stores σ_1, σ_2, $\mathcal{E}'[\![e]\!]\sigma_1 = \mathcal{E}'[\![e]\!]\sigma_2 = v$. Then there are two possibilities to consider.

1. $\mathcal{E}'[\![e]\!]\sigma_1 = \mathcal{E}'[\![e]\!]\sigma_2 = v = 0$
The execution from $(s_1, m_1(\text{loop}^1_c, \sigma_1))$ proceeds as follows.

   $(s_1, m_1(\text{loop}^1_c, \sigma_1))$
   = (while$_{\langle n_1\rangle}(e)$ $\{S_1\}, m_1(\text{loop}^1_c))$
   $\rightarrow$(while$_{\langle n_1\rangle}(0)$ $\{S_1\}, m_1(\text{loop}^1_c))$ by the EEval' rule
   $\rightarrow$(skip, $m_1(\text{loop}^1_c[0/n_1]))$ by the Wh-F rule.

   Similarly, $(s_2, m_2(\text{loop}^2_c, \sigma_2)) \xrightarrow{2}$ (skip, $m_2(\text{loop}^2_c[0/n_2]))$.
   In conclusion, the loop counters of s_1 and s_2 in any states of the execution from (s_1, m_1) and (s_2, m_2) respectively are less than 1, $\forall m'_1, m'_2$ such that $(s_1, m_1) \xrightarrow{*} (S_1{}', m'_1(\text{loop}^{1'}_c)$ and $(s_2, m_2) \xrightarrow{*} (S_2{}', m'_2(\text{loop}^{2'}_c))$, loop$^{1'}_c(n_1) < 1$, loop$^{2'}_c(n_2) < 1$.
2. $\mathcal{E}'[\![e]\!]\sigma_1 = \mathcal{E}'[\![e]\!]\sigma_2 = v \ne 0$
The execution from $(s_1, m_1(\text{loop}^1_c, \sigma_1))$ proceeds as follows.

   $(s_1, m_1(\text{loop}^1_c, \sigma_1))$
   = (while$_{\langle n_1\rangle}(e)$ $\{S_1\}, m_1(\text{loop}^1_c, \sigma_1))$
   $\rightarrow$(while$_{\langle n_1\rangle}(v)$ $\{S_1\}, m_1(\text{loop}^1_c, \sigma_1))$ by the EEval' rule
   $\rightarrow(S_1;$ while$_{\langle n_1\rangle}(e)$ $\{S_1\}, m_1(\text{loop}^1_c[1/(n_1)], \sigma_1))$
   by the Wh-T rule.

Similarly, $(s_2, m_2(\text{loop}_c^2, \sigma_2)) \xrightarrow{2} (S_2; \text{while}_{\langle n_2 \rangle}(e)\{S_2\}, m_2$ $(\text{loop}_c^2[1/(n_2)], \sigma_2))$. Then, the loop counters of s_1 and s_2 are 1, value stores σ_{1_1} and σ_{2_1} agree on values of variables in Imp(x): $\text{loop}_c^1[1/n_1](n_1) =$ $\text{loop}_c^2[1/(n_2)](n_2) = 1$; and $\forall y \in \text{Imp}(x), \sigma_{1_1}(y) = \sigma_{2_1}(y)$. By assumption, the execution of s_1 terminates when started in the state $m_1(, \sigma_1)$, then the execution of S_1 terminates when started in the state $m_1(\text{loop}_c^1[1/(n_1)], \sigma_1)$, $(s_1, m_1) \xrightarrow{*}$ $(S_1; \text{while}_{\langle n_1 \rangle}(e)\{S_1\}, m_1(\text{loop}_c^1[1/(n_1)], \sigma_1)) \xrightarrow{*} (\text{skip}, m_1') \Rightarrow$ $(S_1, m_1(\text{loop}_c^1[1/(n_1)], \sigma_1)) \xrightarrow{*} (\text{skip}, m_{1_1}(\text{loop}_c^{1_1}, \sigma_{1_1}))$. Similarly, the execution of S_2 terminates when started in the state $m_2(\text{loop}_c^2[1/(n_2)], \sigma_2)$, $(S_2, m_2(\text{loop}_c^2[1/(n_2)], \sigma_2)) \xrightarrow{*} (\text{skip}, m_{2_1}(\text{loop}_c^{2_1}, \sigma_{2_1}))$. We show that, after the full execution of S_1 and S_2, the following four properties hold.

- The loop counters of s_1 and s_2 are of value 1, $\text{loop}_c^{1_1}(n_1) = \text{loop}_c^{2_1}(n_2) = 1$.
  By assumption of unique loop labels, $s_1 \notin S_1$. Then, the loop counter value of n_1 is not redefined in the execution of S_1 by corollary E.4, $\text{loop}_c^1[1/(n_1)](n_1) = \text{loop}_c^{1_1}(n_1) = 1$. Similarly, the loop counter value of n_2 is not redefined in the execution of S_2, $\text{loop}_c^2[1/(n_2)](n_2) = \text{loop}_c^{2_1}(n_2) = 1$.
- In any state in the execution $(s_1, m_1) \xrightarrow{*} (s_1, m_{1_1}(\text{loop}_c^{1_1}, \sigma_{1_1}))$, the loop counter of s_1 is less than or equal to 1.
  The loop counter of s_1 is zero in any of the two states in the one step execution $(s_1, m_1) \rightarrow (\text{while}_{\langle n_1 \rangle}(v)\ \{S_1\}, m_1(\text{loop}_c^1, , \sigma_1))$, and the loop counter of s_1 is 1 in any states in the execution
  $(S_1; \text{while}_{\langle n_1 \rangle}(e)\ \{S_1\}, m_1(\text{loop}_c^1[i/(n_1)], \sigma_1)) \xrightarrow{*}$ $(s_1, m_{1_1}(\text{loop}_c^{1_1}, \sigma_{1_1}))$.
- In any state in the executions $(s_2, m_2) \xrightarrow{*} (s_2, m_{2_1}(\text{loop}_c^{2_1}, \sigma_{2_1}))$, the loop counter of s_2 is less than or equal to 1.
  By similar argument for the loop counter of s_1.
- The value stores σ_{1_1} and σ_{2_1} agree on the values of the imported variables in s_1 and s_2 relative to the variable x: $\forall y \in \text{Imp}(x), \sigma_{1_1}(y) = \sigma_{2_1}(y)$.
  We show that the imported variables in S_1 relative to those in Imp(x) are a subset of Imp(x).

  $$\begin{aligned}
  & \text{Imp}(S_1, \text{Imp}(x)) \\
  =\ & \text{Imp}(S_1, \text{Imp}(s_1, \{x\} \cup \text{Use}(e))) \text{ by the definition of Imp}(x) \\
  =\ & \text{Imp}(S_1, \textstyle\bigcup_{j \geq 0} \text{Imp}(S_1^j, \{x\} \cup \text{Use}(e))) \\
  & \text{by the definition of imported variables} \\
  =\ & \textstyle\bigcup_{j \geq 0} \text{Imp}(S_1, \text{Imp}(S_1^j, \{x\} \cup \text{Use}(e))) \text{ by Lemma C.2} \\
  =\ & \textstyle\bigcup_{j > 0} \text{Imp}(S_1^j, \{x\} \cup \text{Use}(e)) \text{ by Lemma C.1} \\
  \subseteq\ & \textstyle\bigcup_{j \geq 0} \text{Imp}(S_1^j, \{x\} \cup \text{Use}(e)) \\
  =\ & \text{Imp}(s_1, \{x\} \cup \text{Use}(e)) = \text{Imp}(x).
  \end{aligned}$$

  Similarly, $\text{Imp}(S_2, \text{Imp}(x)) \subseteq \text{Imp}(x)$. Consequently, the value stores σ_{1_1} and σ_{2_1} agree on the values of the imported variables in S_1 and S_2 relative to those in Imp(x), $\forall y \in \text{Imp}(S_1, \text{Imp}(x)) \cup \text{Imp}(S_2, \text{Imp}(x)), \sigma_1(y) = \sigma_2(y)$. Because S_1 and S_2 have computation of every variable in Imp(x) equivalently when started in states agreeing on the values of the imported variables relative to Imp(x), then value store σ_{1_1} and σ_{2_1} agree on the values of the variables Imp(x), $\forall y \in \text{Imp}(x), \sigma_{1_1}(y) = \sigma_{2_1}(y)$.

It follows that, by corollary E.1,
$(S_1; s_1, m_1(\text{loop}_c^1[1/n_1], \sigma_1)) \xrightarrow{*} (s_1, m_{1_1}(\text{loop}_c^{1_1}, \sigma_{1_1}))$ and $(S_2; s_2, m_2(\text{loop}_c^2[1/n_2], \sigma_2)) \xrightarrow{*} (s_2, m_{2_1}(\text{loop}_c^{2_1}, \sigma_{2_1}))$.

Induction Step.

The induction hypothesis IH is that, for a positive integer i, one of the following holds:

1. The loop counters for s_1 and s_2 are always less than i:
   $\forall m_1', m_2'$ such that $(s_1, m_1) \xrightarrow{*} (S_1', m_1'(\text{loop}_c^{1'})$ and $(s_2, m_2) \xrightarrow{*} (S_2', m_2'(\text{loop}_c^{2'}))$,
   $\text{loop}_c^{1'}(n_1) < i$ and $\text{loop}_c^{2'}(n_2) < i$;
2. There are two configurations (s_1, m_{1_i}) and (s_2, m_{2_i}) reachable from (s_1, m_1) and (s_2, m_2), respectively, in which the loop counters of s_1 and s_2 are equal to i and value stores agree on the values of imported variables relative to x and, for every state in execution, $(s_1, m_1) \xrightarrow{*} (s_1, m_{1_i})$ and $(s_2, m_2) \xrightarrow{*} (s_2, m_{2_i})$ the loop counters for s_1 and s_2 are less than or equal to i respectively:
   $\exists (s_1, m_{1_i}), (s_2, m_{2_i}) : (s_1, m_1) \xrightarrow{*} (s_1, m_{1_i}(\text{loop}_c^{1_i}, \sigma_{1_i})) \wedge$ $(s_2, m_2) \xrightarrow{*} (s_2, m_{2_i}(\text{loop}_c^{2_i}, \sigma_{2_i}))$ where
   - $\text{loop}_c^{1_i}(n_1) = \text{loop}_c^{2_i}(n_2) = i$; and
   - $\forall y \in \text{Imp}(x), \sigma_{1_i}(y) = \sigma_{2_i}(y)$; and
   - $\forall m_1' : (s_1, m_1) \xrightarrow{*} (S_1', m_1'(\text{loop}_c^{1'}) \xrightarrow{*}$ $(s_1, m_{1_i}(\text{loop}_c^{1_i}, \sigma_{1_i}))$, $\text{loop}_c^{1'}(n_1) \leq i$; and
   - $\forall m_2' : (s_2, m_2) \xrightarrow{*} (S_2', m_2'(\text{loop}_c^{2'})) \xrightarrow{*}$ $(s_2, m_{2_i}(\text{loop}_c^{2_i}, \sigma_{2_i}))$, $\text{loop}_c^{2'}(n_2) \leq i$.

Then we show that, for the positive integer $i + 1$, one of the following holds:

1. The loop counters for s_1 and s_2 are always less than $i + 1$:
   $\forall m_1', m_2'$ such that $(s_1, m_1) \xrightarrow{*} (S_1', m_1'(\text{loop}_c^{1'}))$ and (s_2, m_2) $\xrightarrow{*} (S_2', m_2'(\text{loop}_c^{2'}))$,
   $\text{loop}_c^{1'}(n_1) < i + 1$ and $\text{loop}_c^{2'}(n_2) < i + 1$;
2. There are two configurations $(s_1, m_{1_{i+1}})$ and $(s_2, m_{2_{i+1}})$ reachable from (s_1, m_1) and (s_2, m_2), respectively, in which the loop counters of s_1 and s_2 are equal to $i + 1$ and value stores agree on the values of imported variables relative to x and, for every state in executions $(s_1, m_1) \xrightarrow{*} (s_1, m_{1_{i+1}})$ and $(s_2, m_2) \xrightarrow{*} (s_2, m_{2_{i+1}})$ the loop counters for s_1 and s_2 are less than or equal to $i + 1$ respectively:
   $\exists (s_1, m_{1_{i+1}}), (s_2, m_{2_{i+1}}) : (s_1, m_1) \xrightarrow{*} (s_1, m_{1_{i+1}}(\text{loop}_c^{1_{i+1}},$ $\sigma_{1_{i+1}})) \wedge (s_2, m_2) \xrightarrow{*} (s_2, m_{2_{i+1}}(\text{loop}_c^{2_{i+1}}, \sigma_{2_{i+1}}))$ where
   - $\text{loop}_c^{1_{i+1}}(n_1) = \text{loop}_c^{2_{i+1}}(n_2) = i + 1$; and
   - $\forall y \in \text{Imp}(x), \sigma_{1_{i+1}}(y) = \sigma_{2_{i+1}}(y)$; and
   - $\forall m_1' : (s_1, m_1) \xrightarrow{*} (S_1', m_1'(\text{loop}_c^{1'})) \xrightarrow{*}$ $(s_1, m_{1_{i+1}}(\text{loop}_c^{1_{i+1}}, \sigma_{1_{i+1}}))$, $\text{loop}_c^{1'}(n_1) \leq i + 1$; and
   - $\forall m_2' : (s_2, m_2) \xrightarrow{*} (S_2', m_2'(\text{loop}_c^{2'})) \xrightarrow{*}$ $(s_2, m_{2_{i+1}}(\text{loop}_c^{2_{i+1}}, \sigma_{2_{i+1}}))$, $\text{loop}_c^{2'}(n_2) \leq i + 1$.

By the hypothesis IH, one of the following holds:

1. The loop counters for s_1 and s_2 are always less than i:
   $\forall m_1', m_2'$ such that $(s_1, m_1) \xrightarrow{*} (S_1', m_1'(\text{loop}_c^{1'})$ and (s_2, m_2) $\xrightarrow{*} (S_2', m_2'(\text{loop}_c^{2'}))$,
   $\text{loop}_c^{1'}(n_1) < i$ and $\text{loop}_c^{2'}(n_2) < i$;
   When this case holds, then we have the loop counters for s_1 and s_2 are always less than $i + 1$:
   $\forall m_1', m_2'$ such that $(s_1, m_1) \xrightarrow{*} (S_1', m_1'(\text{loop}_c^{1'})$ and (s_2, m_2) $\xrightarrow{*} (S_2', m_2'(\text{loop}_c^{2'}))$,
   $\text{loop}_c^{1'}(n_1) < i + 1$ and $\text{loop}_c^{2'}(n_2) < i + 1$.
2. There are two configurations (s_1, m_{1_i}) and (s_2, m_{2_i}) reachable from (s_1, m_1) and (s_2, m_2), respectively, in which the loop counters of s_1 and s_2 are equal to i and value stores agree on the

values of imported variables relative to x and, for every state in executions $(s_1, m_1) \xrightarrow{*} (s_1, m_{1_i})$ and $(s_2, m_2) \xrightarrow{*} (s_2, m_{2_i})$ the loop counters for s_1 and s_2 are less than or equal to i respectively:
$\exists (s_1, m_{1_i}), (s_2, m_{2_i}) : (s_1, m_1) \xrightarrow{*} (s_1, m_{1_i}(\text{loop}_c^{1_i}, \sigma_{1_i})) \wedge$
$(s_2, m_2) \xrightarrow{*} (s_2, m_{2_i}(\text{loop}_c^{2_i}, \sigma_{2_i}))$ where

- $\text{loop}_c^{1_i}(n_1) = \text{loop}_c^{2_i}(n_2) = i$; and
- $\forall y \in \text{Imp}(x), \sigma_{1_i}(y) = \sigma_{2_i}(y)$; and
- $\forall m'_1 : (s_1, m_1) \xrightarrow{*} (S'_1, m'_1(\text{loop}_c^{1'}) \xrightarrow{*} (s_1, m_{1_i}(\text{loop}_c^{1_i}, , \sigma_{1_i}))$, $\text{loop}_c^{1'}(n_1) \le i$; and
- $\forall m'_2 : (s_2, m_2) \xrightarrow{*} (S'_2, m'_2(\text{loop}_c^{2'})) \xrightarrow{*} (s_2, m_{2_i}(\text{loop}_c^{2_i},), \sigma_{2_i}))$, $\text{loop}_c^{2'}(n_2) \le i$.

By similar argument in base case, evaluations of the predicate expression of s_1 and s_2 w.r.t value stores σ_{1_i} and σ_{2_i} produce same value. Then there are two possibilities:

(a) $\mathcal{E}'[\![e]\!]\sigma_{1_i} = \mathcal{E}'[\![e]\!]\sigma_{2_i} = (0, v_{\mathfrak{of}})$
The execution from $(s_1, m_1(\text{loop}_c^1, \sigma_1))$ proceeds as follows.

$(s_1, m_{1_i}(\text{loop}_c^{1_i}, \sigma_{1_i}))$
$= (\text{while}_{\langle n_1 \rangle}(e)\ \{S_1\}, m_{1_i}(\text{loop}_c^{1_i}, \sigma_{1_i}))$
$\rightarrow (\text{while}_{\langle n_1 \rangle}((0, v_{\mathfrak{of}}))\ \{S_1\}, m_{1_i}(\text{loop}_c^{1_i}, \sigma_{1_i}))$ by the EEval' rule
$\rightarrow (\text{while}_{\langle n_1 \rangle}(0)\ \{S_1\}, m_{1_i}(\text{loop}_c^{1_i}, \sigma_{1_i}))$
by the E-Oflow1 and E-Oflow2 rule
$\rightarrow (\text{skip}, m_{1_i}(\text{loop}_c^{1_i}[0/n_1], \sigma_{1_i}))$ by the Wh-F rule.

By the hypothesis IH, the loop counter of s_1 and s_2 in any configuration in executions $(s_1, m_1) \xrightarrow{*} (s_1, m_{1_i}(\text{loop}_c^{1_i}, \sigma_{1_i}))$ and $(s_2, m_2) \xrightarrow{*} (s_2, m_{2_i}(\text{loop}_c^{2_i}, \sigma_{2_i}))$ respectively are less than or equal to i,
$\forall m'_1 : (s_1, m_1) \xrightarrow{*} (S'_1, m'_1(\text{loop}_c^{1'}) \xrightarrow{*} (s_1, m_{1_i}(\text{loop}_c^{1_i}, , \sigma_{1_i}))$, $\text{loop}_c^{1'}(n_1) \le i$; and
$\forall m'_2 : (s_2, m_2) \xrightarrow{*} (S'_2, m'_2(\text{loop}_c^{2'})) \xrightarrow{*} (s_2, m_{2_i}(\text{loop}_c^{2_i},), \sigma_{2_i}))$, $\text{loop}_c^{2'}(n_2) \le i$.
Therefore, the loop counter of s_1 and s_2 in any configuration in executions
$(s_1, m_1) \xrightarrow{*} (\text{skip}, m_{1_i}(\text{loop}_c^{1_i} \setminus \{(n_1)\}, \sigma_{1_i}))$ and
$(s_2, m_2) \xrightarrow{*} (\text{skip}, m_{2_i}(\text{loop}_c^{2_i}[0/n_2], \sigma_{2_i}))$ respectively are less than $i + 1$.

(b) $\mathcal{E}'[\![e]\!]\sigma_{1_i} = \mathcal{E}'[\![e]\!]\sigma_{2_i} = v \ne 0$
The execution from $(s_1, m_{1_i}(\text{loop}_c^{1_i}, \sigma_{1_i}))$ proceeds as follows.

$(s_1, m_{1_i}(\text{loop}_c^{1_i}, \sigma_{1_i}))$
$= (\text{while}_{\langle n_1 \rangle}(e)\ \{S_1\}, m_{1_i}(\text{loop}_c^{1_i}, \sigma_{1_i}))$
$\rightarrow (\text{while}_{\langle n_1 \rangle}(v)\ \{S_1\}, m_{1_i}(\text{loop}_c^{1_i}, \sigma_{1_i}))$ by the EEval' rule
$\rightarrow (S_1; \text{while}_{\langle n_1 \rangle}(e)\ \{S_1\}, m_{1_i}(\text{loop}_c^{1_i}[i+1/(n_1)]$
$, \sigma_{1_i}))$ by the Wh-T rule.

Similarly, $(s_2, m_{2_i}(\text{loop}_c^{2_i}, \sigma_{2_i})) \xrightarrow{2} (S_2; \text{while}_{\langle n_2 \rangle}(e)\{S_2\}, m_{2_i}(\text{loop}_c^{2_i}[i+1/(n_2)], \sigma_{2_i}))$.
By similar argument in base case, the executions of S_1 and S_2 terminate when started in states $m_{1_i}(\text{loop}_c^{1_i}[i + 1/(n_1)], , \sigma_{1_i})$ and $m_{2_i}(\text{loop}_c^{2_i}[i + 1/(n_2)], \sigma_{2_i})$ respectively, $(S_1; s_1, m_{1_i}(\text{loop}_c^{1_i}[i + 1/(n_1)], \sigma_{1_i})) \xrightarrow{*}$
$(s_1, m_{1_{i+1}}(\text{loop}_c^{1_{i+1}}, \sigma_{1_{i+1}}))$ and $(S_2; s_1, m_{2_i}(\text{loop}_c^{2_i}[i + 1/(n_2)], \sigma_{2_i})) \xrightarrow{*} (s_2, m_{2_{i+1}}(\text{loop}_c^{2_{i+1}}, \sigma_{2_{i+1}}))$ such that all of the following holds:

- $\text{loop}_c^{1_{i+1}}(n_1) = \text{loop}_c^{2_{i+1}}(n_2) = i + 1$; and
- $\forall y \in \text{Imp}(x), \sigma_{1_{i+1}}(y) = \sigma_{2_{i+1}}(y)$, and
- in any state in the execution
$(s_1, m_{1_i}) \xrightarrow{*} (s_1, m_{1_{i+1}}(\text{loop}_c^{1_{i+1}}, \sigma_{1_{i+1}}))$, the loop counter of s_1 is less than or equal to $i + 1$.
- in any state in the executions
$(s_2, m_{2_i}) \xrightarrow{*} (s_2, m_{2_{i+1}}(\text{loop}_c^{2_{i+1}}, \sigma_{2_{i+1}}))$, the loop counter of s_2 is less than or equal to $i + 1$.

With the hypothesis IH, there are two configurations $(s_1, m_{1_{i+1}})$ and $(s_2, m_{2_{i+1}})$ reachable from (s_1, m_1) and (s_2, m_2), respectively, in which the loop counters of s_1 and s_2 are equal to $i + 1$ and value stores agree on the values of imported variables relative to x and, for every state in executions $(s_1, m_1) \xrightarrow{*} (s_1, m_{1_{i+1}})$ and $(s_2, m_2) \xrightarrow{*} (s_2, m_{2_{i+1}})$ the loop counters for s_1 and s_2 are less than or equal to $i + 1$ respectively:
$\exists (s_1, m_{1_{i+1}}), (s_2, m_{2_{i+1}}) :$
$(s_1, m_1) \xrightarrow{*} (s_1, m_{1_{i+1}}(\text{loop}_c^{1_{i+1}}, \sigma_{1_{i+1}})) \wedge$
$(s_2, m_2) \xrightarrow{*} (s_2, m_{2_{i+1}}(\text{loop}_c^{2_{i+1}}, \sigma_{2_{i+1}}))$ where

- $\text{loop}_c^{1_{i+1}}(n_1) = \text{loop}_c^{2_{i+1}}(n_2) = i + 1$; and
- $\forall y \in \text{Imp}(x), \sigma_{1_{i+1}}(y) = \sigma_{2_{i+1}}(y)$; and
- $\forall m'_1 : (s_1, m_1) \xrightarrow{*} (S'_1, m'_1(\text{loop}_c^{1'})) \xrightarrow{*} (s_1, m_{1_{i+1}}(\text{loop}_c^{1_{i+1}}, \sigma_{1_{i+1}}))$, $\text{loop}_c^{1'}(n_1) \le i+1$; and
- $\forall m'_2 : (s_2, m_2) \xrightarrow{*} (S'_2, m'_2(\text{loop}_c^{2'})) \xrightarrow{*} (s_2, m_{2_{i+1}}(\text{loop}_c^{2_{i+1}}, \sigma_{2_{i+1}}))$, $\text{loop}_c^{2'}(n_2) \le i + 1$.

□

Lemma 5.2. *Let s_1 = "while$_{\langle n_1 \rangle}(e)\ \{S_1\}$" and s_2 = "while$_{\langle n_2 \rangle}(e)\ \{S_2\}$" be two while statements with the same set of imported variables relative to a variable x (defined in s_1 and s_2), and whose loop bodies S_1 and S_2 terminatingly compute the variables in Imp(x) equivalently when started in states that agree on the values of the variables imported by S_1 or S_2 relative to Imp(x):*

- *$x \in Def(s_1) \cap Def(s_2)$;*
- *$Imp(s_1, \{x\}) = Imp(s_2, \{x\}) = Imp(x)$;*
- *$\forall y \in Imp(x)\ \forall m_{S_1}(\sigma_{S_1})\ m_{S_2}(\sigma_{S_2})$:*
$((\forall z \in Imp(S_1, Imp(x)) \cup Imp(S_2, Imp(x)), \sigma_{S_1}(z) = \sigma_{S_2}(z)) \Rightarrow ((S_1, m_{S_1}(\sigma_{S_1})) \equiv_y (S_2, m_{S_2}(\sigma_{S_2})))$.

If the executions of s_1 and s_2 terminate when started in states $m_1(loop_c^1, \sigma_1)$ and $m_2(loop_c^2, \sigma_2)$ in which s_1 and s_2 have not already executed (loop counter initially 0: $loop_c^1(n_1) = loop_c^2(n_2) = 0$), and whose value stores σ_1 and σ_2 agree on the values of the variables in Imp(x), $\forall y \in Imp(x)\ \sigma_1(y) = \sigma_2(y)$, when s_1 and s_2 terminate, $(s_1, m_1) \xrightarrow{*} (skip, m_{1_i}(\sigma'_1))$ and $(s_2, m_2) \xrightarrow{*} (skip, m_{2_i}(\sigma'_2))$, value stores σ'_1 and σ'_2 agree on the value of x, $\sigma'_1(x) = \sigma'_2(x)$.

Proof. We show that there must exist a finite integer k such that the loop counters of s_1 and s_2 in executions started in states m_1 and m_2 is always less than k. By the definition of terminating execution, there are only finite number of steps in executions of s_1 and s_2 started in states m_1 and m_2 respectively. Then, by Lemma E.8, there must be a finite integer k such that the loop counter of s_1 and s_2 is always less than k. In the following, we consider k be the smallest positive integer such that the loop counter of s_1 and s_2 in executions started in states m_1 and m_2 is always less than k.

By Lemma 5.1, there are two possibilities:

1. The loop counters for s_1 and s_2 are always less than 1 ($k = 1$): $\forall m'_1, m'_2$ such that $(s_1, m_1) \xrightarrow{*} (S'_1, m'_1(\text{loop}_c^{1'}))$ and $(s_2, m_2) \xrightarrow{*} (S'_2, m'_2(\text{loop}_c^{2'}))$,
$\text{loop}_c^{1'}(n_1) < 1$ and $\text{loop}_c^{2'}(n_2) < 1$;
By the proof in base case of Lemma 5.1, the execution of s_1 proceeds as follows:

(s_1, m_1)
$= (\text{while}_{\langle n_1 \rangle}(e)\ \{S_1\}, m_1(\text{loop}_c^1, \sigma_1))$
$\rightarrow(\text{while}_{\langle n_1 \rangle}(0)\ \{S_1\}, m_1(\text{loop}_c^1, \sigma_1))$ by the EEval' rule
$\rightarrow(\text{skip}, m_1(\text{loop}_c^1[0/(n_1)], \sigma_1))$ by the Wh-F rule.

Similarly, the execution of s_2 proceeds to $(\text{skip}, m_2(\text{loop}_c^2[0/n_2], \sigma_2))$. Therefore, $\sigma_1' = \sigma_1$ and $\sigma_2' = \sigma_2$.
By the definition of imported variables, $x \in \text{Imp}(s_1, \{x\})$. By assumption, value stores σ_1 and σ_2 agree on the value of x, $\sigma_1'(x) = \sigma_1(x) = \sigma_2(x) = \sigma_2'(x)$. The lemma holds.

2. For some finite positive $k(> 1)$, both of the following hold:
   - The loop counters for s_1 and s_2 are always less than k: $\forall m_1', m_2'$ such that $(s_1, m_1) \xrightarrow{*} (S_1', m_1'(\text{loop}_c^{1'})$ and $(s_2, m_2) \xrightarrow{*} (S_2', m_2'(\text{loop}_c^{2'}))$, $\text{loop}_c^{1'}(n_1) < k$ and $\text{loop}_c^{2'}(n_2) < k$;
   - There are two configuration $(s_1, m_{1_{k-1}})$ and $(s_2, m_{2_{k-1}})$ reachable from (s_1, m_1) and (s_2, m_2), respectively, in which the loop counters of s_1 and s_2 are equal to $k-1$ and value stores agree on the values of imported variables relative to x and, for every state in execution, $(s_1, m_1) \xrightarrow{*} (s_1, m_{1_{k-1}})$ or $(s_2, m_2) \xrightarrow{*} (s_2, m_{2_{k-1}})$ the loop counters for s_1 and s_2 are less than or equal to $k-1$ respectively:
     $\exists (s_1, m_{1_{k-1}}), (s_2, m_{2_{k-1}})$:
     $(s_1, m_1) \xrightarrow{*} (s_1, m_{1_{k-1}}(\text{loop}_c^{1_{k-1}}, \sigma_{1_{k-1}})) \wedge$
     $(s_2, m_2) \xrightarrow{*} (s_2, m_{2_{k-1}}(\text{loop}_c^{2_{k-1}}, \sigma_{2_{k-1}}))$ where
     - $\text{loop}_c^{1_{k-1}}(n_1) = \text{loop}_c^{2_{k-1}}(n_2) = k-1$; and
     - $\forall y \in \text{Imp}(x) : \sigma_{1_{k-1}}(y) = \sigma_{2_{k-1}}(y)$; and
     - $\forall m_1' : (s_1, m_1) \xrightarrow{*} (S_1', m_1'(\text{loop}_c^{1'}) \xrightarrow{*} (s_1, m_{1_{k-1}}(\text{loop}_c^{1_{k-1}}, \sigma_{1_{k-1}}))$, $\text{loop}_c^{1'}(n_1) \le k-1$; and
     - $\forall m_2' : (s_2, m_2) \xrightarrow{*} (S_2', m_2'(\text{loop}_c^{2'})) \xrightarrow{*} (s_2, m_{2_{k-1}}(\text{loop}_c^{2_{k-1}}, \sigma_{2_{k-1}}))$, $\text{loop}_c^{2'}(n_2) \le k-1$;

   By proof of Lemma 5.1, value stores $\sigma_{1_{k-1}}$ and $\sigma_{2_{k-1}}$ agree on the values of the variables in Use(e). By Lemma D.1, $\mathcal{E}'[\![e]\!]\sigma_{1_{k-1}} = \mathcal{E}'[\![e]\!]\sigma_{2_{k-1}} = v$. Because the loop counter of s_1 and s_2 is less than k in executions of s_1 and s_2 when started in states m_1 and m_2, then by our semantic rules, the predicate expression of s_1 and s_2 must evaluate to zero w.r.t value stores $\sigma_{1_{k-1}}$ and $\sigma_{2_{k-1}}$, $\mathcal{E}'[\![e]\!]\sigma_{1_{k-1}} = \mathcal{E}'[\![e]\!]\sigma_{2_{k-1}} = (0, v_{\mathtt{of}})$. Then the execution of s_1 proceeds as follows.

   $(\text{while}_{\langle n_1 \rangle}(e)\ \{S_1\}, m_{1_{k-1}}(\text{loop}_c^{1_{k-1}}, \sigma_{1_{k-1}}))$
   $\rightarrow(\text{while}_{\langle n_1 \rangle}((0, v_{\mathtt{of}}))\ \{S_1\}, m_{1_{k-1}}(\text{loop}_c^{1_{k-1}}, \sigma_{1_{k-1}}))$
   by the EEval' rule
   $\rightarrow(\text{while}_{\langle n_1 \rangle}(0)\ \{S_1\}, m_{1_{k-1}}(\text{loop}_c^{1_{k-1}}, \sigma_{1_{k-1}}))$
   by the E-Oflow1 or E-Oflow2 rule
   $\rightarrow(\text{skip}, m_{1_{k-1}}(\text{loop}_c^{1_{k-1}}[0/n_1], \sigma_{1_{k-1}}))$
   by the Wh-F rule.

   Similarly, the execution of s_2 proceeds to $(\text{skip}, m_{2_{k-1}}(\text{loop}_c^{2_{k-1}}[0/n_2], \sigma_{2_{k-1}}))$. Therefore, $\sigma_2' = \sigma_{2_{k-1}}$, $\sigma_1' = \sigma_{1_{k-1}}$. By the definition of imported variables, $x \in \text{Imp}(x)$. In conclusion, $\sigma_2'(x) = \sigma_{2_{k-1}}(x) = \sigma_{1_{k-1}}(x) = \sigma_1'(x)$.

□

Lemma 5.3. *If two statement sequences S_1 and S_2 satisfy the proof rule of terminating computation of a variable x equivalently, then S_1 and S_2 have same imported variables relative to x: $(S_1 \equiv_x^S S_2) \Rightarrow (\mathit{Imp}(S_1, \{x\}) = \mathit{Imp}(S_2, \{x\}))$.*

Proof. By induction on size(S_1)+size(S_2), the sum of the program size of S_1 and S_2.
Base case.
S_1 and S_2 are simple statement. Then the proof is a case analysis according to the cases in the definition of the proof rule of computation equivalently for simple statements.

1. $S_1 = S_2$
   By the definition of imported variables, same statement have same imported variables relative to same x.
2. $S_1 \neq S_2$
   There are two further cases:
   - $S_1 =$ "input id_1", $S_2 =$ "input id_2" and $x \notin \{id_1, id_2\}$.
     When $x = id_I$, by the definition of imported variables, $\text{Imp}(S_1, \{id_I\}) = \text{Imp}(S_2, \{id_I\}) = \{id_I\}$. When $x = id_{IO}$, by the definition of imported variables, $\text{Imp}(S_1, \{id_{IO}\}) = \text{Imp}(S_2, \{id_{IO}\}) = \{id_I, id_{IO}\}$.
     When $x \notin \{id_I, id_{IO}\}$, by the definition of imported variables, $\text{Imp}(S_1, \{x\}) = \text{Imp}(S_2, \{x\}) = \{x\}$.
   - the above cases do not hold and $x \notin \text{Def}(S_1) \cup \text{Def}(S_2)$.
     By the definition of imported variables, $\text{Imp}(S_1, \{x\}) = \text{Imp}(S_2, \{x\}) = \{x\}$.

Induction Step.
The hypothesis IH is that the lemma holds when size(S_1) + size$(S_2) = k \ge 2$.
Then we show the lemma holds when size(S_1)+size$(S_2) = k+1$. The proof is a case analysis based on the cases in the definition of the proof rule of computation equivalently for statement sequence, $S_1 \equiv_x^S S_2$:

1. S_1 and S_2 are one statement such that one of the following holds:
   (a) S_1 and S_2 are If statement that define the variable x:
   $S_1 =$ "If (e) then $\{S_1^t\}$else $\{S_1^f\}$", $S_2 =$ "If (e) then $\{S_2^t\}$ else $\{S_2^f\}$" such that all of the following hold:
   - $x \in \text{Def}(S_1) \cap \text{Def}(S_2)$;
   - $S_1^t \equiv_x^S S_2^t$;
   - $S_1^f \equiv_x^S S_2^f$;

   By the hypothesis IH, the imported variables in S_1^t and S_2^t relative to x are same, $\text{Imp}(S_1^t, \{x\}) = \text{Imp}(S_2^t, \{x\})$. Similarly, $\text{Imp}(S_1^f, \{x\}) = \text{Imp}(S_2^f, \{x\})$. By the definition of imported variables, $\text{Imp}(S_1, \{x\}) = \text{Use}(e) \cup \text{Imp}(S_1^t, \{x\}) \cup \text{Imp}(S_1^f, \{x\})$. Similarly, $\text{Imp}(S_2, \{x\}) = \text{Use}(e) \cup \text{Imp}(S_2^t, \{x\}) \cup \text{Imp}(S_2^f, \{x\})$. Then, the lemma holds.

   (b) S_1 and S_2 are while statement that define the variable x:
   $S_1 =$ "while$_{\langle n_1 \rangle}(e)\ \{S_1''\}$", $S_2 =$ "while$_{\langle n_2 \rangle}(e)\ \{S_2''\}$" such that both of the following hold:
   - $x \in \text{Def}(S_1) \cap \text{Def}(S_2)$;
   - $\forall y \in \text{Imp}(S_1, \{x\}) \cup \text{Imp}(S_2, \{x\}), S_1'' \equiv_y^S S_2''$;

   By the definition of imported variables, $\text{Imp}(S_1, \{x\}) = \bigcup_{i \ge 0} \text{Imp}({S_1''}^i, \{x\} \cup \text{Use}(e))$. Similarly, $\text{Imp}(S_2, \{x\}) = \bigcup_{i \ge 0} \text{Imp}({S_2''}^i, \{x\} \cup \text{Use}(e))$. Then, we show that $\text{Imp}({S_1''}^i, \{x\} \cup \text{Use}(e)) = \text{Imp}({S_2''}^i, \{x\} \cup \text{Use}(e))$ by induction on i.
   Base case.
   By our assumption of the notation S^0, ${S_1''}^0$ = skip, ${S_2''}^0$ = skip.
   Then, $\text{Imp}({S_1''}^0, \{x\} \cup \text{Use}(e))$ = $\{x\} \cup \text{Use}(e)$, $\text{Imp}({S_2''}^0, \{x\} \cup \text{Use}(e))$ = $\{x\} \cup \text{Use}(e)$.
   Hence, $\text{Imp}({S_1''}^0, \{x\} \cup \text{Use}(e))$ = $\text{Imp}({S_2''}^0, \{x\} \cup \text{Use}(e))$.
   Induction step.

The hypothesis IH2 is that $\text{Imp}(S_1^{''i}, \{x\} \cup \text{Use}(e)) = \text{Imp}(S_2^{''i}, \{x\} \cup \text{Use}(e))$ for $i \geq 0$.
Then we show that $\text{Imp}(S_1^{''i+1}, \{x\} \cup \text{Use}(e)) = \text{Imp}(S_2^{''i+1}, \{x\} \cup \text{Use}(e))$.

$$\begin{aligned} & \text{Imp}(S_1^{''i+1}, \{x\} \cup \text{Use}(e)) \\ = \ & \text{Imp}(S_1^{''}, \text{Imp}(S_1^{''i}, \{x\} \cup \text{Use}(e))) \text{ (1) by corollary C.1} \end{aligned}$$

$$\begin{aligned} & \text{Imp}(S_2^{''i+1}, \{x\} \cup \text{Use}(e)) \\ = \ & \text{Imp}(S_2^{''}, \text{Imp}(S_2^{''i}, \{x\} \cup \text{Use}(e))) \text{ (2) by corollary C.1} \end{aligned}$$

$\text{Imp}(S_1^{''i}, \{x\} \cup \text{Use}(e)) = \text{Imp}(S_2^{''i}, \{x\} \cup \text{Use}(e))$
by the hypothesis IH2
$\text{Imp}(S_1^{''}, \text{Imp}(S_1^{''i}, \{x\} \cup \text{Use}(e)))$
$= \text{Imp}(S_2^{''}, \text{Imp}(S_2^{''i}, \{x\} \cup \text{Use}(e)))$
by the hypothesis IH
$\text{Imp}(S_1^{''i+1}, \{x\} \cup \text{Use}(e))$
$= \text{Imp}(S_2^{''i+1}, \{x\} \cup \text{Use}(e))$ by (1),(2)
Therefore, $\text{Imp}(S_1^{''i+1}, \{x\} \cup \text{Use}(e)) = \text{Imp}(S_2^{''i+1}, \{x\} \cup \text{Use}(e))$.
In conclusion, $\text{Imp}(S_1, \{x\}) = \text{Imp}(S_2, \{x\})$. The lemma holds.

(c) S_1 and S_2 do not define the variable x: $x \notin \text{Def}(S_1) \cup \text{Def}(S_2)$.
By the definition of imported variable, the imported variables in S_1 and S_2 relative to x is x, $\text{Imp}(S_1, \{x\}) = \text{Imp}(S_2, \{x\}) = \{x\}$. The lemma holds.

2. S_1 and S_2 are not both one statement such that one of the following holds:
   (a) Last statements both define the variable x such that all of the following hold:
   - $\forall y \in \text{Imp}(s_1, \{x\}) \cup \text{Imp}(s_2, \{x\}), S_1' \equiv_y^S S_2'$;
   - $x \in \text{Def}(s_1) \cap \text{Def}(s_2)$;
   - $s_1 \equiv_x^S s_2$;

   By the hypothesis IH, we have $\text{Imp}(s_1, \{x\}) = \text{Imp}(s_2, \{x\}) = \text{Imp}(\Delta)$. Then, by the hypothesis IH again, we have that $\forall y \in \text{Imp}(\Delta) = \text{Imp}(s_1, \{x\}) = \text{Imp}(s_2, \{x\}), \text{Imp}(S_1', \{y\}) = \text{Imp}(S_2', \{y\})$. By taking the union of all $\forall y \in \text{Imp}(\Delta)$, $\text{Imp}(S_1', \{y\})$ and $\text{Imp}(S_2', \{y\})$, by the Lemma C.2, $\text{Imp}(S_1', \text{Imp}(\Delta)) = \text{Imp}(S_2', \text{Imp}(\Delta))$. By the definition of imported variables,
   $\text{Imp}(S_1, \{x\}) = \text{Imp}(S_1', \text{Imp}(s_1, \{x\}))$, $\text{Imp}(S_2, \{x\}) = \text{Imp}(S_2', \text{Imp}(s_2, \{x\}))$. Therefore, the lemma holds.

   (b) One last statement does not define the variable x: w.l.o.g., $\big(x \notin \text{Def}(s_1) \wedge (S_1' \equiv_x^S S_2)\big)$;
   By the definition of imported variables, we have $\text{Imp}(s_1, \{x\}) = \{x\}$. By the hypothesis IH, $\text{Imp}(S_1', \{x\}) = \text{Imp}(S_2, \{x\})$
   Therefore, by the definition of imported variables, $\text{Imp}(S_1, \{x\}) = \text{Imp}(S_1', \text{Imp}(s_1, \{x\})) = \text{Imp}(S_1', \{x\}) = \text{Imp}(S_2, \{x\})$.

   (c) There are statements moving in/out of If statement:
   $s_1 =$ "If (e) then $\{S_1^t\}$ else $\{S_1^f\}$", $s_2 =$ "If (e) then $\{S_2^t\}$ else $\{S_2^f\}$" such that none of the above cases hold and all of the following hold:
   - $\forall y \in \text{Use}(e), S_1' \equiv_y^S S_2'$;
   - $S_1'; S_1^t \equiv_x^S S_2'; S_2^t$;
   - $S_1'; S_1^f \equiv_x^S S_2'; S_2^f$;
   - $x \in \text{Def}(s_1) \cap \text{Def}(s_2)$;

   We show all of the following hold.
   i. $\text{Imp}(S_1', \text{Use}(e)) = \text{Imp}(S_2', \text{Use}(e))$.
   By the hypothesis IH and the assumption that $\forall y \in \text{Use}(e), S_1' \equiv_y^S S_2'$. Then, by Lemma C.2,
   $\text{Imp}(S_1', \text{Use}(e)) = \text{Imp}(S_2', \text{Use}(e))$.
   ii. $\text{Imp}(S_1', \text{Imp}(S_1^t, \{x\})) = \text{Imp}(S_2', \text{Imp}(S_2^t, \{x\}))$.
   Because size("If(e) then $\{S_t\}$ else $\{S_f\}$") $= 1 +$ size(S_t) + size(S_f), then
   size$(S_1'; S_1^t)$ + size$(S_2'; S_2^t) < k$. By the hypothesis IH,
   $\text{Imp}(S_1'; S_1^t, \{x\}) = \text{Imp}(S_2'; S_2^t, \{x\})$.
   Besides, by Lemma C.1,
   $\text{Imp}(S_1', \text{Imp}(S_1^t, \{x\})) = \text{Imp}(S_1'; S_1^t, \{x\})$
   $= \text{Imp}(S_2'; S_2^t, \{x\}) = \text{Imp}(S_2', \text{Imp}(S_2^t, \{x\}))$.
   iii. $\text{Imp}(S_1', \text{Imp}(S_1^f, \{x\})) = \text{Imp}(S_2', \text{Imp}(S_2^f, \{x\}))$.
   By similar argument in the case that
   $\text{Imp}(S_1', \text{Imp}(S_1^t, \{x\})) = \text{Imp}(S_2', \text{Imp}(S_2^t, \{x\}))$.

   Then, by combining things together,

$$\begin{aligned} & \text{Imp}(S_1, \{x\}) \\ = \ & \text{Imp}(S_1', \text{Imp}(s_1, \{x\})) \\ & \text{by the definition of imported variables} \\ = \ & \text{Imp}(S_1', \text{Imp}(S_1^t, \{x\}) \cup \text{Imp}(S_1^f, \{x\}) \cup \text{Use}(e)) \\ & \text{by the definition of imported variables} \\ = \ & \text{Imp}(S_1', \text{Imp}(S_1^t, \{x\})) \cup \text{Imp}(S_1', \text{Imp}(S_1^f, \{x\})) \\ & \cup \text{Imp}(S_1', \text{Use}(e)) \text{ by Lemma C.2} \\ = \ & \text{Imp}(S_2', \text{Imp}(S_2^t, \{x\})) \cup \text{Imp}(S_2', \text{Imp}(S_2^f, \{x\})) \\ & \cup \text{Imp}(S_2', \text{Use}(e)) \text{ by i, ii, iii} \\ = \ & \text{Imp}(S_2', \text{Imp}(S_2^t, \{x\}) \cup \text{Imp}(S_2^f, \{x\}) \cup \text{Use}(e)) \\ & \text{by Lemma C.2} \\ = \ & \text{Imp}(S_2', \text{Imp}(S_2^t, \{x\}) \cup \text{Imp}(S_2^f, \{x\}) \cup \text{Use}(e)) \\ & \text{by the definition of imported variables} \\ = \ & \text{Imp}(S_2', \text{Imp}(s_2, \{x\})) \\ & \text{by the definition of imported variables} \\ = \ & \text{Imp}(S_2, \{x\}). \end{aligned}$$

   Hence, the lemma holds.

□

5.3 Termination in the same way

We proceed to propose a proof rule under which two statement sequences either both terminate or both do not terminate. We start by giving the definition of termination in the same way. Then we present the proof rule of termination in the same way. Our proof rule of termination in the same way allows updates such as statement duplication or reordering, loop fission or fusion and additional terminating statements. We prove that the proof rule ensures terminating in the same way by induction on the program size of the two programs in the proof rule. We also list auxiliary lemmas required by the proof of termination in the same way.

Definition 14. (Termination in the same way) *Two statement sequences S_1 and S_2 terminate in the same way when started in states m_1 and m_2 respectively, written $(S_1, m_1) \equiv_H (S_2, m_2)$, iff one of the following holds:*

1. *$(S_1, m_1) \xrightarrow{*} (skip, m_1')$ and $(S_2, m_2) \xrightarrow{*} (skip, m_2')$;*
2. *$\forall i \geq 0$, $(S_1, m_1) \xrightarrow{i} (S_1^i, m_1^i)$ and $(S_2, m_2) \xrightarrow{i} (S_2^i, m_2^i)$ where $S_1^i \neq skip, S_2^i \neq skip$.*

5.3.1 Proof rule for termination in the same way

We define proof rules under which two statement sequences S_1 and S_2 terminate in the same way. We summarize the cause of nonterminating execution and then give the proof rule.

We consider two causes of nonterminating executions: crash and infinite iterations of loop statements. As to crash [**?**], we consider four common causes based on our language: expression evaluation exceptions, the lack of input value, input/assignment value type mismatch and array index out of bound. In essence, the causes of nontermination are partly due to the values of some particular variables during executions. We capture variables affecting each source of nontermination; loop deciding variables LVar(S) are variables affecting the evaluation of a loop statements in the statement sequence S, crash deciding variables CVar(S) are variables whose values decide if a crash occurs in S. We list the definitions of LVar(S) and CVar(S) in Definition 15 and 16. Definition 17 summarizes the variables whose values decide if one program terminates.

Definition 15. (Loop deciding variables) *The loop deciding variables of a statement sequence S, written LVar(S), are defined as follows:*

1. *LVar(S) $= \emptyset$ if $\nexists s =$ "while(e) $\{S'\}$" and $s \in S$;*
2. *LVar("If (e) then $\{S_t\}$ else $\{S_f\}$") $=$ Use(e) $\cup$ LVar(S_t) $\cup$ LVar(S_f) if "while(e)$\{S'\}$" $\in S$;*
3. *LVar("while(e)$\{S'\}$") $=$ Imp(S, Use(e) $\cup$ LVar(S'));*
4. *For $k > 0$, LVar($s_1; ...; s_k; s_{k+1}$) $=$ LVar($s_1; ...; s_k$) $\cup$ Imp($s_1; ...; s_k$, LVar(s_{k+1}));*

Definition 16. (Crash deciding variables) *The crash deciding variables of a statement sequence S, written CVar(S), are defined as follows:*

1. *CVar(skip) $= \emptyset$;*
2. *CVar($lval := e$) $=$ Idx($lval$) $\cup$ Use(e) if $(\Gamma \vdash lval :$ Int$) \wedge (\Gamma \vdash e :$ Long$)$;*
3. *CVar($lval := e$) $=$ Idx($lval$)$\cup$Err(e) if $(\Gamma \vdash lval :$ Int$)\wedge(\Gamma \vdash e :$ Long$)$ does not hold;*
4. *CVar(input id) $= \{id_I\}$;*
5. *CVar(output e) $=$ Err(e);*
6. *CVar("If (e) then $\{S_t\}$ else $\{S_f\}$") $=$ Err(e), if CVar(S_t) $\cup$ CVar(S_f) $= \emptyset$;*
7. *CVar("If (e) then $\{S_t\}$ else $\{S_f\}$") $=$ Use(e) $\cup$ CVar(S_t) $\cup$ CVar(S_f), if CVar(S_t) $\cup$ CVar(S_f) $\neq \emptyset$;*
8. *CVar("while$_{\langle n\rangle}$(e)$\{S'\}$") $=$ Imp("while$_{\langle n\rangle}$(e)$\{S'\}$", Use(e) $\cup$ CVar(S'));*
9. *For $k > 0$, CVar($s_1; ...; s_{k+1}$) $=$ CVar($s_1; ...; s_k$) $\cup$ Imp($s_1; ...; s_k$, CVar(s_{k+1}));*

Definition 17. (Termination deciding variables) *The termination deciding variables of statement sequence S are CVar(S)$\cup$LVar(S), written TVar(S).*

Definition 18. (Base cases of the proof rule of termination in the same way) *Two simple statements s_1 and s_2 satisfy the proof rule of termination in the same way, written $s_1 \equiv_H^S s_2$, iff one of the following holds:*

1. *s_1 and s_2 are same, $s_1 = s_2$;*
2. *s_1 and s_2 are input statement with same type variable: $s_1 =$ "input id_1", $s_2 =$ "input id_2" where $(\Gamma_{s_1} \vdash id_1 : \tau) \wedge (\Gamma_{s_2} \vdash id_2 : \tau)$;*
3. *$s_1 =$ "output e" or "$id_1 := e$", $s_2 =$ "output e" or "$id_2 := e$" where both of the following hold:*
   - *There is no possible value mismatch in "$id_1 := e$", $\neg(\Gamma_{s_1} \vdash id_1 :$ Int$) \vee \neg(\Gamma_{s_1} \vdash e :$ Long$) \vee (\Gamma_{s_1} \vdash e :$ Int$)$.*
   - *There is no possible value mismatch in "$id_2 := e$", $\neg(\Gamma_{s_2} \vdash id_2 :$ Int$) \vee \neg(\Gamma_{s_2} \vdash e :$ Long$) \vee (\Gamma_{s_2} \vdash e :$ Int$)$.*

Definition 19. (proof rule of termination in the same way) *Two statement sequences S_1 and S_2 satisfy the proof rule of termination in the same way, written $S_1 \equiv_H^S S_2$, iff one of the following holds:*

1. *S_1 and S_2 are both one statement and one of the following holds.*
   (a) *S_1 and S_2 are simple statements: $s_1 \equiv_H^S s_2$;*
   (b) *S_1 = "If(e) then $\{S_1^t\}$ else $\{S_1^f\}$", S_2 = "If(e) then $\{S_2^t\}$ else $\{S_2^f\}$" and one of the following holds:*
      i. *$S_1^t, S_1^f, S_2^t, S_2^f$ are all sequences of "skip";*
      ii. *At least one of $S_1^t, S_1^f, S_2^t, S_2^f$ is not a sequence of "skip" such that: $(S_1^t \equiv_H^S S_2^t) \wedge (S_1^f \equiv_H^S S_2^f)$;*
   (c) *S_1 = "while$_{\langle n_1\rangle}$(e)$\{S_1''\}$", S_2 = "while$_{\langle n_2\rangle}$(e)$\{S_2''\}$" and both of the following hold:*
      - *$S_1'' \equiv_H^S S_2''$;*
      - *S_1'' and S_2'' have equivalent computation of TVar(S_1) $\cup$ TVar(S_2);*
2. *S_1 and S_2 are not both one statement and one of the following holds:*
   (a) *$S_1 = S_1'; s_1$ and $S_2 = S_2'; s_2$ and all of the following hold:*
      - *$S_1' \equiv_H^S S_2'$;*
      - *S_1' and S_2' have equivalent computation of TVar(s_1) $\cup$ TVar(s_2);*
      - *$s_1 \equiv_H^S s_2$ where s_1 and s_2 are not "skip";*
   (b) *One last statement is "skip": $\big((S_1 = S_1';$ "skip"$)\wedge(S_1' \equiv_H^S S_2)\big) \vee \big((S_2 = S_2';$ "skip"$)\wedge (S_1 \equiv_H^S S_2')\big)$.*
   (c) *One last statement is a "duplicate" statement and one of the following holds:*
      i. *$S_1 = S_1'; s_1'; S_1''; s_1$ and all of the following hold:*
         - *$S_1'; s_1'; S_1'' \equiv_H^S S_2$;*
         - *$(s_1' \equiv_H^S s_1) \wedge (s_1 \neq$ "skip"$)$;*
         - *Def($s_1'; S_1''$) $\cap$ TVar(s_1) $= \emptyset$;*
      ii. *$S_2 = S_2'; s_2'; S_2''; s_2$ and all of the following hold:*
         - *$S_1 \equiv_H^S S_2'; s_2'; S_2''$;*
         - *$(s_2' \equiv_H^S s_2) \wedge (s_2 \neq$ "skip"$)$;*
         - *Def($s_2'; S_2''$) $\cap$ TVar(s_2) $= \emptyset$;*
   (d) *$S_1 = S_1'; s_1; s_1'$ and $S_2 = S_2'; s_2; s_2'$ where s_1 and s_2 are reordered and all of the following hold:*
      - *$S_1' \equiv_H^S S_2'$;*
      - *S_1' and S_2' have equivalent computation of TVar($s_1; s_1'$)$\cup$ TVar($s_2; s_2'$).*
      - *$s_1 \equiv_H^S s_2'$;*
      - *$s_1' \equiv_H^S s_2$;*
      - *Def(s_1) $\cap$ TVar(s_1') $= \emptyset$;*
      - *Def(s_2) $\cap$ TVar(s_2') $= \emptyset$;*

5.3.2 Soundness of the proof rule for termination in the same way

We show that two statement sequences satisfy the proof rules of termination in the same way, and their initial states agree on the values of their termination deciding variables, then they either both terminate or both do not terminate.

Theorem 3. *If two simple statements s_1 and s_2 satisfy the proof rule of termination in the same way, $s_1 \equiv_H^s s_2$, and their initial states $m_1(\mathfrak{f}_1, \sigma_1)$ and $m_2(\mathfrak{f}_2, \sigma_2)$ with crash flags not set, $\mathfrak{f}_1 = \mathfrak{f}_2 = 0$, and whose value stores agree on values of the termination deciding variables of s_1 and s_2, $\forall x \in$ TVar(s_1) $\cup$ TVar(s_2) : $\sigma_1(x) = \sigma_2(x)$, when executions of s_1 and s_2 start in states m_1 and m_2 respectively, then s_1 and s_2 terminate in the same way when started in states m_1 and m_2 respectively: $(s_1, m_1) \equiv_H (s_2, m_2)$.*

Proof. The proof is a case analysis of those cases in the definition of $s_1 \equiv_H^s s_2$. Because s_1 is a simple statement and s_1's execution is without function call, we only care the crash variables of s_1 in the termination deciding variables of s_1, CVar(s_1). Similarly, we only care CVar(s_2).

First s_1 and s_2 are same: $s_1 = s_2$;

We show the theorem by induction on abstract syntax of s_1 and s_2.

1. $s_1 = s_2 =$ skip.
   By definition of termination in the same way, both s_1 and s_2 terminate. The theorem holds.
2. $s_1 = s_2 =$ "$lval := e$".
   There are further cases regarding what $lval$ is.
   (a) $lval = id$.
   By definition, CVar(s_1) = CVar(s_2) = Err(e) or Use(e) based on if there is possible value mismatch (e.g., assigning value defined only in type Long to a variable of type Int). There are two subcases.
   - Left value id is of type Int and expression e is of type Long but not type Int, $(\Gamma \vdash id : \text{Int}) \wedge (\Gamma \vdash e : \text{Long}) \wedge \neg(\Gamma \vdash e : \text{Int})$.
     By definition, CVar(s_1) = CVar(s_2) = Use(e). By assumption, $\forall x \in \text{Use}(e), \sigma_1(x) = \sigma_2(x)$. By Lemma D.1, the expression evaluates to the same value w.r.t two pairs of value stores σ_1 and σ_2 respectively,
     - Both evaluations of expression lead to crash, $\mathcal{E}[\![e]\!]\sigma_1 = \mathcal{E}[\![e]\!]\sigma_2 = (\text{error}, v_{\mathfrak{of}})$.
       Then the execution of s_1 is as follows:
       (s_1, m_1)
       $= (id := e, m_1(\sigma_1))$
       $\rightarrow (id := (\text{error}, *), m_1(\sigma_1))$ by rule EEval'
       $\rightarrow (id := 0, m_1(1/\mathfrak{f}))$ by rule ECrash.
       $\xrightarrow{i} (id := 0, m_1(1/\mathfrak{f}))$ for any $i > 0$ by rule Crash.
       Similarly, s_2 does not terminate. The theorem holds.
     - Both evaluations of expression lead to no crash, $\mathcal{E}[\![e]\!]\sigma_1 = \mathcal{E}[\![e]\!]\sigma_2 = (v, v_{\mathfrak{of}})$.
       Then there are cases regarding if value mismatch occurs.
       $\surd$ The value v is only defined in type Long, $(\Gamma \vdash v : \text{Long}) \wedge \neg(\Gamma \vdash v : \text{Int})$.
       The execution of s_1 is as follows:
       (s_1, m_1)
       $= (id := e, m_1(\sigma_1))$
       $\rightarrow (id := (v, v_{\mathfrak{of}}), m_1(\sigma_1))$ by rule EEval
       $\rightarrow (id := v, m_1(\sigma_1))$ by rule EOflow-1 or EOflow-2.
       $\rightarrow (id := v, m_1(1/\mathfrak{f}))$ by rule Assign-Err.
       $\xrightarrow{i} (id := v, m_1(1/\mathfrak{f}))$ for any $i > 0$ by rule Crash.
       Similarly, s_2 does not terminate. The theorem holds.
       $\surd$ The value v is defined in type Int, $\Gamma \vdash v : \text{Int}$.
       Assuming that the variable id is a global one, the execution of s_1 is as follows:
       (s_1, m_1)
       $= (id := e, m_1(\sigma_1))$
       $\rightarrow (id := (v, v_{\mathfrak{of}}), m_1(\sigma_1))$ by rule EEval
       $\rightarrow (id := v, m_1(\sigma_1))$ by rule EOflow-1 or EOflow-2.
       $\rightarrow (\text{skip}, m_1(\sigma_1(\sigma_1[v/id])))$ by rule Assign.
       Similarly, s_2 terminate. The theorem holds.
       When the variable id is a local variable, by similar argument for the global variable, we can show that s_1 and s_2 terminate. Then the theorem holds.
   - It is not the case that left value id is of type Int and the expression e is of type Long only, $\neg\big((\Gamma \vdash id : \text{Int}) \wedge (\Gamma \vdash e : \text{Long}) \wedge \neg(\Gamma \vdash e : \text{Int})\big)$.
     There are two cases based on if there is crash in evaluation of expression e.
     $\surd$ Both evaluations of expression lead to crash, $\mathcal{E}[\![e]\!]\sigma_1 = \mathcal{E}[\![e]\!]\sigma_2 = (\text{error}, v_{\mathfrak{of}})$.
     By the same argument in case where left value id is of type Int and the expression e is of type Long only, this theorem holds.
     $\surd$ Both evaluations of expression lead to no crash, $\mathcal{E}[\![e]\!]\sigma_1 = \mathcal{E}[\![e]\!]\sigma_2 = (v, v_{\mathfrak{of}})$.
     By the same argument in subcase of no value mismatch in case where left value id is of type Int and the expression e is of type Long only, this theorem holds.

   (b) $lval = id[n]$.
   There are two subcases based on if n is within the array bound of id. By our assumption, array variable id is of the same bound in two programs. W.l.o.g., we assume id is local variable.
   i. n is out of bound of array variable id, $((id, n) \mapsto v_1) \notin \sigma_1$ and $((id, n) \mapsto v_2) \notin \sigma_2$;
   Then the execution of s_1 continues as follows:
   (s_1, m_1)
   $= (id[n] := e, m_1(\sigma_1))$
   $\rightarrow (id[n] := e, m_1(1/\mathfrak{f})$ by rule Arr-3
   $\xrightarrow{i} (id[n] := e, m_1(1/\mathfrak{f}))$ by rule Crash.
   Similarly, s_2 does not terminate. The theorem holds.
   ii. n is within the bound of array variable id, $((id, n) \mapsto v_1) \in \sigma_1$ and $((id, n) \mapsto v_2) \in \sigma_2$;
   There are cases of CVar(s_1) and CVar(s_2) based on if there is possible value mismatch exception in s_1 and s_2.
   - Left value $id[n]$ is of type Int and expression e is of type Long but not type Int, $(\Gamma \vdash id[n] : \text{Int}) \wedge (\Gamma \vdash e : \text{Long}) \wedge \neg(\Gamma \vdash e : \text{Int})$.
     By definition, CVar(s_1) = CVar(s_2) = Use(e). By assumption, $\forall x \in \text{Use}(e), \sigma_1(x) = \sigma_2(x)$. By Lemma D.1, the expression evaluates to the same value w.r.t two value stores σ_1 and σ_2 respectively,
     - Both evaluations of expression lead to crash, $\mathcal{E}[\![e]\!]\sigma_1 = \mathcal{E}[\![e]\!]\sigma_2 = (\text{error}, v_{\mathfrak{of}})$.
       Then the execution of s_1 is as follows:
       (s_1, m_1)
       $= (id[n] := e, m_1(\sigma_1))$
       $\rightarrow (id[n] := (\text{error}, *), m_1(\sigma_1))$ by rule EEval'
       $\rightarrow (id[n] := 0, m_1(1/\mathfrak{f}))$ by rule ECrash.
       $\xrightarrow{i} (id[n] := 0, m_1(1/\mathfrak{f}))$ for any $i > 0$ by rule Crash.
       Similarly, s_2 does not terminate. The theorem holds.
     - Both evaluations of expression lead to no crash, $\mathcal{E}[\![e]\!]\sigma_1 = \mathcal{E}[\![e]\!]\sigma_2 = (v, v_{\mathfrak{of}})$.
       Then there are cases regarding if value mismatch occurs.
       $\surd$ The value v is only defined in type Long, $(\Gamma \vdash v : \text{Long}) \wedge \neg(\Gamma \vdash v : \text{Int})$.
       The execution of s_1 is as follows:
       (s_1, m_1)
       $= (id[n] := e, m_1(\sigma_1))$
       $\rightarrow (id[n] := (v, v_{\mathfrak{of}}), m_1(\sigma_1))$ by rule EEval'
       $\rightarrow (id[n] := v, m_1(\sigma_1))$ by rule EOflow-1 or EOflow-2.
       $\rightarrow (id[n] := v, m_1(1/\mathfrak{f}))$ by rule Assign-Err.
       $\xrightarrow{i} (id[n] := v, m_1(1/\mathfrak{f}))$ for any $i > 0$ by rule Crash.

Similarly, s_2 does not terminate. The theorem holds.

$\surd$ The value v is defined in type Int, $\Gamma \vdash v$: Int. The execution of s_1 is as follows:

(s_1, m_1)
$= (id[n] := e, m_1(\sigma_1))$
$\rightarrow (id[n] := (v, v_{\mathfrak{of}}), m_1(\sigma_1))$ by rule EEval'
$\rightarrow (id[n] := v, m_1(\sigma_1))$
by rule EOflow-1 or EOflow-2.
$\rightarrow (\text{skip}, m_1(\sigma_1(\sigma_1[v/(id, n)])))$
by rule Assign-A.

Similarly, s_2 terminate. The theorem holds. When the variable id is a global variable, by similar argument for the global variable, we can show that s_1 and s_2 terminate. Then the theorem holds.

- It is not the case that left value id is of type Int and the expression e is of type Long only, $\neg\big((\Gamma \vdash id : \text{Int}) \wedge (\Gamma \vdash e : \text{Long}) \wedge \neg(\Gamma \vdash e : \text{Int})\big)$.
  There are two cases based on if there is crash in evaluation of expression e.
  - Both evaluations of expression lead to crash, $\mathcal{E}[\![e]\!]\sigma_1 = \mathcal{E}[\![e]\!]\sigma_2 = (\text{error}, v_{\mathfrak{of}})$.
    By the same argument in case where left value id is of type Int and the expression e is of type Long only, this theorem holds.
  - Both evaluations of expression lead to no crash, $\mathcal{E}[\![e]\!]\sigma_1 = \mathcal{E}[\![e]\!]\sigma_2 = (v, v_{\mathfrak{of}})$.
    By the same argument in subcase of no value mismatch in case where left value id is of type Int and the expression e is of type Long only, this theorem holds.

If array variable id is a global variable, by similar argument above, the theorem holds.

(c) $lval = id_1[id_2]$.
By definition, $\text{Idx}(s_1) = \text{Idx}(s_2) = \{id_2\} \subseteq \text{CVar}(s_1) = \text{CVar}(s_2)$. By assumption, $\sigma_1(id_2) = \sigma_2(id_2) = n$. By the same argument in the case where $lval = id[n]$, the theorem holds.

3. $s_1 = s_2 =$ "input id",
By definition, $\text{CVar}(s_1) = \text{CVar}(s_2) = \{id_I\}$. By assumption $\sigma_1(id_I) = \sigma_2(id_I)$. There are cases regarding if input sequence is empty or not.

(a) There is empty input sequence, $\sigma_1(id_I) = \sigma_2(id_I) = \varnothing$.
Then the execution of s_1 continues as follows:

(s_1, m_1)
$= (\text{input}\, id, m_1(\sigma_1))$
$\rightarrow (\text{input}\, id, m_1(1/\mathfrak{f}))$ by rule In-7
$\xrightarrow{i} (\text{input}\, id, m_1(1/\mathfrak{f}))$ by rule Crash.

Similarly, s_2 does not terminate. The theorem holds.

(b) There is nonempty input sequence, $\sigma_1(id_I) = \sigma_2(id_I) \neq \varnothing$.
There are cases regarding if type of the variable id is Long or not.

i. id is of type Long, $\Gamma \vdash id$: Long;
Assuming id is a local variable, then the execution of s_1 continues as follows:

(s_1, m_1)
$= (\text{input}\, id, m_1(\sigma_1))$
$\rightarrow (\text{skip}, m_1(\sigma_1[v_{io}/id, \text{tl}(\sigma_1(id_I))/id_I,$
"$\sigma_1(id_{IO}) \cdot \underline{v}_{io}$"$/id_{IO}]))$ by rule In-3.

Similarly, s_2 terminates. The theorem holds.

When the variable id is a global variable, by similar argument, the theorem holds.

ii. id is of type Int or enumeration, $\Gamma \vdash id$: Int or enum id';
There are cases regarding if the head of input sequence can be transformed to type of id. Let $v_{io} = \text{hd}(\sigma_1(id_I))$.

- id is of type Int.
  If v_{io} is not of type Int, $\Gamma \vdash v_{io}$: Long and $\neg(\Gamma \vdash v_{io}$: Int), then the execution of s_1 continues as follows:

  (s_1, m_1)
  $= (\text{input}\, id, m_1(\sigma_1))$
  $\rightarrow (\text{input}\, id, m_1(1/\mathfrak{f}))$ by Rule In-4.
  $\xrightarrow{i} (\text{input}\, id, m_1(1/\mathfrak{f}))$ by Rule crash.

  Similarly, s_2 does not terminate. The theorem holds.
  If v_{io} is of type Int, $\Gamma \vdash v_{io}$: Long and $\Gamma \vdash v_{io}$: Int, assuming id is a local variable, then the execution of s_1 continues as follows:

  (s_1, m_1)
  $= (\text{input}\, id, m_1(\sigma_1))$
  $\rightarrow (\text{skip}, m_1(\sigma_1[v_{io}/id, \text{tl}(\sigma_1(id_I))/id_I,$
  "$\sigma_1(id_{IO}) \cdot \underline{v}_{io}$"$/id_{IO}]))$ by Rule In-8.

  Similarly, s_2 terminates. The theorem holds.
  When id is a global variable, by similar argument, the theorem holds.
- If id is of type enum $id' = \{l_1, ..., l_k\}$.
  If $(v_{io} < 1) \vee (v_{io} > k)$, then the execution of s_1 continues as follows:

  (s_1, m_1)
  $= (\text{input}\, id, m_1(\sigma_1))$
  $\rightarrow (\text{input}\, id, m_1(1/\mathfrak{f}))$ by Rule In-6.
  $\xrightarrow{i} (\text{input}\, id, m_1(1/\mathfrak{f}))$ by Rule crash.

  Similarly, s_2 does not terminate. The theorem holds. When id is a global variable, by similar argument, the theorem holds.
  If $1 \leq v_{io} \leq k$, assuming id is a local variable, then the execution of s_1 continues as follows:

  (s_1, m_1)
  $= (\text{input}\, id, m_1(\sigma_1))$
  $\rightarrow (\text{skip}, m_1(\sigma_1[l_{v_{io}}/id, \text{tl}(\sigma_1(id_I))/id_I,$
  "$\sigma_1(id_{IO}) \cdot \underline{v}_{io}$"$/id_{IO}]))$ by Rule In-5.

  Similarly, s_2 terminates. The theorem holds. When id is a global variable, by similar argument, the theorem holds.

(c) $s_1 = s_2 =$ "output e";
There are two cases based on if evaluation of expression e crashes. By definition, $\text{CVar}(s_1) = \text{CVar}(s_2) = \text{Err}(e)$. By assumption, $\forall x \in \text{Err}(e), \sigma_1(x) = \sigma_2(x)$. By Lemma D.2, evaluation of the expression e w.r.t two value stores σ_1 and σ_2 either both crash or both do not crash.

i. There is crash in evaluation of the expression e w.r.t two value stores σ_1 and σ_2, $\mathcal{E}[\![e]\!]\sigma_1 = (\text{error}, v^1_{\mathfrak{of}})$ and $\mathcal{E}[\![e]\!]\sigma_2 = (\text{error}, v^2_{\mathfrak{of}})$.
The execution of s_1 continues as follows:

(s_1, m_1)
$= (\text{output}\, e, m_1(\sigma_1))$
$\rightarrow (\text{output}\, (\text{error}, v^1_{\mathfrak{of}}), m_1(1/\mathfrak{f}))$ by Rule EEval'
$\rightarrow (\text{output}\, 0, m_1(1/\mathfrak{f}))$ by Rule ECrash.
$\xrightarrow{i} (\text{output}\, 0, m_1(1/\mathfrak{f}))$ by Rule crash.

Similarly, s_2 does not terminate. The theorem holds.

ii. There is no crash in evaluation of the expression e w.r.t two value stores σ_1 and σ_2, $\mathcal{E}[\![e]\!]\sigma_1 = (v_1, v^1_{\mathfrak{of}})$ and $\mathcal{E}[\![e]\!]\sigma_2 = (v_2, v^2_{\mathfrak{of}})$.
According to rule Out-1 and Out-2, there is no exception in transformation of different typed output value. We therefore only show the execution for output value of Int type. The execution of s_1 continues as follows:
(s_1, m_1)
$=$ (output $e, m_1(\sigma_1)$)
$\rightarrow$(output $(v_1, v^1_{\mathfrak{of}}), m_1(1/\mathfrak{f})$) by Rule EEval'
$\rightarrow$(output $v_1, m_1(v^1_{\mathfrak{of}}/\mathfrak{of})$)
by Rule EOflow-1 or EOflow-2.
$\rightarrow$(skip, $m_1(\sigma_1[``\sigma(id_{IO}) \cdot \overline{v}_1"/id_{IO}])$)
by Rule Out-1.
Similarly, s_2 terminates. Theorem holds.

Second s_1 and s_2 are input statement with same type variable: s_1 = "input id_1", s_2 = "input id_2" where $(\Gamma_{s_1} \vdash id_1 : t) \wedge (\Gamma_{s_2} \vdash id_2 : t)$;
The theorem holds by similar argument for the case $s_1 = s_2 =$ input id.

Third s_1 = "output e" or "$id_1 := e$", s_2 = "output e" or "$id_2 := e$" where both of the following hold:

- There is no possible value mismatch in "$id_1 := e$", $\neg(\Gamma_{s_1} \vdash id_1 : \text{Int}) \vee \neg(\Gamma_{s_1} \vdash e : \text{Long}) \vee (\Gamma_{s_1} \vdash e : \text{Int})$.
- There is no possible value mismatch in "$id_2 := e$", $\neg(\Gamma_{s_2} \vdash id_2 : \text{Int}) \vee \neg(\Gamma_{s_2} \vdash e : \text{Long}) \vee (\Gamma_{s_2} \vdash e : \text{Int})$.

We show that the evaluations of the expression e w.r.t the value stores σ_1 and σ_2 either both raise an exception or both do not. By the definition of crash variables, the crash variables of s_1 are those obtained by the function Err(e), CVar(s_1) = Err(e). Similarly, the termination deciding variables of s_2 are Err(e). By assumption, the initial value stores σ_1 and σ_2 agree on values of those in CVar(s_1) and CVar(s_2), $\forall x \in \text{Err}(e) = (\text{CVar}(s_1) \cup \text{CVar}(s_2)) : \sigma_1(x) = \sigma_2(x)$. By Lemma D.2, the evaluations of expression e w.r.t two value stores, σ_1 and σ_2, either both raise an exception or both do not raise an exception.

1. The evaluations of the expression e raise an exception w.r.t two value stores σ_1 and σ_2, $\mathcal{E}'[\![e]\!]\sigma_1 = (\text{error}, v^1_{\mathfrak{of}}), \mathcal{E}'[\![e]\!]\sigma_2 = (\text{error}, v^2_{\mathfrak{of}})$:
We show the execution of s_1 proceeds to an configuration where the crash flag is set and then does not terminate.
When s_1 = "output e", the execution of "output e" proceeds as follows.
(output $e, m_1(\sigma_1)$)
$\rightarrow$(output (error, $v^1_{\mathfrak{of}}), m_1(\sigma_1)$) by rule EEval'
$\rightarrow$(output $0, m_1(1/\mathfrak{f})$) by rule ECrash
$\xrightarrow{i}$(output $0, m_1(1/\mathfrak{f})$) for any $i \geq 0$, by rule Crash.
When s_1 = "$id_1 := e$", the execution of "$id_1 := e$" proceeds as follows.
$(id_1 := e, m_1(\sigma_1))$
$\rightarrow(id_1 :=$ (error, $v^1_{\mathfrak{of}}), m_1(\sigma_1))$ by rule EEval'
$\rightarrow(id_1 := 0, m_1(1/\mathfrak{f}))$ by rule ECrash
$\xrightarrow{i}(id_1 := 0, m_1(1/\mathfrak{f}))$ for any $i \geq 0$, by rule Crash.
Similarly, the execution of s_2 proceeds to a configuration where the crash flag is set. Then, by the crash rule, the execution of s_2 does not terminate. The theorem 3 holds.
2. the evaluations of expression e do not raise an exception w.r.t two value stores, σ_1 and σ_2, $\mathcal{E}'[\![e]\!]\sigma_1 = (v_1, v^1_{\mathfrak{of}}), \mathcal{E}'[\![e]\!]\sigma_2 = (v_2, v^2_{\mathfrak{of}})$:
We show the execution of s_1 terminates.
When s_1 = output (e), the execution of output (e) proceeds as follows. W.l.o.g, we assume expression e is of type Int. This is allowed by the condition that it does not hold that $(\Gamma_{s_1} \vdash e : \text{Long}) \wedge \neg(\Gamma_{s_1} \vdash e : \text{Int})$.
(output $e, m_1(\sigma_1)$)
$\rightarrow$(output $(v_1, v^1_{\mathfrak{of}}), m_1(\sigma_1)$) by rule EEval'
$\rightarrow$(output $v_1, m_1(v^1_{\mathfrak{of}}/\mathfrak{of}, \sigma_1)$)
by rule E-Oflow1 or E-Oflow2
$\rightarrow$(skip, $m_1(\sigma_1[``\sigma_1(id_{IO}) \cdot \bar{v_1}"/id_{IO}])$) by rule Out.
When s_1 = "$id_1 := e$", by assumption, the expression e is of type Int, there is no possible value mismatch in execution of "$id_1 := e$" because the only possible value mismatch occurs when assigning a value of type Long but not Int to a variable of type Int. By the condition $\neg(\Gamma_{s_1} \vdash id_1 : \text{Int}) \vee \neg(\Gamma_{s_1} \vdash e : \text{Long}) \vee (\Gamma_{s_2} \vdash e : \text{Int})$, when expression e is of type Long, then the variable id_1 is not of type Int. In summary, there is no value mismatch.
The execution of "$id_1 := e$" proceeds as follows.
$(id_1 := e, m_1(\sigma_1))$
$\rightarrow(id_1 := (v_1, v^1_{\mathfrak{of}}), m_1(\sigma_1))$ by rule EEval'
$\rightarrow(id_1 := v_1, m_1(v^1_{\mathfrak{of}}/\mathfrak{of}, \sigma_1))$ by rule EEval'
$\rightarrow$(skip, $m_1(\sigma_1[v_1/id_1])$) by the rule Assign.

When id_1 is a variable of enumeration or Long type, by similar argument, the theorem still holds.
Similarly, the execution of s_2 terminates when started in the state $m_2(\sigma_2)$. Theorem 3 holds.

□

Theorem 4. *If two statement sequences S_1 and S_2 satisfy the proof rule of termination in the same way, $S_1 \equiv^S_H S_2$, and their respective initial states $m_1(\mathfrak{f}_1, \sigma_1)$ and $m_2(\mathfrak{f}_2, \sigma_2)$ with crash flags not set, $\mathfrak{f}_1 = \mathfrak{f}_2 = 0$, and whose value stores agree on values of the termination deciding variables of S_1 and S_2, $\forall x \in$ TVar$(S_1) \cup$ TVar$(S_2) : \sigma_1(x) = \sigma_2(x)$, then S_1 and S_2 terminate in the same way when started in states m_1 and m_2 respectively: $(S_1, m_1) \equiv_H (S_2, m_2)$.*

Proof. The proof is by induction on size(S_1) + size(S_2), the sum of program size of S_1 and S_2.
Base case. S_1 and S_2 are simple statement. By Theorem 3, Theorem 4 holds.
Induction step.
There are two hypotheses. The hypothesis IH is that Theorem 4 holds when size(S_1) + size$(S_2) = k \geq 2$.
We show Theorem 4 holds when size(S_1) + size$(S_2) = k + 1$.
The proof of Theorem 4 is a case analysis according to the cases in the definition of the proof rule of termination in the same way, $S_1 \equiv^S_H S_2$.

1. S_1 and S_2 are one statement and one of the following holds.
   (a) S_1 = "If(e) then $\{S^t_1\}$ else $\{S^f_1\}$", S_2 = "If(e) then $\{S^t_2\}$ else $\{S^f_2\}$" such that one of the following holds:
      i. $S^t_1, S^f_1, S^t_2, S^f_2$ are all sequences of "skip";
      We show that the evaluation of expression e w.r.t the value store σ_1 and σ_2 either both raise an exception or both do not. By the definition of crash/loop variables, CVar(S^t_1) = CVar$(S^f_1) = \emptyset$, LVar$(S_1) = \emptyset$. By the definition of termination deciding variables, the termination deciding variables of S_1 is the crash variables of S_1, TVar(S_1) = CVar(S_1) = Err(e). By assumption, the value stores σ_1 and σ_2 agree on the values of those in the crash variables of S_1 and S_2, $\forall x \in$ Err(e) = TVar(S_1) = TVar$(S_2), \sigma_1(x) = \sigma_2(x)$. By

the property of the expression meaning function $\mathcal{E}$, the evaluation of predicate expression e of S_1 and S_2 w.r.t value store σ_1 and σ_2 either both crash or both do not crash, $(\mathcal{E}[\![e]\!]\sigma_1 = \mathcal{E}[\![e]\!]\sigma_2 = \text{error}) \vee \big((\mathcal{E}[\![e]\!]\sigma_1 \neq \text{error}) \wedge (\mathcal{E}[\![e]\!]\sigma_2 \neq \text{error})\big)$. Then we show that Theorem 4 holds in either of the two possibilities.

A. $\mathcal{E}[\![e]\!]\sigma_1 = \mathcal{E}[\![e]\!]\sigma_2 = \text{error}$.
The execution of S_1 proceeds as follows:

$(\text{If}(e) \text{ then } \{S_1^t\} \text{ else } \{S_1^f\}, m_1(\sigma_1))$
$\rightarrow(\text{If}(\text{error}) \text{ then } \{S_1^t\} \text{ else } \{S_1^f\}, m_1(\sigma_1))$ by rule EEval
$\rightarrow(\text{If}(0) \text{ then } \{S_1^t\} \text{ else } \{S_1^f\}, m_1(1/\mathfrak{f}, \sigma_1))$ by rule ECrash
$\xrightarrow{i}(\text{If}(0) \text{ then } \{S_1^t\} \text{ else } \{S_1^f\}, m_1(1/\mathfrak{f}, \sigma_1))$ for any $i \geq 0$, by rule Crash.

Similarly, the execution of S_2 started in the state $m_2(\sigma_2)$ does not terminate. The theorem 4 holds.

B. $(\mathcal{E}[\![e]\!]\sigma_1 \neq \text{error}) \wedge (\mathcal{E}[\![e]\!]\sigma_2 \neq \text{error})$.
W.l.o.g, $\mathcal{E}[\![e]\!]\sigma_1 = v_1 \neq 0$, $\mathcal{E}[\![e]\!]\sigma_2 = 0$. Then the execution of S_1 proceeds as follows.

$(\text{If}(e) \text{ then } \{S_1^t\} \text{ else } \{S_1^f\}, m_1(\sigma_1))$
$\rightarrow(\text{If}(v_1) \text{ then } \{S_1^t\} \text{ else } \{S_1^f\}, m_1(\sigma_1))$ by rule EEval
$\rightarrow(S_1^t, m_1(\sigma_1))$ by rule If-T
$\xrightarrow{*}(\text{skip}, m_1')$ by rule Skip.

Similarly, the execution of S_2 started in the state $m_2(\sigma_2)$ terminates. The theorem 4 holds.

ii. At least one of $S_1^t, S_1^f, S_2^t, S_2^f$ is not a sequence of "skip" and $(S_1^t \equiv_H^S S_2^t) \wedge (S_1^f \equiv_H^S S_2^f)$;
W.l.o.g., S_1^t is not of "skip" only. We show that the evaluation of the expression e w.r.t the value stores σ_1 and σ_2 either both raise an exception or both produce the same integer value. Then there is either some loop statement in S_1^t or the crash variables of S_1^t are not $\emptyset$ or both.

A. When there is some loop statement in S_1^t, then, by the definition of loop variables, the loop variables of S_1 include all variables used in the predicate expression of S_1, LVar(S_1) = Use$(e) \cup$ LVar$(S_1^t) \cup$ LVar(S_1^f).

B. When the crash variables of S_1^t are not $\emptyset$, then, by the definition of crash variables, the crash variables of S_1 include all variables used in the predicate expression of S_1, CVar(S_1) = Use$(e) \cup$ CVar$(S_1^t) \cup$ CVar(S_1^f).

In summary, all variables used in predicate expression of S_1 is a subset of termination deciding variables of S_1, Use$(e) \subseteq$ TVar(S_1). By assumption, the value store σ_1 and σ_2 agree on the values of those in the termination deciding variables of S_1 and S_2. It follows, by the property of expression meaning function $\mathcal{E}$, the evaluation of the predicate expression e of S_1 and S_2 produce the same value w.r.t the value store σ_1 and σ_2, $\mathcal{E}[\![e]\!]\sigma_1 = \mathcal{E}[\![e]\!]\sigma_2$. Then either the evaluations of the predicate expression e of S_1 and S_2 both crash w.r.t the value store σ_1 and σ_2, or both evaluations produce the same integer value, $(\mathcal{E}[\![e]\!]\sigma_1 = \mathcal{E}[\![e]\!]\sigma_2 = \text{error}) \vee (\mathcal{E}[\![e]\!]\sigma_1 = \mathcal{E}[\![e]\!]\sigma_2 = v \neq \text{error})$. We show Theorem 4 holds in either of the two possibilities.

A. $\mathcal{E}[\![e]\!]\sigma_1 = \mathcal{E}[\![e]\!]\sigma_2 = \text{error}$.
The execution of S_1 proceeds as follows:

$(\text{If}(e) \text{ then } \{S_1^t\} \text{ else } \{S_1^f\}, m_1(\sigma_1))$
$\rightarrow(\text{If}(\text{error}) \text{ then } \{S_1^t\} \text{ else } \{S_1^f\}, m_1(\sigma_1))$ by rule EEval
$\rightarrow(\text{If}(0) \text{ then } \{S_1^t\} \text{ else } \{S_1^f\}, m_1(1/\mathfrak{f}, \sigma_1))$ by rule ECrash
$\xrightarrow{i}(\text{If}(0) \text{ then } \{S_1^t\} \text{ else } \{S_1^f\}, m_1(1/\mathfrak{f}, \sigma_1))$ for any $i \geq 0$, by rule Crash.

Similarly, the execution of S_2 started from state $m_2(\sigma_2)$ does not terminate. The theorem 4 holds.

B. $\mathcal{E}[\![e]\!]\sigma_1 = \mathcal{E}[\![e]\!]\sigma_2 = v \neq \text{error}$, w.l.o.g., $v = 0$.
Then the execution of S_1 proceeds as follows:

$(\text{If}(e) \text{ then } \{S_1^t\} \text{ else } \{S_1^f\}, m_1(\sigma_1))$
$\rightarrow(\text{If}(0) \text{ then } \{S_1^t\} \text{ else } \{S_1^f\}, m_1(\sigma_1))$ by rule EEval
$\rightarrow(S_1^f, m_1(\sigma_1))$ by rule If-F.

Similarly, after two steps of execution, S_2 gets to the configuration $(S_2^f, m_2(\sigma_2))$.

We show that S_1^f and S_2^f terminate in the same way when started in the state $m_1(\sigma_1)$ and $m_2(\sigma_2)$ respectively. Because $S_1^f \equiv_H^S S_2^f$, by Corollary 5.1, the termination deciding variables of S_1^f and S_2^f are same, TVar(S_1^f) = TVar(S_2^f). By the definition of crash/loop variables, CVar$(S_1^f) \subseteq$ CVar(S_1) and LVar$(S_1^f) \subseteq$ LVar(S_1). Hence, the termination deciding variables of S_1^f are a subset of the termination deciding variables of S_1, TVar$(S_1^f) \subseteq$ TVar(S_1). Similarly, TVar$(S_2^f) \subseteq$ TVar(S_2). Therefore, the value store σ_1 and σ_2 agree on the values of those in the termination deciding variables of S_1^f and S_2^f, $\forall y \in$ TVar$(S_1^f) \cup$ TVar(S_2^f) : $\sigma_1(y) = \sigma_2(y)$. In addition, the sum of program size of S_1^f and S_2^f is less than k because program size of each of S_1^t and S_2^t is great than or equal to one, size(S_1^f) + size$(S_2^f) < k$. As is shown, crash flags are not set. Therefore, by the hypothesis IH, S_1^f and S_2^f terminate in the same way when started in state $m_1(\mathfrak{f}_1, \sigma_1)$ and $m_2(\mathfrak{f}_2, \sigma_2)$, $(S_1^f, m_1(\mathfrak{f}_1, \sigma_1)) \equiv_H (S_2^f, m_2(\mathfrak{f}_2, \sigma_2))$. Hence, Theorem 4 holds.

(b) S_1 = "while$_{\langle n_1 \rangle}(e)$ $\{S_1''\}$", S_2 = "while$_{\langle n_2 \rangle}(e)$ $\{S_2''\}$" such that both of the following hold:

- $S_1'' \equiv_H^S S_2''$;
- S_1'' and S_2'' have equivalent computation of TVar$(S_1) \cup$ TVar(S_2);

By Corollary 5.3, we show S_1 and S_2 terminate in the same way when started from state $m_1(\mathfrak{f}_1, m_c^1, \sigma_1)$ and $m_2(\mathfrak{f}_2, m_c^2, \sigma_2)$ respectively. We need to show that all required conditions are satisfied.

- The crash flags are not set, $\mathfrak{f}_1 = \mathfrak{f}_2 = 0$.
- The loop counter value of S_1 and S_2 are zero: $m_c^1(n_1) = m_c^2(n_2) = 0$.
- The value stores σ_1 and σ_2 agree on the values of those in the termination deciding variables of S_1 and S_2, $\forall x \in$ TVar$(S_1) \cap$ TVar(S_2) : $\sigma_1(x) = \sigma_2(x)$.
The three above conditions are from assumption.
- S_1 and S_2 have same set of termination deciding variables, TVar(S_1) = TVar(S_2).
By Corollary 5.1.
- The loop body S_1'' of S_1 and S_2'' of S_2 terminate in the same way when started in state $m_{S_1}(\mathfrak{f}_{S_1}, \sigma_{S_1})$ and $m_{S_2}(\mathfrak{f}_{S_2}, \sigma_{S_2})$ with crash flags not set and in which value stores agree on the values of those in the termination deciding variables of S_1'' and S_2'': $((\forall x \in$ TVar$(S_1'') \cup$ TVar(S_2'') : $\sigma_{S_1}(x) = \sigma_{S_2}(x)) \wedge (\mathfrak{f}_{S_1} = \mathfrak{f}_{S_2} = 0)) \Rightarrow (S_1'', m_{S_1}(\mathfrak{f}_{S_1}, \sigma_{S_1})) \equiv_H (S_2'', m_{S_2}(\mathfrak{f}_{S_2}, \sigma_{S_2}))$.
By the definition of program size, size(S_1) = size(S_1'')+1, size(S_2) = size(S_2'')+1. Then, size(S_1'')+size$(S_2'') <$

k. Then, by the hypothesis IH, the loop body S_1'' of S_1 and S_2'' of S_2 terminate in the same way when started in state $m_{S_1}(\sigma_{S_1})$ and $m_{S_2}(\sigma_{S_2})$ with crash flags not set and whose value stores agree on values of the termination deciding variables of S_1'' and S_2''.

Then, by Corollary 5.3, S_1 and S_2 terminate in the same way when started in the states $m_1(m_c^1, \sigma_1)$ and $m_2(m_c^2, \sigma_2)$ respectively. The theorem 4 holds.

2. S_1 and S_2 are not both one statement and one of the following holds:
   (a) $S_1 = S_1'; s_1, S_2 = S_2'; s_2$ and all of the following hold:
   - $S_1' \equiv_H^S S_2'$;
   - S_1' and S_2' have equivalent computation of TVar$(s_1) \cup$ TVar(s_2);
   - $s_1 \equiv_H^S s_2$ where s_1 and s_2 are not sequences of "skip";

   By the hypothesis IH, we show that S_1' and S_2' terminate in the same way when started in the states $m_1(\mathfrak{f}_1, \sigma_1), m_2(\mathfrak{f}_2, \sigma_2)$ respectively, $(S_1', m_1(\mathfrak{f}_1, \sigma_1)) \equiv_H (S_2', m_2(\mathfrak{f}_2, \sigma_2))$. We need to show all required conditions are satisfied.
   - Crash flags are not set, $\mathfrak{f}_1 = \mathfrak{f}_2 = 0$;
     By assumption.
   - size(S_1') + size$(S_2') < k$.
     By the definition, size$(s_1) \geq 1$, size$(s_2) \geq 1$. Hence size(S_1') + size$(S_2') < k$.
   - Value stores σ_1 and σ_2 agree on values of the termination deciding variables of S_1' and S_2'.
     Besides, by the definition of loop/crash variables, LVar$(S_1') \subseteq$ LVar(S_1) and CVar$(S_1') \subseteq$ CVar(S_1). Hence, TVar$(S_1') \subseteq$ TVar(S_1). Similarly, TVar$(S_2') \subseteq$ TVar(S_2). Then, value stores σ_1 and σ_2 agree on the values of those in the termination deciding variables of S_1' and S_2', $\forall x \in$ TVar$(S_1') \cup$ TVar(S_2') : $\sigma_1(x) = \sigma_2(x)$.

   Then, by the hypothesis IH, S_1' and S_2' terminate in the same way when started in the states $m_1(\mathfrak{f}_1, \sigma_1), m_2(\mathfrak{f}_2, \sigma_2)$ respectively, $(S_1', m_1(\mathfrak{f}_1, \sigma_1)) \equiv_H (S_2', m_2(\mathfrak{f}_2, \sigma_2))$.

   If the execution of S_1' and S_2' terminate when started in the states $m_1(\mathfrak{f}_1, \sigma_1)$ and $m_2(\mathfrak{f}_2, \sigma_2)$ respectively, we show that s_1 and s_2 terminate in the same way. We prove that S_1' and S_2' equivalently compute the termination deciding variables of s_1 and s_2 by Theorem 2.
   - Crash flags are not set, $\mathfrak{f}_1 = \mathfrak{f}_2 = 0$;
     By definition of terminating execution of S_1' and S_2' when started in states m_1 and m_2 respectively.
   - The executions of S_1' and S_2' terminate when started in the states $m_1(\sigma_1)$ and $m_2(\sigma_2)$.
     By assumption, $(S_1', m_1(\sigma_1)) \xrightarrow{*} (\text{skip}, m_1'(\sigma_1'))$ and $(S_2', m_2(\sigma_2)) \xrightarrow{*} (\text{skip}, m_2'(\sigma_2'))$.
   - s_1 and s_2 have same termination deciding variables.
     By Corollary 5.1, s_1 and s_2 have same termination deciding variables, TVar(s_1) = TVar(s_2) = TVar(s).
   - Value stores σ_1 and σ_2 agree on the values of variables in Imp$(S_1', \text{TVar}(s)) \cup$ Imp$(S_2', \text{TVar}(s))$.
     By the definition of loop/crash variables, Imp$(S_1', \text{LVar}(s_1)) \subseteq$ LVar(S_1) and Imp$(S_1', \text{CVar}(s_1)) \subseteq$ CVar(S_1). Hence, by Lemma C.2, the imported variables in S_1' relative to the termination deciding variables of s_1 is a subset of the termination deciding variables of S_1, Imp$(S_1', \text{TVar}(s))$ $\subseteq$ TVar(S_1). Similarly, Imp$(S_2', \text{TVar}(s)) \subseteq$ TVar(S_2). Thus, by assumption, the value stores σ_1 and σ_2 agree on the values of the variables in Imp$(S_1', \text{TVar}(s)) \cup$ Imp$(S_2', \text{TVar}(s))$.

   By Theorem 2, $\forall x \in$ TVar(s) : $\sigma_1'(x) = \sigma_2'(x)$.

   By Corollary E.1, $(S_1'; s_1, m_1(\sigma_1)) \xrightarrow{*} (s_1, m_1'(\mathfrak{f}_1, \sigma_1'))$ and $(S_2'; s_2, m_2(\sigma_2)) \xrightarrow{*} (s_2, m_2'(\mathfrak{f}_2, \sigma_2'))$. Then, by the hypothesis IH, we show that s_1 and s_2 terminate in the same way when started in the states $m_1'(\sigma_1')$ and $m_2'(\sigma_2')$. We show that all required conditions are satisfied. size(s_1) + size$(s_2) < k$ because size$(S_1') \geq 1$, size$(S_2') \geq 1$ by the definition of program size. If s_1, s_2 are loop statement, then, by the assumption of unique loop labels, $s_1 \notin S_1', s_2 \notin S_2'$. Then, by Corollary E.4, the loop counter value of s_1 and s_2 is not redefined in the execution of S_1' and S_2' respectively. By the hypothesis IH, s_1 and s_2 terminate in the same way when started in the states $m_1'(\mathfrak{f}_1, \sigma_1')$ and $m_2'(\mathfrak{f}_2, \sigma_2')$, $(s_1, m_1'(\mathfrak{f}_1, \sigma_1')) \equiv_H (s_2, m_2'(\mathfrak{f}_2, \sigma_2'))$. The theorem 4 holds.

   (b) One last statement is "skip": w.l.o.g., $(s_2 = \text{"skip"}) \wedge (S_1 \equiv_H^S S_2')$.
   We show that S_1 and S_2' terminate in the same way when started in the states $m_1(\sigma_1)$ and $m_2(\sigma_2)$ respectively by the hypothesis IH. By the definition of crash/loop variables, CVar$(S_2') \subseteq$ CVar(S_2), LVar$(S_2') \subseteq$ LVar(S_2). Then, by assumption, $\forall x \in$ TVar$(S_2') \cup$ TVar(S_1) : $\sigma_1(x) = \sigma_2(x)$. Besides, size $(s_2) \geq 1$ by the definition of program size. Then size (S_1) + size $(S_2') \leq k$. By the hypothesis IH, S_1 and S_2' terminate in the same way when started in the states $m_1(\mathfrak{f}_1, \sigma_1), m_2(\mathfrak{f}_2, \sigma_2)$, $(S_1, m_1(\mathfrak{f}_1, \sigma_1)) \equiv_H (S_2', m_2(\mathfrak{f}_2, \sigma_2))$.
   When the execution of S_1 and S_2' terminate when started in the states $m_1(\sigma_1)$ and $m_2(\sigma_2)$ respectively, s_2 terminates after the execution of S_2' by the definition of terminating execution.

   (c) One last statement is a "duplicate" statement such that one of the following holds:
   W.l.o.g., $S_2 = S_2'; s_2'; S_2''; s_2$ and all of the following hold:
   - $S_1 \equiv_H^S S_2'; s_2'; S_2''$;
   - $s_2' \equiv_H^S s_2$;
   - Def$(s_2'; S_2'') \cap$ TVar$(s_2) = \emptyset$;
   - $s_2 \neq$ "skip";

   We show that S_1 and $S_2'; s_2'; S_2''$ terminate in the same way when started in the states $m_1(\mathfrak{f}_1, \sigma_1)$,
   $m_2(\mathfrak{f}_2, \sigma_2)$ respectively by the hypothesis IH. The proof is same as that in case a).
   We show that s_2 terminates if the execution of $S_2'; s_2'; S_2''$ terminates. We need to prove that s_2' and s_2 start in the states agreeing on the values of variables in TVar(s_2). By assumption, $S_2'; s_2'; S_2''$ terminates, $(S_2'; s_2'; S_2'', m_2(\mathfrak{f}_2, \sigma_2)) \xrightarrow{*}$ $(\text{skip}, m_2'(\mathfrak{f}_2, \sigma_2'))$. Then, by Corollary E.1, $(S_2'; s_2'; S_2''; s_2, m_2(\mathfrak{f}_2, \sigma_2))$ $\xrightarrow{*} (s_2, m_2'(\mathfrak{f}_2, \sigma_2'))$. In addition, the execution of S_2' and s_2' must terminate because the execution of $S_2'; s_2'; S_2''$ terminates,
   $(S_2'; s_2'; S_2''; s_2, m_2(\mathfrak{f}_2, \sigma_2)) \xrightarrow{*} (s_2'; S_2''; s_2, m_2''(\mathfrak{f}_2, \sigma_2'')) \xrightarrow{*}$ $(s_2, m_2'(\mathfrak{f}_2, \sigma_2'))$.
   By assumptin, Def$(s_2'; S_2'') \cap$ TVar$(s_2) = \emptyset$. Then, by Corollary E.2, the value store σ_2'' and σ_2' agree on values of the termination deciding variables of s_2, $\forall x \in$ TVar(s_2) : $\sigma_2''(x) = \sigma_2'(x)$. By Corollary 5.1, TVar$(s_2')$ = TVar(s_2). Because the execution of s_2' terminates, then the execution of s_2 terminates when started in the state $m_2'(\mathfrak{f}_2, \sigma_2')$ by the hypothesis IH, $(s_2, m_2'(\mathfrak{f}_2, \sigma_2')) \xrightarrow{*} (\text{skip}, m_2'')$.
   In addition, we show that there is no input statement in s_2 by contradiction. Suppose there is input statement in s_2. By Lemma 5.11, $id_I \in$ CVar(s_2). Hence, the input sequence variable is in the termination deciding variables of s_2, $id_I \in$ TVar(s_2). By Corollary 5.1, TVar(s_2) =

TVar(s'_2). Then, there must be one input statement in s'_2. Otherwise, by Lemma 5.2, the input sequence variable is not in the termination deciding variables of s'_2. A contradiction against the result that $id_I \in$ TVar(s'_2). Since there is one input statement in s'_2, by Lemma 5.11, $id_I \in$ Def(s'_2). Thus, by defintion, $id_I \in$ Def$(s'_2; S''_2)$. Then, Def$(s'_2; S''_2) \cap$ TVar$(s_2) \neq \emptyset$. A contradiction. Therefore, there is no input statement in s_2.

(d) $S_1 = S'_1; s_1; s'_1$; and $S_2 = S'_2; s_2; s'_2$ where s_1 and s_2 are reordered and all of the following hold:
- $S'_1 \equiv^S_H S'_2$;
- S'_1 and S'_2 have equivalent computation of TVar$(s_1; s'_1) \cup$ TVar$(s_2; s'_2)$.
- $s_1 \equiv^S_H s'_2$;
- $s'_1 \equiv^S_H s_2$;
- Def$(s_1) \cap$ TVar$(s'_1) = \emptyset$;
- Def$(s_2) \cap$ TVar$(s'_2) = \emptyset$;

The proof is to show that if S_1 terminates when started in the state m_1, the S_2 terminates when started in the state m_2, and vice versa. Due to the symmetric conditions, it is suffice to show one direction that, w.l.o.g., $(S_1, m_1) \xrightarrow{*}$ (skip$, m'_1) \Rightarrow (S_2, m_2) \xrightarrow{*}$ (skip$, m'_2)$.

We show that the execution of S'_2 terminates by the hypothesis IH. We need to show that all required conditions are satisfied.
- size(S'_1) + size$(S'_2) < k$.
  This is because size$(s_1; s'_1) > 1$, size$(s_2; s'_2) > 1$.
- Initial value stores σ_1 and σ_2 agree on values of the termination deciding variables of S'_1 and S'_2, $\forall x \in$ TVar$(S'_1) \cup$ TVar(S'_2) : $\sigma_1(x) = \sigma_2(x)$.
  We show that TVar$(S'_1) \subseteq$ TVar(S_1). In the following, we prove that CVar$(S'_1) \subseteq$ CVar(S_1).
  $$\begin{array}{ll} & \text{CVar}(S'_1) \\ \subseteq & \text{CVar}(S'_1; s_1) \text{ by the defintion of CVar}(S'_1; s_1) \\ \subseteq & \text{CVar}(S'_1; s_1; s'_1) \text{ by the defintion of CVar}(S'_1; s_1; s'_1) \end{array}$$
  Similarly, LVar$(S'_1) \subseteq$ LVar(S_1). Hence, TVar$(S'_1) \subseteq$ TVar(S_1). Similarly, TVar$(S'_2) \subseteq$ TVar(S_2). By assumption, initial value stores σ_1 and σ_2 agree on values of the termination deciding variables of S'_1 and S'_2.

By the hypothesis IH, $(S'_1, m_1(\sigma_1)) \equiv_H (S'_2, m_2(\sigma_2))$. Because the execution of S_1 terminates, then S'_1 terminates when started in the state $m_1(\sigma_1)$, $(S'_1, m_1(\sigma_1)) \xrightarrow{*}$ (skip$, m'_1(\sigma'_1))$. Therefore, S'_2 termiantes when started in the state $m_2(\sigma_2)$, $(S'_2, m_2(\sigma_2)) \xrightarrow{*}$ (skip$, m'_2(\sigma'_2))$.

We show that after the execution of S'_1 and S'_2, value stores agree on values of the termination deciding variables of $s_1; s'_1$ and $s_2; s'_2$, $\forall x \in$ TVar$(s_1; s'_1) \cup$TVar$(s_2; s'_2)$, $\sigma'_1(x) = \sigma'_2(x)$. We split the argument into two steps.

i. We show that TVar$(s_1; s'_1) =$ TVar$(s_2; s'_2)$.
   By Corollary 5.1, TVar$(s_1) =$ TVar(s'_2) and TVar$(s'_1) =$ TVar(s_2). Then we show that TVar$(s_1; s'_1) =$ TVar$(s_1) \cup$ TVar(s'_1). To do that, we show that CVar$(s_1; s'_1) =$ CVar$(s_1) \cup$ CVar(s'_1).
   $$\begin{array}{ll} & \text{CVar}(s_1; s'_1) \\ = & \text{CVar}(s_1) \cup \text{Imp}(s_1, \text{CVar}(s'_1)) \text{ by the defintion of CVar}(s_1; s'_1) \\ = & \text{CVar}(s_1) \cup \text{CVar}(s'_1) \text{ by Def}(s_1) \cap \text{TVar}(s'_1) = \emptyset \text{ and} \\ & \text{the defintion of Imp}(\cdot). \end{array}$$
   Similarly, LVar$(s_1; s'_1) =$ LVar$(s_1) \cup$ LVar(s'_1). Thus, TVar$(s_1; s'_1) =$ TVar$(s_1) \cup$TVar(s'_1). Similarly, TVar$(s_2; s'_2) =$ TVar$(s_2) \cup$ TVar(s'_2). In summary, TVar$(s_1; s'_1) =$ TVar$(s_2; s'_2)$.

ii. We show that Imp$(S'_1,$ TVar$(s_1; s'_1)) \subseteq$ TVar(S_1) and Imp$(S'_2,$ TVar$(s_2; s'_2)) \subseteq$ TVar(S_2).
   W.l.o.g, we show that Imp$(S'_1,$ TVar$(s_1; s'_1)) \subseteq$ TVar(S_1).
   Specifically, we show Imp$(S'_1,$ CVar$(s_1; s'_1)) \subseteq$ CVar(S_1).
   $$\begin{array}{ll} & \text{CVar}(s_1; s'_1) \\ = & \text{CVar}(s_1) \cup \text{Imp}(s_1, \text{CVar}(s'_1)) \text{ (1)} \\ & \text{by the defintion of CVar}(s_1; s'_1) \end{array}$$
   $$\begin{array}{ll} & \text{Imp}(S'_1, \text{CVar}(s_1; s'_1)) \\ = & \text{Imp}(S'_1, \text{CVar}(s_1) \cup \text{Imp}(s_1, \text{CVar}(s'_1))) \text{ by (1)} \\ = & \text{Imp}(S'_1, \text{CVar}(s_1)) \cup \text{Imp}(S'_1, \text{Imp}(s_1, \text{CVar}(s'_1))) \text{ (2)} \\ & \text{by Lemma C.2} \end{array}$$
   $$\begin{array}{ll} & \text{Imp}(S'_1, \text{CVar}(s_1)) \\ \subseteq & \text{CVar}(S'_1; s_1) \text{ by the defintion of CVar}(\cdot) \\ \subseteq & \text{CVar}(S'_1; s_1; s'_1) \text{ by the defintion of CVar}(\cdot) \end{array}$$
   $$\begin{array}{ll} & \text{Imp}(S'_1, \text{Imp}(s_1, \text{CVar}(s'_1))) \\ = & \text{Imp}(S'_1; s_1, \text{CVar}(s'_1)) \text{ by Lemma C.1} \\ \subseteq & \text{CVar}(S'_1; s_1; s'_1) \text{ by the defintion of CVar}(\cdot). \end{array}$$
   $$\begin{array}{ll} & \text{Imp}(S'_1, \text{CVar}(s_1)) \cup \text{Imp}(S'_1, \text{Imp}(s_1, \text{CVar}(s'_1))) \\ \subseteq & \text{CVar}(S'_1; s_1; s'_1) \text{ by (3) and (4).} \end{array}$$
   In conclusion, Imp$(S'_1,$ CVar$(s_1; s'_1)) \subseteq$ CVar(S_1). Similarly, Imp$(S'_1,$ LVar$(s_1; s'_1)) \subseteq$ LVar(S_1). Thus, Imp$(S'_1,$ TVar$(s_1; s'_1)) \subseteq$ TVar(S_1). Similarly, Imp$(S'_2,$ TVar$(s_2; s'_2)) \subseteq$ TVar(S_2).

Then, by Theorem 2, after terminating execution of S'_1 and S'_2, value stores σ'_1 and σ'_2 agree on values of the termination deciding variables of $s_1; s'_1$ and $s_2; s'_2$, $\forall x \in$ TVar$(s_1; s'_1) \cup$ TVar$(s_2; s'_2)$: $\sigma'_1(x) = \sigma'_2(x)$.

We show that the execution of s_2 terminates by the hypothesis IH. By Corollary E.1, $(S'_1; s_1; s'_1, m_1(\sigma_1)) \xrightarrow{*} (s_1; s'_1, m'_1(\sigma'_1))$ and $(S'_2; s_2; s'_2, m_2(\sigma_2)) \xrightarrow{*} (s_2; s'_2, m'_2(\sigma'_2))$. By assumption, S_1 terminates, then s_1 terminates, $(s_1, m'_1(\sigma'_1)) \xrightarrow{*}$ (skip$, m''_1(\sigma''_1))$. Because $s'_1 \equiv^S_H s_2$, to apply the induction hypothesis, we need to show that all required conditions hold.
- size(s_2) + size$(s'_1) < k$.
  By definition, size$(S'_2) > 1$, size$(S'_1) > 1$, size$(s_1) > 1$, size$(s'_2) > 1$.
- Value stores σ''_1 and σ'_2 agree on values of the termination deciding variables of s'_1 and s_2. $\forall x \in$ TVar$(s'_1) \cup$ TVar(s_2) : $\sigma''_1(x) = \sigma'_2(x)$.
  By Corollary 5.1, TVar$(s'_1) =$ TVar(s_2). Because of the condition Def$(s_1) \cap$ TVar$(s'_1) = \emptyset$, by Corollary E.2, value stores σ''_1 and σ'_1 agree on values of the termination deciding variables of s'_1, $\forall x \in$ TVar(s'_1) : $\sigma''_1(x) = \sigma'_1(x)$. By the argument above, $\forall x \in$ TVar(s_2) : $\sigma'_1(x) = \sigma'_2(x)$. Thus, the condition holds.

By the induction hypothesis IH, $(s'_1, m''_1(\sigma''_1)) \equiv_H (s_2, m'_2(\sigma'_2))$. Because the execution of s'_1 terminates, then the exeuction of s_2 terminates when started in the state $m'_2(\sigma'_2)$, $(s_2, m'_2(\sigma'_2)) \xrightarrow{*}$ (skip$, m''_2(\sigma''_2))$.

We show that the execution of s'_2 terminates. This is by the similar argument that s_2 terminates.

In conclusion, S_2 terminates when started in the state $m_2(\sigma_2)$. The theorem holds.

In addition, we show that it is impossible that s_1 and s'_1 both include input statements by contradiction. Suppose there are input statements in both s_1 and s'_1. By Lemma 5.11, $id_I \in$ Def$(s_1) \cap$ TVar(s'_1). A contradiction against the condition that Def$(s_1) \cap$ TVar$(s'_1) = \emptyset$. Similarly, there are no input statements in both s_2 and s'_2.

□

5.3.3 Supporting lemmas for the soundness proof of termination in the same way

The supporting lemmas include various properties of TVar(S), two statement sequences satisfying the proof rule of termination in the same way consume the same number of input values when both terminate, and the proof for the case of while statement of theorem 4.

The properties of the termination deciding variables

Lemma 5.4. *The crash variables of $S_1; S'_1$ is same as the union of the crash variables of S_1 and the imported variables in S_1 relative to the crash variables of S'_1, CVar($S_1; S'_1$) = CVar(S_1) $\cup$ Imp(S_1, CVar (S'_1)).*

Proof. Let $S'_1 = s_1; ...; s_k$ for $k > 0$. We show the lemma by induction on k.

Base case.

By the definition of CVar(S), the lemma holds.

Induction step.

The hypothesis IH is that CVar($S_1; s_1; ...; s_k$) = CVar(S_1) $\cup$ Imp(S_1, CVar($s_1; ...; s_k$)) for $k \geq 1$.

Then we show that the lemma holds when $S'_1 = s_1; ...; s_{k+1}$.

$$\begin{aligned}&\text{CVar}(S_1; s_1; ...; s_{k+1})\\ =\ &\text{CVar}(S_1; s_1) \cup \text{Imp}(S_1; s_1, \text{CVar}(s_2; ...; s_{k+1})) \text{ by IH}\end{aligned}$$

$$\begin{aligned}&\text{CVar}(S_1; s_1)\\ =\ &\text{CVar}(S_1) \cup \text{Imp}(S_1, \text{CVar}(s_1)) \ (1)\end{aligned}$$

$$\begin{aligned}&\text{Imp}(S_1; s_1, \text{CVar}\ (s_2; ...; s_{k+1}))\\ =\ &\text{Imp}(S_1, \text{Imp}(s_1; \text{CVar}\ (s_2; ...; s_{k+1}))) \ (2)\end{aligned}$$

Combining (1) and (2), we have

$$\begin{aligned}&\text{CVar}(S_1; s_1) \cup \text{Imp}(S_1; s_1, \text{CVar}\ (s_2; ...; s_{k+1}))\\ =\ &\text{CVar}(S_1) \cup \text{Imp}\ (S_1, \text{CVar}\ (s_1))\\ &\cup \text{Imp}\ (S_1, \text{Imp}(s_1; \text{CVar}\ (s_2; ...; s_{k+1})))\\ =\ &\text{CVar}(S_1) \cup \text{Imp}(S_1, \text{CVar}(s_1) \cup \text{Imp}(s_1; \text{CVar}(s_2; ...; s_{k+1})))\\ &\text{by Lemma C.2}\\ =\ &\text{CVar}(S_1) \cup \text{Imp}(S_1, \text{CVar}(s_1; ...; s_{k+1})).\end{aligned}$$

□

Lemma 5.5. *The loop deciding variables of $S_1; S'_1$ is same as the union of the loop deciding variables of S_1 and the imported variables in S_1 relative to the loop deciding variables of S'_1, LVar($S_1; S'_1$) = LVar(S_1) $\cup$ Imp(S_1, LVar (S'_1)).*

By proof of Lemma 5.5 similar to that of lemma 5.4 above.

Lemma 5.6. *If two statement sequences S_1 and S_2 satisfy the proof rule of termination in the same way, then S_1 and S_2 have same loop variables, ($S_1 \equiv^S_H S_2$) $\Rightarrow$ (LVar(S_1) = LVar(S_2)).*

Proof. By induction on size(S_1)+size(S_2), the sum of the program size of S_1 and S_2.

Base case.

S_1 and S_2 are simple statement. There are three base cases according to the definition of $s_1 \equiv^S_H s_2$.

1. two same simple statements, $S_1 = S_2$;
2. S_1 and S_2 are input statement with same type variable: $S_1 =$ "input id_1", $S_2 =$ "input id_2" where $(\Gamma_{S_1} \vdash id_1 : t) \wedge (\Gamma_{S_2} \vdash id_2 : t)$;.
3. $S_1 =$ "output e" or "$id_1 := e$", $S_2 =$ "output e" or "$id_2 := e$" where both of the following hold:
   - There is no possible value mismatch in "$id_1 := e$", $\neg(\Gamma_{S_1} \vdash id_1 : \text{Int}) \vee \neg(\Gamma_{S_1} \vdash e : \text{Long}) \vee (\Gamma_{S_1} \vdash e : \text{Int})$.
   - There is no possible value mismatch in "$id_2 := e$", $\neg(\Gamma_{S_2} \vdash id_2 : \text{Int}) \vee \neg(\Gamma_{S_2} \vdash e : \text{Long}) \vee (\Gamma_{S_2} \vdash e : \text{Int})$.

By the definition of loop variables, LVar(S_1) = LVar(S_2) = $\emptyset$ in both base cases. Therefore, Lemma 5.6 holds.

Induction Step.

The hypothesis IH is that Lemma 5.6 holds when size(S_1) + size(S_2) = $k \geq 2$.

We show that Lemma 5.6 holds when size(S_1) + size(S_2) = $k + 1$.

The proof is a case analysis according to the cases in the definition of ($S_1 \equiv^S_H S_2$):

1. S_1 and S_2 are one statement and one of the following holds.
   (a) S_1 = "If(e) then $\{S^t_1\}$ else $\{S^f_1\}$", S_2 = "If(e) then $\{S^t_2\}$ else $\{S^f_2\}$" such that one of the following holds:
      i. $S^t_1, S^f_1, S^t_2, S^f_2$ are all sequences of "skip";
      By the definition of loop variables, LVar(S^t_1) = LVar(S^f_1) = LVar(S^t_2) = LVar(S^f_2) = $\emptyset$. Therefore, by the definition of loop variables, LVar(S_1) = LVar(S_2) = $\emptyset$. The lemma 5.6 holds.
      ii. At least one of $S^t_1, S^f_1, S^t_2, S^f_2$ is not a sequence of "skip" such that: $(S^t_1 \equiv^S_H S^t_2) \wedge (S^f_1 \equiv^S_H S^f_2)$; size($S^t_1$) + size($S^t_2$) $< k$, size(S^f_1) + size(S^f_2) $< k$.
      Then, by the hypothesis IH1, LVar(S^t_1) = LVar(S^t_2), LVar(S^f_1) = LVar(S^f_2). Consequently, (LVar(S^t_1) $\cup$ LVar(S^f_1)) = (LVar(S^t_2) $\cup$ LVar(S^f_2)) = LVar(Δ). When LVar(Δ) = $\emptyset$, then LVar(S_1) = LVar(S_2) = $\emptyset$ by the definition of loop variables. When LVar(Δ) $\neq \emptyset$, then LVar(S_1) = LVar(S_2) = LVar(Δ) $\cup$ Use(e) by the definition of loop variables. The lemma 5.6 holds.
   (b) S_1 = "while$_{\langle n_1 \rangle}$(e)$\{S''_1\}$", S_2 = "while$_{\langle n_2 \rangle}$(e)$\{S''_2\}$" such that both of the following hold:
      - $S''_1 \equiv^S_H S''_2$;
      - S''_1 and S''_2 have equivalent computation of TVar(S_1) $\cup$ TVar(S_2);

      By the hypothesis IH1, LVar(S''_1) = LVar(S''_2). Then Use(e) $\cup$ LVar(S''_1) = Use(e) $\cup$ LVar(S''_2). Then, we show that:
      $\forall i \geq 0$, Imp(S''^i_1, Use(e)$\cup$LVar(S''_1)) = Imp(S''^i_2, Use(e)$\cup$ LVar(S''_2)) by induction on i.
      Base case
      By our notation S^0, S''^0_1 = skip, S''^0_2 = skip. Then, by the definition of imported variables,
      Imp(S''^0_1, Use(e) $\cup$ LVar(S''_1)) = Use(e) $\cup$ LVar(S''_1),
      Imp(S''^0_2, Use(e) $\cup$ LVar(S''_2)) = Use(e) $\cup$ LVar(S''_2).
      Then, Imp(S''^0_1, Use(e)$\cup$LVar(S''_1)) = Imp(S''^0_2, Use(e)$\cup$ LVar(S''_2)).
      Induction step
      The hypothesis IH3 is that, $\forall i \geq 0$, Imp(S''^i_1, Use(e) $\cup$ LVar(S''_1)) = Imp(S''^i_2, Use(e) $\cup$ LVar(S''_2)).
      Then we show that Imp(S''^{i+1}_1, Use(e) $\cup$ LVar(S''_1)) = Imp(S''^{i+1}_2, Use(e) $\cup$ LVar(S''_2)).
      By Corollary C.1,
      Imp(S''^{i+1}_1, Use(e)$\cup$LVar(S''_1)) = Imp(S''_1, Imp(S''^i_1, Use(e)$\cup$ LVar(S''_1))),
      Imp(S''^{i+1}_2, Use(e)$\cup$LVar(S''_2)) = Imp(S''_2, Imp(S''^i_2, Use(e)$\cup$ LVar(S''_2))).
      By the hypothesis IH3, Imp(S''^i_1, Use(e) $\cup$ LVar(S''_1)) = Imp(S''^i_2, Use(e) $\cup$ LVar(S''_2)) = LVar(Δ).

Besides, by the definition of loop variables, LVar$(\Delta) \subseteq$ LVar(S_1), LVar$(\Delta) \subseteq$ LVar(S_2).
Then,

$$\begin{aligned}&\text{Imp}(S_1'', \text{Imp}(S_1''^i, \text{Use}(e) \cup \text{LVar}(S_1'')))\\ =\ & \text{Imp}(S_1'', \text{LVar}(\Delta))\\ =\ & \text{Imp}(S_2'', \text{LVar}(\Delta)) \text{ by Lemma 5.3}\\ =\ & \text{Imp}(S_2'', \text{Imp}(S_2''^i, \text{Use}(e) \cup \text{LVar}(S_2'')))\end{aligned}$$

In summary, LVar$(S_1))$ = LVar(S_2). The lemma 5.6 holds.

2. S_1 and S_2 are not both one statement and one of the following holds:
   (a) $S_1 = S_1'; s_1$ and $S_2 = S_2'; s_2$ such that all of the following hold:
   - $S_1' \equiv_H^S S_2'$;
   - S_1' and S_2' have equivalent computation of TVar$(s_1) \cup$ TVar(s_2);
   - $s_1 \equiv_H^S s_2$ where s_1 and s_2 are not "skip";

   By the hypothesis IH1, LVar(S_1') = LVar(S_2'), LVar(s_1) = LVar(s_2) = LVar(Δ). Besides,
   Imp$(S_1', \text{LVar}(s_1))$ = LVar$(S_2', \text{LVar}(s_2))$ by Lemma 5.3. Therefore, LVar(S_1) = LVar(S_2) by the definition of loop variables. The lemma 5.6 holds.

   (b) One last statement is "skip": w.l.o.g.,
   $\big((S_1 \equiv_H^S S_2') \wedge (s_2 = \text{"skip"})\big)$.
   By the hypothesis IH1, LVar(S_1) = LVar(S_2') . Besides, LVar$(s_2) = \emptyset$ by the definition of loop variables. Therefore, LVar(S_1) = LVar(S_2) = LVar$(S_2') \cup$ Imp$(S_2', \emptyset)$. The lemma 5.6 holds.

   (c) One last statement is a "duplicate" statement such that one of the following holds:
   W.l.o.g. $S_1 = S_1'; s_1'; S_1''; s_1$ and all of the following hold:
   - $S_1'; s_1'; S_1'' \equiv_H^S S_2$;
   - $s_1' \equiv_H^S s_1$;
   - Def$(s_1'; S_1'') \cap$ TVar$(s_1) = \emptyset$;
   - $s_2 \neq$ "skip";

   By the hypothesis IH, LVar$(S_1'; s_1'; S_1'')$ = LVar(S_2).
   Then, we show that LVar$(S_1'; s_1'; S_1'')$ = LVar$(S_1'; s_1'; S_1''; s_1)$.
   By the induction hypothesis IH, LVar(s_1') = LVar(s_1).

$$\begin{aligned}&\text{LVar}(S_1'; s_1'; S_1''; s_1)\\ =\ & \text{LVar}(S_1'; s_1'; S_1'') \cup \text{Imp}(S_1'; s_1'; S_1'', \text{LVar}(s_1))\\ & \text{by the definition of loop variables}\end{aligned}$$

$$\begin{aligned}&\text{Imp}(S_1'; s_1'; S_1'', \text{LVar}(s_1))\\ =\ & \text{Imp}(S_1', \text{Imp}(s_1'; S_1'', \text{LVar}(s_1))) \text{ by Lemma C.1}\\ =\ & \text{Imp}(S_1', \text{LVar}(s_1)) \text{ by Def}(s_1'; S_1'') \cap \text{TVar}(s_1) = \emptyset\\ =\ & \text{Imp}(S_1', \text{LVar}(s_1')) \text{ by LVar}(s_1') = \text{LVar}(s_1)\\ \subseteq\ & \text{LVar}(S_1'; s_1') \text{ by the definition of loop variables}\\ \subseteq\ & \text{LVar}(S_1'; s_1'; S_1'') \text{ by Lemma 5.5.}\end{aligned}$$

   In conclusion, LVar$(S_1'; s_1'; S_1''; s_1)$ = LVar$(S_1'; s_1'; S_1'')$. The lemma holds.

   (d) $S_1 = S_1'; s_1; s_1';$ and $S_2 = S_2'; s_2; s_2'$ where s_1 and s_2 are reordered and all of the following hold:
   - $S_1' \equiv_H^S S_2'$;
   - S_1' and S_2' have equivalent computation of TVar$(s_1; s_1') \cup$ TVar$(s_2; s_2')$.
   - $s_1 \equiv_H^S s_2'$;
   - $s_1' \equiv_H^S s_2$;
   - Def$(s_1) \cap$ TVar$(s_1') = \emptyset$;
   - Def$(s_2) \cap$ TVar$(s_2') = \emptyset$;

   By the hypothesis IH, LVar(S_1') = LVar(S_2'), LVar(s_1) = LVar(s_2'), LVar(s_1') = LVar(s_2).

   In the following, we show LVar(S_1) = LVar(S_2) in three steps.

   i. We show LVar$(S_1'; s_1; s_1')$ = LVar(S_1') $\cup$Imp$(S_1', \text{LVar}(s_1)) \cup$ Imp$(S_1', \text{LVar}(s_1'))$.

$$\begin{aligned}&\text{LVar}(S_1'; s_1; s_1')\\ =\ & \text{LVar}(S_1'; s_1) \cup \text{Imp}(S_1'; s_1, \text{LVar}(s_1'))\\ & \text{by the definition of loop variables}\end{aligned}$$

$$\begin{aligned}&\text{LVar}(S_1'; s_1)\\ =\ & \text{LVar}(S_1') \cup \text{Imp}(S_1', \text{LVar}(s_1)) \ (2)\\ & \text{by the definition of loop variables}\end{aligned}$$

$$\begin{aligned}&\text{Imp}(S_1'; s_1, \text{LVar}(s_1'))\\ =\ & \text{Imp}(S_1', \text{Imp}(s_1, \text{LVar}(s_1'))) \text{ by Lemma C.1}\\ =\ & \text{Imp}(S_1', \text{LVar}(s_1')) \ (3)\\ & \text{by the condition Def}(s_1) \cap \text{TVar}(s_1') = \emptyset\end{aligned}$$

   According to (2) and (3), LVar$(S_1'; s_1; s_1')$ = LVar$(S_1') \cup$ Imp$(S_1', \text{LVar}(s_1)) \cup$ Imp$(S_1', \text{LVar}(s_1'))$.
   Similarly, LVar$(S_2'; s_2; s_2')$ = LVar$(S_2') \cup$ Imp$(S_2', \text{LVar}(s_2)) \cup$ Imp$(S_2', \text{LVar}(s_2'))$.

   ii. We show that Imp$(S_1', \text{LVar}(s_1))$ = Imp$(S_2', \text{LVar}(s_2'))$. Imp$(S_2', \text{LVar}(s_2'))$.
   We need to show that LVar$(s_1) \subseteq$ LVar$(s_1; s_1')$ and LVar$(s_2') \subseteq$ LVar$(s_2; s_2')$. By the definition of loop variables, LVar$(s_1) \subseteq$ LVar$(s_1; s_1')$. By the definition of loop variables again, LVar$(s_2; s_2')$ = LVar$(s_2) \cup$ Imp$(s_2, \text{LVar}(s_2'))$. Because Def$(s_2) \cap$ TVar$(s_2') = \emptyset$, Imp$(s_2, \text{LVar}(s_2'))$ = LVar(s_2').
   By the induction hypothesis IH, LVar(s_1) = LVar(s_2').
   By Lemma 5.3, $\forall x \in$ LVar(s_1) = LVar(s_2'), Imp$(S_1', \{x\})$ = Imp$(S_2', \{x\})$. By Lemma C.2, Imp$(S_1', \text{LVar}(s_1))$ = Imp$(S_2', \text{LVar}(s_2'))$.

   iii. We show that Imp$(S_1', \text{LVar}(s_1'))$ = Imp$(S_2', \text{LVar}(s_2))$.
   By the similar argument that Imp$(S_1', \text{LVar}(s_1))$ = Imp$(S_2', \text{LVar}(s_2'))$.

   In conclusion, LVar$(S_1'; s_1; s_1')$ = LVar$(S_2'; s_2; s_2')$.

□

Lemma 5.7. *If two statement sequences S_1 and S_2 satisfy the proof rule of termination in the same way, then S_1 and S_2 have same crash variables, $(S_1 \equiv_H^S S_2) \Rightarrow$ (CVar(S_1) = CVar(S_2)).*

By proof similar to those for Lemma 5.6.

Corollary 5.1. *If two statement sequences S_1 and S_2 satisfy the proof rule of termination in the same way, then S_1 and S_2 have same termination deciding variables, $(S_1 \equiv_H^S S_2) \Rightarrow$ (TVar(S_1) = TVar(S_2)).*

By Lemma 5.6, and 5.7.

Properties of the input sequence variable

Lemma 5.8. *If there is no input statement in a statement sequence S, then the input sequence variable is not in the defined variables of S, ($\nexists$"input x" $\in S) \Rightarrow id_I \notin$ Def(S).*

Proof. By induction on abstract syntax of S. □

Lemma 5.9. *If there is no input statement in a statement sequence S, then the input sequence variable is not in the crash variables of S, ($\nexists$"input x" $\in S) \Rightarrow (id_I \notin$ CVar$(S))$.*

Proof. By induction on abstract syntax of S. □

Lemma 5.10. *If there is no input statement in a statement sequence S, then the input sequence variable is in the loop variables of S, $(\nexists$"input x" $\in S) \Rightarrow (id_I \notin LVar(S))$.*

Proof. By induction on abstract syntax of S. ☐

Corollary 5.2. *If there is no input statement in a statement sequence S, then the input sequence variable is in the termination deciding variables of S, $(\nexists$"input x" $\in S) \Rightarrow (id_I \notin TVar(S))$.*

By Lemma 5.9 and 5.10.

Lemma 5.11. *If there is one input statement in a statement sequence S, then the input sequence variable is in the crash variables and defined variables of S, $(\exists$"input x" $\in S) \Rightarrow (id_I \in CVar(S)) \wedge (id_I \in Def(S))$.*

Proof. By induction on abstract syntax of S. ☐

Lemma 5.12. *If there is one input statement in a statement sequence S, then the imported variables in S relative to the input sequence variable are a subset of the crash variables of S, $(\exists$"input x" $\in S) \Rightarrow (Imp(S, \{id_I\}) \subseteq CVar(S))$.*

Proof. By induction on abstract syntax of S.

1. $S =$ "input x".
   By the definition of CVar$(\cdot)$ and Imp$(\cdot)$, CVar$(S) =$ Imp$(S, \{id_I\}) = \{id_I\}$.
2. $S =$ "If(e) then $\{S_t\}$ else $\{S_f\}$".
   W.l.o.g., there is input statement in S_t, by the induction hypothesis, Imp$(S_t, \{id_I\}) \subseteq$ CVar(S_t). There are two subcases regarding if input statement is in S_f.
   (a) There is input statement in S_f.
   By the induction hypothesis, Imp$(S_f, \{id_I\}) \subseteq$ CVar(S_f). Hence, the lemma holds.
   (b) There is no input statement in S_f.
   By the definition of imported variables, Imp$(S_f, \{id_I\}) = \{id_I\}$. By Lemma 5.11, $id_I \in$ CVar(S_t). Therefore, the lemma holds.
3. $S =$ "while$_{\langle n\rangle}(e)\{S'\}$".
   By the induction hypothesis, Imp$(S', \{id_I\}) \subseteq$ CVar(S').
   By the definition of Imp$(\cdot)$, Imp$(S', \{id_I\}) = \bigcup_{i\geq 0}$ Imp$({S'}^i, \{id_I\} \cup$ Use$(e))$. By the definition of CVar$(\cdot)$, CVar$(S') = \bigcup_{i\geq 0}$ Imp$({S'}^i,$ CVar$(S') \cup$ Use$(e))$.
   By induction on i, we show that, $\forall i \geq 0$, Imp$({S'}^i, \{id_I\} \cup$ Use$(e)) \subseteq$ Imp$({S'}^i,$ CVar$(S') \cup$ Use$(e))$.
   Base case $i = 0$.
   By notation ${S'}^0 =$ skip.

   Imp$({S'}^0, \{id_I\} \cup$ Use$(e))$
   $= \{id_I\} \cup$ Use(e) by the definition of imported variables

   Imp$({S'}^0,$ CVar$(S') \cup$ Use$(e))$
   $=$ CVar$(S') \cup$ Use(e) by the definition of imported variables

   $id_I \subseteq$ CVar(S') (1) by Lemma 5.11

   Imp$({S'}^0, \{id_I\} \cup$ Use$(e))$
   $\subseteq$ Imp$({S'}^0,$ CVar$(S') \cup$ Use$(e))$ by Lemma C.2.

   Induction step.
   The hypothesis IH1 is that Imp$({S'}^i, \{id_I\} \cup$ Use$(e)) \subseteq$ Imp$({S'}^i,$ CVar$(S') \cup$ Use$(e))$ for $i > 0$.
   Then we show that Imp$({S'}^{i+1}, \{id_I\} \cup$ Use$(e))$
   $\subseteq$ Imp$({S'}^{i+1},$ CVar$(S') \cup$ Use$(e))$

   Imp$({S'}^i, \{id_I\} \cup$ Use$(e))$
   $\subseteq$ Imp$({S'}^i,$ CVar$(S') \cup$ Use$(e))$ (1) by the hypothesis IH1

   Imp$({S'}^{i+1}, \{id_I\} \cup$ Use$(e))$
   $=$ Imp$(S',$ Imp$({S'}^i, \{id_I\} \cup$ Use$(e)))$ (2) by Corollary C.1

   Imp$({S'}^{i+1},$ CVar$(S') \cup$ Use$(e))$
   $=$ Imp$(S',$ Imp$({S'}^i,$ CVar$(S') \cup$ Use$(e)))$ (3) by Corollary C.1.

   Combining (1), (2) and (3):

   Imp$(S',$ Imp$({S'}^i, \{id_I\} \cup$ Use$(e)))$
   $\subseteq$ Imp$(S',$ Imp$({S'}^i,$ CVar$(S') \cup$ Use$(e)))$ by Lemma C.2.

   Therefore, Imp$({S'}^{i+1}, \{id_I\} \cup$Use$(e)) \subseteq$ Imp$({S'}^{i+1},$ CVar$(S') \cup$ Use$(e))$.
   In conclusion, Imp$(S, \{id_I\}) \subseteq$ CVar(S).
4. $S = s_1; ...; s_k$, for $k > 0$.
   By induction on k.
   Base case. $k = 1$.
   By above cases, the lemma holds.
   Induction step.
   The induction hypothesis IH2 is that the lemma holds when $k > 0$. We show that the lemma holds when $S = s_1; ...; s_{k+1}$.
   By the definition of crash variables, CVar$(s_1; ...; s_{k+1}) =$ CVar$(s_1; ...; s_k) \cup$ Imp$(s_1; ...; s_k,$ CVar$(s_{k+1}))$.There are two possibilities.
   (a) $\nexists$"input x" $\in s_{k+1}$.
   By Lemma 5.8, $id_I \notin$ Def(S).
   Imp$(s_1; ...; s_{k+1}, \{id_I\})$
   $=$ Imp$(s_1; ...; s_k, \{id_I\})$ by $id_I \notin$ Def(S) and the definition of imported variables
   $\subseteq$ CVar$(s_1; ...; s_k)$ by the hypothesis IH2
   $\subseteq$ CVar$(s_1; ...; s_{k+1})$ by the definition of crash variables

   (b) $\exists$"input x" $\in s_{k+1}$.
   Imp$(s_1; ...; s_{k+1}, \{id_I\})$
   $=$ Imp$(s_1; ...; s_k,$ Imp$(s_{k+1}, \{id_I\}))$
   by the definition of imported variables

   Imp$(s_{k+1}, \{id_I\})$
   $\subseteq$ CVar(s_{k+1}) by the hypothesis IH2

   Imp$(s_1; ...; s_k,$ Imp$(s_{k+1}, \{id_I\}))$
   $\subseteq$ Imp$(s_1; ...; s_k,$ CVar$(s_{k+1}))$ by Lemma C.2
   $\subseteq$ CVar$(s_1; ...; s_{k+1})$ by the definition of crash variables.

☐

Lemma 5.13. *If two programs S_1 and S_2 satisfy the proof rule of termination in the same way, then S_1 and S_2 satisfy the proof rule of terminating computation in the same way of the input sequence, $(S_1 \equiv_H^S S_2) \Rightarrow (S_1 \equiv_{id_I}^S S_2)$.*

Proof. By induction on size$(S_1) +$ size(S_2).
Base case. S_1 and S_2 are simple statements.
There are three cases.

1. S_1 and S_2 are "skip": S_1 = S_2 = "skip";
2. S_1 and S_2 are input statement: $S_1 =$ "input id_1", $S_2 =$ "input id_2";

3. s_1 and s_2 are with the same expression: s_1 = "output e" or "$id_1 := e$", s_2 = "output e" or "$id_2 := e$".
   By definition of the proof rule of equivalent computation, the lemma holds in above three cases.

Induction step.

The hypothesis IH is that the lemma holds when size(S_1) + size$(S_2) = k \geq 2$.

Then, we show that the lemma holds when size(S_1)+size$(S_2) = k + 1$. The proof is a case analysis of the cases in the proof rule of termination in the same way.

1. S_1 and S_2 are one statement and one of the followings holds.
   (a) S_1 = "If(e) then $\{S_1^t\}$ else $\{S_1^f\}$", S_2 = "If(e) then $\{S_2^t\}$ else $\{S_2^f\}$" and one of the followings holds:
      i. $S_1^t, S_1^f, S_2^t, S_2^f$ are all sequences of "skip";
         By Lemma 5.8, $id_I \notin$ Def$(S_1) \cap$ DefS_2. The lemma holds.
      ii. At least one of $S_1^t, S_1^f, S_2^t, S_2^f$ is not a sequence of "skip" such that:
         $(S_1^t \equiv_H^S S_2^t) \wedge (S_1^f \equiv_H^S S_2^f)$;
         Because size$(S_1) = 1+$size(S_1^t)+size(S_1^f), size$(S_2) = 1+$size(S_2^t)+size(S_2^f). Therefore, size(S_1^t)+size$(S_2^t) < k$, size(S_1^f) + size$(S_2^f) < k$. By the induction hypothesis IH, $(S_1^t \equiv_{id_I}^S S_2^t) \wedge (S_1^f \equiv_{id_I}^S S_2^f)$. Then, the lemma holds by the definition of $S_1 \equiv_{id_I}^S S_2$.
   (b) S_1 = "while$_{\langle n_1 \rangle}(e)\{S_1''\}$", S_2 = "while$_{\langle n_2 \rangle}(e)\{S_2''\}$" and both of the followings hold:
      - $S_1'' \equiv_H^S S_2''$;
      - S_1'' and S_2'' have equivalent computation of TVar$(S_1) \cup$ TVar(S_2);

      By the induction hypothesis IH, $S_1'' \equiv_{id_I}^S S_2''$. In addition, by Corollary 5.1, TVar(S_1) = TVar(S_2). There are two cases.
      i. $id_I \in$ TVar(S_1) = TVar(S_2).
         We show that $id_I \in$ Def$(S_1) \cap$ Def(S_2). If there is no input statement in S_1 or S_2, then, by Corollary 5.2, $id_I \notin$ TVar$(S_1) \cap$ TVar(S_2). A contradiction. Thus, there is input statement in S_1 and S_2, by Lemma 5.11, $id_I \in$ Def$(S_1) \cap$ Def(S_2).
         By Lemma 5.12, Imp$(S_1, \{id_I\}) \subseteq$ CVar(S_1). Similarly, Imp$(S_2, \{id_I\}) \subseteq$ TVar(S_2). Hence, loop bodies of S_1 and S_2 equivalently compute every of the imported variables in S_1 and S_2 relative to the input sequence variable, $\forall x \in$ Imp$(S_1, \{id_I\})\cup$Imp$(S_2, \{id_I\})$, $S_1'' \equiv_x^S S_2''$. Thus, the lemma holds.
      ii. $id_I \notin$ TVar(S_1) = TVar(S_2).
         Then there is no input statement in S_1 and S_2. Otherwise, by Lemma 5.11, $(id_I \in$ CVar$(S_1)) \vee (id_I \in$ CVar$(S_2))$. A contradiction. Then by Lemma 5.8, $id_I \notin$ (Def$(S_1) \cap$ Def(S_2)). Hence, the lemma holds.
2. S_1 and S_2 are not both one statement and one of the followings holds:
   (a) $S_1 = S_1'; s_1$ and $S_2 = S_2'; s_2$ and all of the followings hold:
      - $S_1' \equiv_H^S S_2'$;
      - S_1' and S_2' have equivalent computation of TVar$(s_1) \cup$ TVar(s_2);
      - $s_1 \equiv_H^S s_2$ where s_1 and s_2 are not "skip";

      By the induction hypothesis IH, $s_1 \equiv_{id_I}^S s_2$. By Corollary 5.1, TVar(s_1) = TVar(s_2). There are two cases.
      i. $id_I \in$ TVar(s_1) = TVar(s_2)
         Then there is input statement in s_1 and s_2. Otherwise, by Lemma 5.2, $id_I \notin$ TVar(s_1) = TVar(s_2). A contradiction. Then, by Lemma 5.11, $id_I \in$ Def$(s_1) \cap$ Def(s_2).
         By Lemma 5.3, Imp$(s_1, \{id_I\})$ = Imp$(s_2, \{id_I\})$.
         By Lemma 5.12, Imp$(s_1, \{id_I\}) \subseteq$ CVar$(s_1, \{id_I\})$.
         Therefore, S_1' and S_2' equivalently compute Imp$(s_1, \{id_I\})\cup$ Imp$(s_2, \{id_I\})$. The lemma holds.
      ii. $id_I \notin$ TVar(s_1) = TVar(s_2).
         Then, there is no input statement in s_1 and s_2. Otherwise, by Lemma 5.11, $id_I \in$ CVar(s_1) = CVar(s_2). A contradiction. Then, by Lemma 5.8, $id_I \notin$ Def$(s_1) \cup$ Def(s_2). By the induction hypothesis IH, $S_1' \equiv_{id_I}^S S_2'$. The lemma holds.
   (b) One last statement is "skip": W.l.o.g., $\big((S_1' \equiv_H^S S_2) \wedge (s_1 =$ "skip"$)\big)$
      By the induction hypothesis, $S_1' \equiv_{id_I}^S S_2$. By definition, $id_I \notin$ Def(s_1). The lemma holds.
   (c) One last statement is a "duplicate" statement such that one of the followings holds:
      W.l.o.g., $S_1 = S_1'; s_1'; S_1''; s_1$ and all of the followings hold:
      - $S_1'; s_1'; S_1'' \equiv_H^S S_2$;
      - $s_1' \equiv_H^S s_1$;
      - Def$(s_1'; S_1'') \cap$ TVar$(s_1) = \emptyset$;
      - $s_2 \neq$ "skip".

      By the induction hypothesis, $S_1'; s_1'; S_1'' \equiv_{id_I}^S S_2$. In the proof of Theorem 3, there is no input statement in s_2. Because $\forall x$: "input x" $\notin s_2$, by Lemma 5.8, $id_I \notin$ Def(s_1). The lemma holds.
   (d) $S_1 = S_1'; s_1; s_1'$; and $S_2 = S_2'; s_2; s_2'$ where s_1 and s_2 are reordered and all of the followings hold:
      - $S_1' \equiv_H^S S_2'$;
      - S_1' and S_2' have equivalent computation of TVar$(s_1; s_1')\cup$ TVar$(s_2; s_2')$.
      - $s_1 \equiv_H^S s_2'$;
      - $s_1' \equiv_H^S s_2$;
      - Def$(s_1) \cap$ TVar$(s_1') = \emptyset$;
      - Def$(s_2) \cap$ TVar$(s_2') = \emptyset$;

      In the proof of Theorem 4, we showed that s_1 and s_2 do not both include input statement, s_2 and s_2' do not both include input statement. There are two subcases.
      i. There are no input statements in both s_1 and s_1'.
         We show that there are no input statements in both s_2 and s_2'. By Corollary 5.1, TVar(s_1) = TVar(s_2') and TVar(s_1') = TVar(s_2). By Corollary 5.2, $id_I \notin$ TVar$(s_1)\cup$TVar(s_1'). Thus, $id_I \notin$ TVar$(s_2)\cup$TVar(s_2').
         If there is input statement in s_2 or s_2', then, by Lemma 5.11, $id_I \notin$ TVar$(s_2) \cup$ TVar(s_2'). A contradiction. In summary, there are no input statements in both s_2 and s_2'.
         By Lemma 5.8, $id_I \notin$ Def$(s_1; s_1')$ and $id_I \notin$ Def$(s_2; s_2')$.
         By the induction hypothesis, $S_1' \equiv_{id_I}^S S_2'$. Therefore, the lemma holds.
      ii. W.l.o.g, there are input statements in s_1 only.
         By similar argument in the proof of Theorem 4 that s_1 and s_2 do not both include input statements, we can show that there is no input statement in s_2 and there is input statement in s_2'.
         In the following, the proof is of two steps.
         A. We show that $s_1; s_1' \equiv_{id_I}^S s_2'$.
            By the induction hypothesis IH, $s_1 \equiv_{id_I}^S s_2'$. Because there is no input statement in s_1', then by

Lemma 5.8, $id_I \notin$ Def(s_1'). Thus, $s_1; s_1' \equiv^S_{id_I} s_2'$ by definition.

B. We show that S_1' and $S_2'; s_2$ equivalently compute Imp$(s_1; s_1', \{id_I\}) \cup$ Imp$(s_2', \{id_I\})$.
The argument is of two parts. First, we need to show that Def$(s_2) \cap$ Imp$(s_2', \{id_I\}) = \emptyset$. By Lemma 5.12, Imp$(s_2', \{id_I\}) \subseteq$ CVar(s_2'). Thus, Imp$(s_2', \{id_I\}) \subseteq$ TVar(s_2'). By assumption, Def$(s_2) \cap$ TVar$(s_2') = \emptyset$. Then, Def$(s_2) \cap$ Imp$(s_2', \{id_I\}) = \emptyset$.
By Lemma 5.3, Imp$(s_1; s_1', \{id_I\}) =$ Imp$(s_2', \{id_I\})$.
Thus, Def$(s_2) \cap$ Imp$(s_1; s_1', \{id_I\}) = \emptyset$.
Second, we show that Imp$(s_2', \{id_I\}) \subseteq$ TVar$(s_2; s_2')$ and Imp$(s_1; s_1', \{id_I\}) \subseteq$ TVar$(s_1; s_1')$. By Lemma 5.12, Imp$(s_1; s_1', \{id_I\}) \subseteq$ TVar$(s_1; s_1')$ and Imp$(s_2', \{id_I\}) \subseteq$ TVar(s_2'). Then we show that TVar$(s_2') \subseteq$ TVar$(s_2; s_2')$.
We need to show that CVar$(s_2') \subseteq$ CVar$(s_2; s_2')$ and LVar$(s_2') \subseteq$ LVar$(s_2; s_2')$.

$$\begin{aligned}
& \text{CVar}(s_2; s_2') \\
= {} & \text{CVar}(s_2) \cup \text{Imp}(s_2, \text{CVar}(s_2')) \\
& \text{by the definition of crash variables}
\end{aligned}$$

$$\begin{aligned}
& \text{Imp}(s_2, \text{CVar}(s_2')) \\
= {} & \text{CVar}(s_2') \text{ (1) by the assumption} \\
& \text{Def}(s_2) \cap \text{TVar}(s_2') = \emptyset
\end{aligned}$$

$$\begin{aligned}
& \text{CVar}(s_2) \cup \text{Imp}(s_2, \text{CVar}(s_2')) \\
= {} & \text{CVar}(s_2) \cup \text{CVar}(s_2') \text{ by (1)}
\end{aligned}$$

Similarly, LVar$(s_2') \subseteq$ LVar$(s_2; s_2')$. Thus, TVar$(s_2') \subseteq$ TVar$(s_2; s_2')$.
By assumption, $\forall x \in$ Imp$(s_1; s_1', \{id_I\}) \cup$ Imp$(s_2', \{id_I\})$: $S_1' \equiv^S_x S_2'$. In addition, Def$(s_2) \cap ($Imp$(s_1; s_1', \{id_I\}) \cup$ Imp$(s_2', \{id_I\})) = \emptyset$. Thus, $\forall x \in$ Imp$(s_1; s_1', \{id_I\}) \cup$ Imp$(s_2', \{id_I\})$: $S_1' \equiv^S_x S_2'; s_2$. The lemma holds.

□

Lemma 5.14. *If two programs S_1 and S_2 satisfy the proof rule of termination in the same way, and S_1 and S_2 both terminate when started in their initial states with crash flags not set, $\mathfrak{f}_1 = \mathfrak{f}_2 = 0$, whose value stores agree on values of variables of the termination deciding variables of S_1 and S_2, $\forall x \in TVar(S_1) \cup TVar(S_2), \sigma_1(x) = \sigma_2(x)$, and S_1 and S_2 are fed with the same infinite input sequence, $\sigma_1(id_I) = \sigma_2(id_I)$, $(S_1, m_1(\mathfrak{f}_1, \sigma_1) \xrightarrow{*} (skip, m_1'(\sigma_1'))$ and $(S_2, m_2(\mathfrak{f}_2, \sigma_2) \xrightarrow{*} (skip, m_2'(\sigma_2'))$, then the execution of S_1 and S_2 consume the same number of input values, $\sigma_1'(id_I) = \sigma_2'(id_I)$.*

Proof. By Lemma 5.13, $S_1 \equiv^S_{id_I} S_2$. By Lemma 5.12, Imp$(S_1, id_I) \subseteq$ CVar(S_1) and Imp$(S_2, id_I) \subseteq$ CVar(S_2). By assumption, $\forall x \in$ Imp$(S_1, id_I) \cup$ Imp(S_2, id_I) : $\sigma_1(x) = \sigma_2(x)$. By Theorem 2, $\sigma_1'(id_I) = \sigma_2'(id_I)$. □

Theorem of two loop statements terminating in the same way

Lemma 5.15. *Let s_1 = "while$_{\langle n_1 \rangle}(e)\{S_1\}$" and s_2 = "while$_{\langle n_2 \rangle}(e)\{S_2\}$" be two while statements with the same set of termination deciding variables in program P_1 and P_2 respectively, whose bodies S_1 and S_2 satisfy the proof rule of equivalently computation of variables in TVar(s), and S_1 and S_2 terminate in the same way when started in states with crash flags not set and agreeing on values of variables in TVar$(S_1) \cup$ TVar(S_2):*

- *$TVar(s_1) = TVar(s_2) = TVar(s)$;*
- *$\forall x \in TVar(s)$: $S_1 \equiv^S_x S_2$;*
- $\forall m_{S_1}(\mathfrak{f}_{S_1}, \sigma_{S_1})\, m_{S_2}(\mathfrak{f}_{S_2}, \sigma_{S_2})$:
$(((\forall z \in TVar(S_1) \cup TVar(S_2)), \sigma_{S_1}(z) = \sigma_{S_2}(z)) \wedge (\mathfrak{f}_{S_1} = \mathfrak{f}_{S_2} = 0)) \Rightarrow$
$(S_1, m_{S_1}(\mathfrak{f}_{S_1}, \sigma_{S_1})) \equiv_H (S_2, m_{S_2}(\mathfrak{f}_{S_2}, \sigma_{S_2}))$.

If s_1 and s_2 start in the state $m_1(\mathfrak{f}_1, loop^1_c, \sigma_1)$ and $m_2(\mathfrak{f}_2, loop^2_c, \sigma_2)$ respectively in which crash flags are not set, $\mathfrak{f}_1 = \mathfrak{f}_2 = 0$, s_1 and s_2 have not already executed, $loop^1_c(n_1) = loop^2_c(n_2) = 0$, value stores σ_1 and σ_2 agree on values of variables in TVar(s), $\forall x \in TVar(s)$: $\sigma_1(x) = \sigma_2(x)$, then, for any positive integer i, one of the following holds:

1. *The loop counters for s_1 and s_2 are less than i where s_1 and s_2 terminate in the same way:*
$\forall m_1'\, m_2'$: $(s_1, m_1) \xrightarrow{*} (S_1', m_1'(loop^{1'}_c))$ and $(s_2, m_2) \xrightarrow{*} (S_2', m_2'(loop^{2'}_c))$, $loop^{1'}_c(n_1) < i$ and $m^{2'}_c(n_2) < i$ and one of the following holds:
   (a) s_1 and s_2 both terminate:
   $(s_1, m_1) \xrightarrow{*} (skip, m_1'')$ and $(s_2, m_2) \xrightarrow{*} (skip, m_2'')$.
   (b) s_1 and s_2 both do not terminate:
   $\forall k > 0$: $(s_1, m_1) \xrightarrow{k} (S_{1_k}, m_{1_k})$ and $(s_2, m_2) \xrightarrow{k} (S_{2_k}, m_{2_k})$ in which $S_{1_k} \neq skip$, $S_{2_k} \neq skip$.
2. *The loop counters for s_1 and s_2 are less than or equal to i where s_1 and s_2 do not terminate such that there are no configurations (s_1, m_{1_i}) and (s_2, m_{2_i}) reachable from (s_1, m_1) and (s_2, m_2), respectively, in which crash flags are not set, the loop counters of s_1 and s_2 are equal to i, and value stores agree on the values of variables in TVar(s):*
   - *$\forall m_1'\, m_2'$: $(s_1, m_1) \xrightarrow{*} (S_1, m_1'(loop^{1'}_c)), (s_2, m_2) \xrightarrow{*} (S_2, m_2'(loop^{2'}_c))$ where $loop^{1'}_c(n_1) \leq i, loop^{2'}_c(n_2) \leq i$;*
   - *$\forall k > 0$:*
   $(s_1, m_1) \xrightarrow{k} (S_{1_k}, m_{1_k}), (s_2, m_2) \xrightarrow{k} (S_{2_k}, m_{2_k})$ where $S_{1_k} \neq skip, S_{2_k} \neq skip$; and
   - *$\nexists (s_1, m_{1_i})\, (s_2, m_{2_i})$:*
   $(s_1, m_1) \xrightarrow{*} (s_1, m_{1_i}(\mathfrak{f}_1, loop^{1_i}_c, \sigma_{1_i})) \wedge$
   $(s_2, m_2) \xrightarrow{*} (s_2, m_{2_i}(\mathfrak{f}_2, loop^{2_i}_c, \sigma_{2_i}))$ where
     - *$\mathfrak{f}_1 = \mathfrak{f}_2 = 0$; and*
     - *$loop^{1_i}_c(n_1) = loop^{2_i}_c(n_2) = i$; and*
     - *$\forall x \in TVar(s)$: $\sigma_{1_i}(x) = \sigma_{2_i}(x)$.*
3. *There are two configurations (s_1, m_{1_i}) and (s_2, m_{2_i}) reachable from (s_1, m_1) and (s_2, m_2), respectively, in which both crash flags are not set, the loop counters of s_1 and s_2 are equal to i and value stores agree on the values of variables in TVar(s), and for every state in execution $(s_1, m_1) \xrightarrow{*} (s_1, m_{1_i})$ or $(s_2, m_2) \xrightarrow{*} (s_2, m_{2_i})$, the loop counters for s_1 and s_2 are less than or equal to i respectively:*
$\exists (s_1, m_{1_i})\, (s_2, m_{2_i})$: $(s_1, m_1) \xrightarrow{*} (s_1, m_{1_i}(\mathfrak{f}_1, loop^{1_i}_c, \sigma_{1_i})) \wedge$
$(s_2, m_2) \xrightarrow{*} (s_2, m_{2_i}(\mathfrak{f}_2, loop^{2_i}_c, \sigma_{2_i}))$ where
   - *$\mathfrak{f}_1 = \mathfrak{f}_2 = 0$; and*
   - *$loop^{1_i}_c(n_1) = loop^{2_i}_c(n_2) = i$; and*
   - *$\forall x \in TVar(s)$: $\sigma_{1_i}(x) = \sigma_{2_i}(x)$; and*
   - *$\forall m_1'$: $(s_1, m_1) \xrightarrow{*} (S_1', m_1'(m^{1'}_c)) \xrightarrow{*} (s_1, m_{1_i})$, $loop^{1'}_c(n_1) \leq i$; and*
   - *$\forall m_2'$: $(s_2, m_2) \xrightarrow{*} (S_2', m_2'(m^{2'}_c)) \xrightarrow{*} (s_2, m_{2_i})$, $loop^{2'}_c(n_2) \leq i$;*

Proof. By induction on i.
Base case. $i = 1$.

By assumption, initial loop counters of s_1 and s_2 are of value zero. Initial value stores σ_1 and σ_2 agree on the values of variables in TVar(s). Then we show one of the following cases hold:

1. The loop counters for s_1 and s_2 are less than 1, s_1 and s_2 terminate in the same way:
   $\forall m_1' \, m_2'$ such that $(s_1, m_1) \xrightarrow{*} (S_1', m_1'(\text{loop}_c^{1'}))$ and $(s_2, m_2) \xrightarrow{*} (S_2', m_2'(\text{loop}_c^{2'}))$,
   $m_c^{1'}(n_1) < 1$ and $m_c^{2'}(n_2) < 1$ and one of the following holds:
   (a) s_1 and s_2 both terminate:
   $(s_1, m_1) \xrightarrow{*} (\text{skip}, m_1'')$ and $(s_2, m_2) \xrightarrow{*} (\text{skip}, m_2'')$.
   (b) s_1 and s_2 both do not terminate:
   $\forall k > 0, (s_1, m_1) \xrightarrow{k} (S_{1_k}, m_{1_k})$ and $(s_2, m_2) \xrightarrow{k} (S_{2_k}, m_{2_k})$ in which $S_{1_k} \neq \text{skip}, S_{2_k} \neq \text{skip}$.
2. The loop counters for s_1 and s_2 are less than or equal to 1, and s_1 and s_2 do not terminate such that there are no configurations (s_1, m_{1_1}) and (s_2, m_{2_1}) reachable from (s_1, m_1) and (s_2, m_2), respectively, in which crash flags are not set, the loop counters of s_1 and s_2 are equal to 1 and value stores agree on the values of variables in TVar(s):
   - $\forall m_1' \, m_2' \; : \; (s_1, m_1) \xrightarrow{*} (S_1, m_1'(\text{loop}_c^{1'})), (s_2, m_2) \xrightarrow{*} (S_2, m_2'(\text{loop}_c^{2'}))$ where $\text{loop}_c^{1'}(n_1) \leq i, \text{loop}_c^{2'}(n_2) \leq i$;
   - $\forall k > 0 \; : \; (s_1, m_1) \xrightarrow{k} (S_{1_k}, m_{1_k}), (s_2, m_2) \xrightarrow{k} (S_{2_k}, m_{2_k})$ where $S_{1_k} \neq \text{skip}, S_{2_k} \neq \text{skip}$; and
     $\nexists (s_1, m_{1_1}), (s_2, m_{2_1}) \; : \; (s_1, m_1) \xrightarrow{*} (s_1, m_{1_1}(\mathfrak{f}_1, \text{loop}_c^{1_1}, \sigma_{1_1})) \wedge (s_2, m_2) \xrightarrow{*} (s_2, m_{2_1}(\mathfrak{f}_2, \text{loop}_c^{2_1}, \sigma_{2_1}))$ where
     - $\mathfrak{f}_1 = \mathfrak{f}_2 = 0$; and
     - $\text{loop}_c^{1_1}(n_1) = \text{loop}_c^{2_1}(n_2) = 1$; and
     - $\forall x \in \text{TVar}(s) \; : \; \sigma_{1_1}(x) = \sigma_{2_1}(x)$.
3. There are two configurations (s_1, m_{1_1}) and (s_2, m_{2_1}) reachable from (s_1, m_1) and (s_2, m_2) respectively, in which the loop counters of s_1 and s_2 are equal to 1 and value stores agree on the values of variables in TVar(s) and, for every state in execution, $(s_1, m_1) \xrightarrow{*} (s_1, m_{1_1})$ or $(s_2, m_2) \xrightarrow{*} (s_2, m_{2_1})$ the loop counters for s_1 and s_2 are less than or equal to 1 respectively:
   $\exists (s_1, m_{1_1}) \, (s_2, m_{2_1}) \; : \; (s_1, m_1) \xrightarrow{*} (s_1, m_{1_1}(\mathfrak{f}_1, \text{loop}_c^{1_1}, \sigma_{1_1})) \wedge (s_2, m_2) \xrightarrow{*} (s_2, m_{2_1}(\mathfrak{f}_2, \text{loop}_c^{2_1}, \sigma_{2_1}))$ where
   - $\mathfrak{f}_1 = \mathfrak{f}_2 = 0$; and
   - $\text{loop}_c^{1_1}(n_1) = \text{loop}_c^{2_1}(n_1) = 1$; and
   - $\forall x \in \text{TVar}(s) \; : \; \sigma_{1_1}(x) = \sigma_{2_1}(x)$; and
   - $\forall m_1' \; : \; (s_1, m_1) \xrightarrow{*} (S_1', m_1'(\text{loop}_c^{1'})) \xrightarrow{*} (s_1, m_{1_1}), \; \text{loop}_c^{1'}(n_1) \leq 1$; and
   - $\forall m_2' \; : \; (s_2, m_2) \xrightarrow{*} (S_2', m_2'(\text{loop}_c^{2'})) \xrightarrow{*} (s_2, m_{2_1}), \; \text{loop}_c^{2'}(n_2) \leq 1$.

We show evaluations of the predicate expression of s_1 and s_2 w.r.t value stores σ_1 and σ_2 produce same value. By the definition of loop variables, LVar$(s_1) = \bigcup_{j \geq 0} \text{Imp}(S_1^j, \text{LVar}(S_1) \cup \text{Use}(e))$. By our notation of S^0, $S_1^0 = \text{skip}$. By the definition of loop variables, Use$(e) \subseteq$ LVar(s) = LVar(s_1). By assumption, value stores σ_1 and σ_2 agree on the values of the variables in Use(e). By Lemma D.1, the predicate expression e of s_1 and s_2 evaluates to same value v w.r.t value stores σ_1, σ_2, $\mathcal{E}'[\![e]\!]\sigma_1 = \mathcal{E}'[\![e]\!]\sigma_2$. Then there are three possibilities.

1. $\mathcal{E}'[\![e]\!]\sigma_1 = \mathcal{E}'[\![e]\!]\sigma_2 = (\text{error}, *)$
   The execution from $(s_1, m_1(\mathfrak{f}_1, \text{loop}_c^1, \sigma_1))$ proceeds as follows.

   $(s_1, m_1(\mathfrak{f}_1, \text{loop}_c^1, \sigma_1))$
   $= (\text{while}_{\langle n_1 \rangle}(e) \; \{S_1\}, m_1(\mathfrak{f}_1, \text{loop}_c^1, \sigma_1))$
   $\rightarrow (\text{while}_{\langle n_1 \rangle}((\text{error}, *)) \; \{S_1\}, m_1(\mathfrak{f}_1, \text{loop}_c^1, \sigma_1))$ by the EEval' rule
   $\rightarrow (\text{while}_{\langle n_1 \rangle}(0) \; \{S_1\}, m_1(1/\mathfrak{f}_1))$ by the ECrash rule
   $\xrightarrow{k} (\text{while}_{\langle n_1 \rangle}(0) \; \{S_1\}, m_1(1/\mathfrak{f}_1, \text{loop}_c^1, \sigma_1))$,
   for any $k \geq 0$, by the Crash rule.

   Similarly, the execution of s_2 started in the state $m_2(\mathfrak{f}_2, \text{loop}_c^2, \sigma_2)$ does not terminate.
   The loop counters of s_1 and s_2 are less than 1:
   $\forall m_1' \, m_2' \; : \; (s_1, m_1) \xrightarrow{*} (S_1', m_1'(\text{loop}_c^{1'}))$ and $(s_2, m_2) \xrightarrow{*} (S_2', m_2'(\text{loop}_c^{2'}))$ where $\text{loop}_c^{1'}(n_1) < 1$ and $\text{loop}_c^{2'}(n_2) < 1$.
   Besides, s_1 and s_2 both do not terminate when started in states m_1 and m_2, $\forall k > 0 \; : \; (s_1, m_1) \xrightarrow{k} (S_{1_k}, m_{1_k})$ and $(s_2, m_2) \xrightarrow{k} (S_{2_k}, m_{2_k})$ in which $S_{1_k} \neq \text{skip}, S_{2_k} \neq \text{skip}$.
2. $\mathcal{E}'[\![e]\!]\sigma_1 = \mathcal{E}'[\![e]\!]\sigma_2 = (0, v_{\mathfrak{of}})$
   The execution of s_1 proceeds as follows.

   $(s_1, m_1(\text{loop}_c^1, \sigma_1))$
   $= (\text{while}_{\langle n_1 \rangle}(e) \; \{S_1\}, m_1(\text{loop}_c^1))$
   $\rightarrow (\text{while}_{\langle n_1 \rangle}((0, v_{\mathfrak{of}})) \; \{S_1\}, m_1(\text{loop}_c^1))$ by the EEval' rule
   $\rightarrow (\text{while}_{\langle n_1 \rangle}(0) \; \{S_1\}, m_1(\text{loop}_c^1))$ by the E-Oflow1 or E-Oflow2 rule
   $\rightarrow (\text{skip}, m_1)$ by the Wh-F1 rule.

   Similarly, $(s_2, m_2(\text{loop}_c^2, \sigma_2)) \xrightarrow{2} (\text{skip}, m_2)$.
   The loop counters for s_1 and s_2 are less than 1:
   $\forall m_1' \, m_2' \; : \; (s_1, m_1) \xrightarrow{*} (S_1', m_1'(\text{loop}_c^{1'}))$ and $(s_2, m_2) \xrightarrow{*} (S_2', m_2'(\text{loop}_c^{2'}))$ where $\text{loop}_c^{1'}(n_1) < 1$ and $\text{loop}_c^{2'}(n_2) < 1$.
   Besides, s_1 and s_2 both terminate when started in states m_1 and m_2:
   $(s_1, m_1) \xrightarrow{*} (\text{skip}, m_1'')$ and $(s_2, m_2) \xrightarrow{*} (\text{skip}, m_2'')$.
3. $\mathcal{E}'[\![e]\!]\sigma_1 = \mathcal{E}'[\![e]\!]\sigma_2 = (v, v_{\mathfrak{of}})$ where $v \notin \{0, \text{error}\}$;
   The execution from $(s_1, m_1(\text{loop}_c^1, \sigma_1))$ proceeds as follows.

   $(s_1, m_1(\text{loop}_c^1, \sigma_1))$
   $= (\text{while}_{\langle n_1 \rangle}(e) \; \{S_1\}, m_1(\text{loop}_c^1, \sigma_1))$
   $\rightarrow (\text{while}_{\langle n_1 \rangle}((v, v_{\mathfrak{of}})) \; \{S_1\}, m_1(\text{loop}_c^1, \sigma_1))$ by the EEval' rule
   $\rightarrow (\text{while}_{\langle n_1 \rangle}(v) \; \{S_1\}, m_1(\text{loop}_c^1, \sigma_1))$
   by rule E-Oflow1 or E-Oflow2
   $\rightarrow (S_1; \text{while}_{\langle n_1 \rangle}(e) \; \{S_1\}, m_1(\text{loop}_c^1[1/(n_1)], \sigma_1))$
   by the Wh-T1 rule.

   Similarly, $(s_2, m_2(\text{loop}_c^2, \sigma_2)) \xrightarrow{2} (S_2; \text{while}_{\langle n_2 \rangle}(e)\{S_2\}, m_2(\text{loop}_c^2[1/(n_2)], \sigma_2))$. After two steps of executions of s_1 and s_2, crash flags are not set, the loop counter value of s_1 and s_2 are 1, value stores σ_1 and σ_2 agree on values of variables in TVar(s).
   We show that TVar$(S_1) \subseteq$ TVar(s). By definition of loop variables, LVar$(s_1) = \bigcup_{j \geq 0} \text{Imp}(S_1^j, \text{LVar}(S_1) \cup \text{Use}(e))$. By notation of S^0, $S^0 = \text{skip}$. By definition of imported variables, Imp$(S_1^0, \text{LVar}(S_1) \cup \text{Use}(e))$ = LVar$(S_1) \cup$ Use(e). Then LVar$(S_1) \subseteq$ LVar(s). By similar argument, we have CVar$(S_1) \subseteq$ CVar(s). Hence, TVar$(S_1) \subseteq$ TVar(s). Similarly, TVar$(S_2) \subseteq$ TVar(s). By assumption, S_1 and S_2 either both terminate or both do not terminate when started in state $m_1(\text{loop}_c^1[1/(n_1)], \sigma_1)$ and $m_2(\text{loop}_c^2[1/(n_2)], \sigma_2)$ in which $\forall y \in \text{TVar}(S_1) \cup \text{TVar}(S_2), \sigma_1(y) = \sigma_2(y)$ and crash flags are not set. Then there are two possibilities:
   (a) S_1 and S_2 both terminate when started in states $m_1(\text{loop}_c^1[1/(n_1)], \sigma_1)$ and $m_2(\text{loop}_c^2[1/(n_2)], \sigma_2)$ respectively:
   $(S_1, m_1(\text{loop}_c^1[1/(n_1)], \sigma_1)) \xrightarrow{*} (\text{skip}, m_{1_1}(\mathfrak{f}_1, \text{loop}_c^{1_1}, \sigma_{1_1}))$
   and
   $(S_2, m_2(\text{loop}_c^2[1/(n_2)], \sigma_2)) \xrightarrow{*} (\text{skip}, m_{2_1}(\mathfrak{f}_2, \text{loop}_c^{2_1}, \sigma_{2_1}))$.
   We show that, after the full execution of S_1 and S_2, the following five properties hold.

- The crash flags are not set.
  By the definition of terminating execution, crash flags are not set, $\mathfrak{f}_1 = \mathfrak{f}_2 = 0$.
- The loop counter of s_1 and s_2 are of value 1, $\text{loop}_c^{1_1}(n_1) = \text{loop}_c^{2_1}(n_2) = 1$.
  By the assumption of unique loop labels, $s_1 \notin S_1$. Then, the loop counter value of n_1 is not redefined in the execution of S_1 by corollary E.2, $\text{loop}_c^1[1/n_1](n_1) = \text{loop}_c^{1_1}(n_1) = 1$. Similarly, the loop counter value of n_2 is not redefined in the execution of S_2, $\text{loop}_c^2[1/(n_2)](n_2) = \text{loop}_c^{2_1}(n_2) = 1$.
- In any state in the execution $(s_1, m_1) \xrightarrow{*} (s_1, m_{1_1}(\text{loop}_c^{1_1}, \sigma_{1_1}))$, the loop counter of s_1 is less than or equal to 1.
  As is shown above, the loop counter of s_1 is zero in any of the two states in the one step execution $(s_1, m_1) \to (\text{while}_{\langle n_1 \rangle}(v)\ \{S_1\}, m_1(\text{loop}_c^1, \sigma_1))$, and the loop counter of s_1 is 1 in any states in the execution $(S_1; \text{while}_{\langle n_1 \rangle}(e)\ \ \{S_1\}, m_1(\text{loop}_c^1[i/(n_1)], \sigma_1)) \xrightarrow{*} (s_1, m_{1_1}(\text{loop}_c^{1_1}, \sigma_{1_1}))$.
- In any state in the executions $(s_2, m_2) \xrightarrow{*} (s_2, m_{2_1}(\text{loop}_c^{2_1}, \sigma_{2_1}))$, the loop counter of s_2 is less than or equal to 1.
  By similar argument above.
- The value stores σ_{1_1} and σ_{2_1} agree on values of the termination deciding variables in s_1 and s_2: $\forall x \in \text{TVar}(s), \sigma_{1_1}(x) = \sigma_{2_1}(x)$.
  We show that the imported variables in S_1 relative to those in $\text{LVar}(s)$ are a subset of $\text{LVar}(s)$ and the imported variables in S_1 relative to those in $\text{CVar}(s)$ are a subset of $\text{CVar}(s)$.

  $\text{LVar}(s_1)$
  $= \bigcup_{j \geq 0} \text{Imp}(S_1^j, \text{LVar}(S_1) \cup \text{Use}(e))$ (1)
  by the definition of loop variables.

  $\text{Imp}(S_1, \text{LVar}(s)) = \text{Imp}(S_1, \text{LVar}(s_1))$
  $= \text{Imp}(S_1, \text{Imp}(s_1, \text{Use}(e) \cup \text{LVar}(S_1)))$
  by the definition of $\text{LVar}(s)$
  $= \text{Imp}(S_1, \bigcup_{j \geq 0} \text{Imp}(S_1^j, \text{LVar}(S_1) \cup \text{Use}(e)))$ by (1)
  $= \bigcup_{j \geq 0} \text{Imp}(S_1, \text{Imp}(S_1^j, \text{LVar}(S_1) \cup \text{Use}(e)))$
  by Lemma C.2
  $= \bigcup_{j > 0} \text{Imp}(S_1^j, \text{LVar}(S_1) \cup \text{Use}(e))$ by Lemma C.1
  $\subseteq \bigcup_{j \geq 0} \text{Imp}(S_1^j, \text{LVar}(S_1) \cup \text{Use}(e))$
  $= \text{Imp}(s_1, \text{LVar}(S_1) \cup \text{Use}(e)) = \text{LVar}(s_1) = \text{LVar}(s)$.

  Similarly, $\text{Imp}(S_1, \text{CVar}(s)) \subseteq \text{CVar}(s)$. Hence, $\text{Imp}(S_1, \text{TVar}(s)) \subseteq \text{TVar}(s)$. In the same way, we can show that $\text{Imp}(S_2, \text{TVar}(s)) \subseteq \text{TVar}(s)$. Consequently, the value stores σ_{1_1} and σ_{2_1} agree on the values of the imported variables in S_1 and S_2 relative to those in $\text{TVar}(s)$, $\forall x \in \text{Imp}(S_1, \text{TVar}(s)) \cup \text{Imp}(S_2, \text{TVar}(s)), \sigma_1(x,) = \sigma_2(x)$. Because S_1 and S_2 have equivalent computation of every variable in $\text{TVar}(s)$ when started in states agreeing on the values of the imported variables relative to $\text{TVar}(s)$, by Theorem 1, value stores σ_{1_1} and σ_{2_1} agree on the values of the variables $\text{TVar}(s)$, $\forall x \in \text{TVar}(s), \sigma_{1_1}(x) = \sigma_{2_1}(x)$.

It follows that, by Corollary E.1,
$(S_1; \text{while}_{\langle n_1 \rangle}(e)\{S_1\}, m_1(\text{loop}_c^1[1/(n_1)], \sigma_1)) \xrightarrow{*}$
$(\text{while}_{\langle n_1 \rangle}(e)\ \{S_1\}, m_{1_1}(\text{loop}_c^{1_1}, \sigma_{1_1})) = (s_1, m_{1_1}(\text{loop}_c^{1_1}, \sigma_{1_1}))$
and
$(S_2; \text{while}_{\langle n_2 \rangle}(e)\ \{S_2\}, m_2(\text{loop}_c^2[1/(n_2)], \sigma_2)) \xrightarrow{*}$
$(\text{while}_{\langle n_2 \rangle}(e)\ \{S_2\}, m_{2_1}(\text{loop}_c^{2_1}, \sigma_{2_1})) = (s_2, m_{2_1}(\text{loop}_c^{2_1}, \sigma_{2_1}))$.

(b) S_1 and S_2 do not terminate when started in states $m_1(\text{loop}_c^1[1/(n_1)], \sigma_1)$ and $m_2(\text{loop}_c^2[1/(n_2)], \sigma_2)$ respectively:
$\forall k > 0, (S_1, m_1(\text{loop}_c^1[1/(n_1)], \sigma_1)) \xrightarrow{k} (S_{1_k}, m_{1_{1_k}}(\text{loop}_c^{1_{1_k}}, \sigma_{1_{1_k}}))$ and
$(S_2, m_2(\text{loop}_c^2[1/(n_2)], \sigma_2)) \xrightarrow{k} (S_{2_k}, m_{2_{1_k}}(\text{loop}_c^{2_{1_k}}, \sigma_{2_{1_k}}))$ in which $S_{1_k} \neq \text{skip}, S_{2_k} \neq \text{skip}$.
By our assumption of unique loop labels, $s_1 \notin S_1$. Then, $\forall k > 0, \text{loop}_c^{1_{1_k}}(n_1) = \text{loop}_c^1[1/(n_1)](n_1) = 1$. Similarly, $\forall k > 0, \text{loop}_c^{2_{1_k}}(n_2)$
$= \text{loop}_c^2[1/(n_2)](n_2) = 1$. In addition, by Lemma E.2,
$\forall k > 0, (S_1; s_1, m_1(\text{loop}_c^1[1/(n_1)], \sigma_1)) \xrightarrow{k} (S_k; s_1, m_{1_k}(\text{loop}_c^{1_k}, \sigma_{1_k}))$ and
$(S_2; s_2, m_2(\text{loop}_c^2[1/(n_2)], \sigma_2)) \xrightarrow{k} (S_{2_k}; s_2, m_{2_k}(\text{loop}_c^{2_k}, \sigma_{2_k}))$
in which $S_{1_k} \neq \text{skip}, S_{2_k} \neq \text{skip}$.
In summary, loop counters of s_1 and s_2 are less than or equal to 1, and s_1 and s_2 do not terminate such that there are no configurations (s_1, m_{1_1}) and (s_2, m_{2_1}) reachable from (s_1, m_1) and (s_2, m_2), respectively, in which crash flags are not set, the loop counters of s_1 and s_2 are equal to 1 and value stores agree on the values of variables in $\text{TVar}(s)$.

Induction Step.
The induction hypothesis IH is that, for a positive integer i, one of the following holds:

1. The loop counters for s_1 and s_2 are less than i, and s_1 and s_2 both terminate in the same way:
   $\forall m_1'\ m_2'$ such that $(s_1, m_1) \xrightarrow{*} (S_1', m_1'(\text{loop}_c^{1'}))$ and $(s_2, m_2) \xrightarrow{*} (S_2', m_2'(\text{loop}_c^{2'}))$,
   $\text{loop}_c^{1'}(n_1) < i$ and $\text{loop}_c^{2'}(n_2) < i$ and one of the following holds:
   (a) s_1 and s_2 both terminate:
   $(s_1, m_1) \xrightarrow{*} (\text{skip}, m_1'')$ and $(s_2, m_2) \xrightarrow{*} (\text{skip}, m_2'')$.
   (b) s_1 and s_2 both do not terminate:
   $\forall k > 0, (s_1, m_1) \xrightarrow{k} (S_{1_k}, m_{1_k})$ and $(s_2, m_2) \xrightarrow{k} (S_{2_k}, m_{2_k})$ in which $S_{1_k} \neq \text{skip}, S_{2_k} \neq \text{skip}$.
2. The loop counters for s_1 and s_2 are less than or equal to i, and s_1 and s_2 do not terminate such that there are no configurations (s_1, m_{1_i}) and (s_2, m_{2_i}) reachable from (s_1, m_1) and (s_2, m_2), respectively, in which crash flags are not set, the loop counters of s_1 and s_2 are equal to i and value stores agree on the values of variables in $\text{TVar}(s)$:
   - $\forall m_1'\ m_2'\ :\ (s_1, m_1) \xrightarrow{*} (S_1, m_1'(\text{loop}_c^{1'})), (s_2, m_2) \xrightarrow{*} (S_2, m_2'(\text{loop}_c^{2'}))$ where $\text{loop}_c^{1'}(n_1) \leq i, \text{loop}_c^{2'}(n_2) \leq i$;
   - $\forall k > 0\ :\ (s_1, m_1) \xrightarrow{k} (S_{1_k}, m_{1_k}), (s_2, m_2) \xrightarrow{k} (S_{2_k}, m_{2_k})$ where $S_{1_k} \neq \text{skip}, S_{2_k} \neq \text{skip}$; and
   - $\nexists (s_1, m_{1_i}), (s_2, m_{2_i})\ :\ (s_1, m_1) \xrightarrow{*} (s_1, m_{1_i}(\mathfrak{f}_1, \text{loop}_c^{1_i}, \sigma_{1_i})) \wedge (s_2, m_2) \xrightarrow{*} (s_2, m_{2_i}(\mathfrak{f}_2, \text{loop}_c^{2_i}, \sigma_{2_i}))$ where
     - $\mathfrak{f}_1 = \mathfrak{f}_2 = 0$; and
     - $\text{loop}_c^{1_i}(n_1) = \text{loop}_c^{2_i}(n_2) = i$; and
     - $\forall x \in \text{TVar}(s)\ :\ \sigma_{1_i}(x) = \sigma_{2_i}(x)$.
3. There are two configurations (s_1, m_{1_i}) and (s_2, m_{2_i}) reachable from (s_1, m_1) and (s_2, m_2), respectively, in which crash flags are not set, the loop counters of s_1 and s_2 are equal to i and value stores agree on the values of variables in $\text{TVar}(s)$ and, for every state in execution, $(s_1, m_1) \xrightarrow{*} (s_1, m_{1_i})$ or $(s_2, m_2) \xrightarrow{*} (s_2, m_{2_i})$ the loop counters for s_1 and s_2 are less than or equal to i respectively:

$\exists(s_1, m_{1_i})\,(s_2, m_{2_i}) \,:\, (s_1, m_1) \xrightarrow{*} (s_1, m_{1_i}(\mathfrak{f}_1, \text{loop}_c^{1_i}, \sigma_{1_i}))\wedge$ $(s_2, m_2) \xrightarrow{*} (s_2, m_{2_i}(\mathfrak{f}_2, \text{loop}_c^{2_i}, \sigma_{2_i}))$ where

- $\mathfrak{f}_1 = \mathfrak{f}_2 = 0$; and
- $\text{loop}_c^{1_i}(n_1) = \text{loop}_c^{2_i}(n_2) = i$; and
- $\forall x \in \text{TVar}(s) \,:\, \sigma_{1_i}(x) = \sigma_{2_i}(x,$; and
- $\forall m_1' \,:\, (s_1, m_1) \xrightarrow{*} (S_1', m_1'(\text{loop}_c^{1'})) \xrightarrow{*} (s_1, m_{1_i})$, $\text{loop}_c^{1'}(n_1) \le i$; and
- $\forall m_2' \,:\, (s_2, m_2) \xrightarrow{*} (S_2', m_2'(\text{loop}_c^{2'})) \xrightarrow{*} (s_2, m_{2_i})$, $\text{loop}_c^{2'}(n_2) \le i$;

Then we show that, for the positive integer $i+1$, one of the following holds:

1. The loop counters for s_1 and s_2 are less than $i+1$, and s_1 and s_2 both terminate in the same way:
   $\forall m_1'\, m_2'$ such that $(s_1, m_1) \xrightarrow{*} (S_1', m_1'(\text{loop}_c^{1'}))$ and $(s_2, m_2) \xrightarrow{*} (S_2', m_2'(\text{loop}_c^{2'}))$,
   $\text{loop}_c^{1'}(n_1) < i+1$ and $\text{loop}_c^{2'}(n_2) < i+1$ and one of the following holds:
   (a) s_1 and s_2 both terminate:
   $(s_1, m_1) \xrightarrow{*} (\text{skip}, m_1'')$ and $(s_2, m_2) \xrightarrow{*} (\text{skip}, m_2'')$.
   (b) s_1 and s_2 both do not terminate:
   $\forall k > 0, (s_1, m_1) \xrightarrow{k} (S_{1_k}, m_{1_k})$ and $(s_2, m_2) \xrightarrow{k} (S_{2_k}, m_{2_k})$ in which $S_{1_k} \neq \text{skip}, S_{2_k} \neq \text{skip}$.
2. The loop counters for s_1 and s_2 are less than or equal to $i+1$, and s_1 and s_2 do not terminate such that there are no configurations $(s_1, m_{1_{i+1}})$ and $(s_2, m_{2_{i+1}})$ reachable from (s_1, m_1) and (s_2, m_2), respectively, in which crash flags are set, the loop counters of s_1 and s_2 are equal to $i+1$ and value stores agree on the values of variables in TVar(s):
   - $\forall m_1'\, m_2' \,:\, (s_1, m_1) \xrightarrow{*} (S_1, m_1'(\text{loop}_c^{1'})), (s_2, m_2) \xrightarrow{*} (S_2, m_2'(\text{loop}_c^{2'}))$ where $\text{loop}_c^{1'}(n_1) \le i+1, \text{loop}_c^{2'}(n_2) \le i+1$;
   - $\forall k > 0 \,:\, (s_1, m_1) \xrightarrow{k} (S_{1_k}, m_{1_k}), (s_2, m_2) \xrightarrow{k} (S_{2_k}, m_{2_k})$ where $S_{1_k} \neq \text{skip}, S_{2_k} \neq \text{skip}$; and
   - $\nexists(s_1, m_{1_{i+1}}), (s_2, m_{2_{i+1}}) \,:$ $(s_1, m_1) \xrightarrow{*} (s_1, m_{1_{i+1}}(\mathfrak{f}_1, \text{loop}_c^{1_{i+1}}, \sigma_{1_{i+1}}))\wedge(s_2, m_2) \xrightarrow{*} (s_2, m_{2_{i+1}}(\mathfrak{f}_2, \text{loop}_c^{2_{i+1}}, \sigma_{2_{i+1}}))$ where
     - $\mathfrak{f}_1 = \mathfrak{f}_2 = 0$; and
     - $\text{loop}_c^{1_{i+1}}(n_1) = \text{loop}_c^{2_{i+1}}(n_2) = i+1$; and
     - $\forall x \in \text{TVar}(s) \,:\, \sigma_{1_{i+1}}(x) = \sigma_{2_{i+1}}(x)$.
3. There are two configurations $(s_1, m_{1_{i+1}})$ and $(s_2, m_{2_{i+1}})$ reachable from (s_1, m_1) and (s_2, m_2), respectively, in which the loop counters of s_1 and s_2 are equal to $i+1$ and value stores agree on the values of variables in TVar(s) and, for every state in execution, $(s_1, m_1) \xrightarrow{*} (s_1, m_{1_{i+1}})$ or $(s_2, m_2) \xrightarrow{*} (s_2, m_{2_{i+1}})$ the loop counters for s_1 and s_2 are less than or equal to $i+1$ respectively:
   $\exists(s_1, m_{1_{i+1}})\,(s_2, m_{2_{i+1}}) \,:$ $(s_1, m_1) \xrightarrow{*} (s_1, m_{1_{i+1}}(\text{loop}_c^{1_{i+1}}, \sigma_{1_{i+1}})) \wedge (s_2, m_2) \xrightarrow{*} (s_2, m_{2_{i+1}}(\text{loop}_c^{2_{i+1}}, \sigma_{2_{i+1}}))$ where
   - $\text{loop}_c^{1_{i+1}}(n_1) = \text{loop}_c^{2_{i+1}}(n_2) = i+1$; and
   - $\forall x \in \text{TVar}(s) \,:\, \sigma_{1_{i+1}}(x) = \sigma_{2_{i+1}}(x)$; and
   - $\forall m_1' \,:\, (s_1, m_1) \xrightarrow{*} (S_1', m_1'(\text{loop}_c^{1'})) \xrightarrow{*} (s_1, m_{1_i})$, $\text{loop}_c^{1'}(n_1) \le i+1$; and
   - $\forall m_2' \,:\, (s_2, m_2) \xrightarrow{*} (S_2', m_2'(\text{loop}_c^{2'})) \xrightarrow{*} (s_2, m_{2_i})$, $\text{loop}_c^{2'}(n_2) \le i+1$;

By the hypothesis IH, one of the following holds:

1. The loop counters for s_1 and s_2 are less than i:
   $\forall m_1'\, m_2'$ such that $(s_1, m_1) \xrightarrow{*} (S_1', m_1'(\text{loop}_c^{1'}))$ and $(s_2, m_2) \xrightarrow{*} (S_2', m_2'(\text{loop}_c^{2'}))$,
   $\text{loop}_c^{1'}(n_1) < i$ and $\text{loop}_c^{2'}(n_2) < i$ and one of the following holds:
   (a) s_1 and s_2 both terminate:
   $(s_1, m_1) \xrightarrow{*} (\text{skip}, m_1'')$ and $(s_2, m_2) \xrightarrow{*} (\text{skip}, m_2'')$.
   (b) s_1 and s_2 both do not terminate:
   $\forall k > 0, (s_1, m_1) \xrightarrow{k} (S_{1_k}, m_{1_k})$ and $(s_2, m_2) \xrightarrow{k} (S_{2_k}, m_{2_k})$ in which $S_{1_k} \neq \text{skip}, S_{2_k} \neq \text{skip}$.

   When this case holds, then we have the loop counters for s_1 and s_2 are less than $i+1$, and s_1 and s_2 both terminate in the same way:
   $\forall m_1'\, m_2'$ such that $(s_1, m_1) \xrightarrow{*} (S_1', m_1'(\text{loop}_c^{1'}))$ and $(s_2, m_2) \xrightarrow{*} (S_2', m_2'(\text{loop}_c^{2'}))$,
   $\text{loop}_c^{1'}(n_1) < i+1$ and $\text{loop}_c^{2'}(n_2) < i+1$ and one of the following holds:
   (a) s_1 and s_2 both terminate:
   $(s_1, m_1) \xrightarrow{*} (\text{skip}, m_1'')$ and $(s_2, m_2) \xrightarrow{*} (\text{skip}, m_2'')$.
   (b) s_1 and s_2 both do not terminate:
   $\forall k > 0, (s_1, m_1) \xrightarrow{k} (S_{1_k}, m_{1_k})$ and $(s_2, m_2) \xrightarrow{k} (S_{2_k}, m_{2_k})$ in which $S_{1_k} \neq \text{skip}, S_{2_k} \neq \text{skip}$.
2. The loop counters for s_1 and s_2 are less than or equal to i, and s_1 and s_2 both do not terminate such that there are no configurations (s_1, m_{1_i}) and (s_2, m_{2_i}) reachable from (s_1, m_1) and (s_2, m_2), respectively, in which the loop counters of s_1 and s_2 are equal to i and value stores agree on the values of variables in TVar(s):
   - $\forall m_1'\, m_2' \,:\, (s_1, m_1) \xrightarrow{*} (S_1, m_1'(\text{loop}_c^{1'})), (s_2, m_2) \xrightarrow{*} (S_2, m_2'(\text{loop}_c^{2'}))$ where $\text{loop}_c^{1'}(n_1) \le i, \text{loop}_c^{2'}(n_2) \le i$;
   - $\forall k > 0 \,:\, (s_1, m_1) \xrightarrow{k} (S_{1_k}, m_{1_k}), (s_2, m_2) \xrightarrow{k} (S_{2_k}, m_{2_k})$ where $S_{1_k} \neq \text{skip}, S_{2_k} \neq \text{skip}$; and
   $\nexists(s_1, m_{1_i}), (s_2, m_{2_i}) \,:\, (s_1, m_1) \xrightarrow{*} (s_1, m_{1_i}(\mathfrak{f}_1, \text{loop}_c^{1_i}, \sigma_{1_i}))\wedge$ $(s_2, m_2) \xrightarrow{*} (s_2, m_{2_i}(\mathfrak{f}_2, \text{loop}_c^{2_i}, \sigma_{2_i}))$ where
     - $\mathfrak{f}_1 = \mathfrak{f}_2 = 0$; and
     - $\text{loop}_c^{1_i}(n_1) = \text{loop}_c^{2_i}(n_2) = i$; and
     - $\forall x \in \text{TVar}(s) \,:\, \sigma_{1_i}(x) = \sigma_{2_i}(x)$.

   When this case holds, we have the loop counter of s_1 and s_2 are less than $i+1$, and s_1 and s_2 both do not terminate:
   $\forall m_1'\, m_2'$ such that $(s_1, m_1) \xrightarrow{*} (S_1', m_1'(\text{loop}_c^{1'}))$ and $(s_2, m_2) \xrightarrow{*} (S_2', m_2'(\text{loop}_c^{2'}))$,
   $\text{loop}_c^{1'}(n_1) < i+1$ and $\text{loop}_c^{2'}(n_2) < i+1$ and s_1 and s_2 both do not terminate:
   $\forall k > 0, (s_1, m_1) \xrightarrow{k} (S_{1_k}, m_{1_k})$ and $(s_2, m_2) \xrightarrow{k} (S_{2_k}, m_{2_k})$ in which $S_{1_k} \neq \text{skip}, S_{2_k} \neq \text{skip}$.
3. There are two configurations (s_1, m_{1_i}) and (s_2, m_{2_i}) reachable from (s_1, m_1) and (s_2, m_2), respectively, in which crash flags are not set, the loop counters of s_1 and s_2 are equal to i and value stores agree on the values of variables in TVar(s) and, for every state in executions $(s_1, m_1) \xrightarrow{*} (s_1, m_{1_i})$ and $(s_2, m_2) \xrightarrow{*} (s_2, m_{2_i})$ the loop counters for s_1 and s_2 are less than or equal to i respectively:
   $\exists(s_1, m_{1_i})\,(s_2, m_{2_i}) \,:\, (s_1, m_1) \xrightarrow{*} (s_1, m_{1_i}(\mathfrak{f}_1, \text{loop}_c^{1_i}, \sigma_{1_i}))\wedge$ $(s_2, m_2) \xrightarrow{*} (s_2, m_{2_i}(\mathfrak{f}_2, \text{loop}_c^{2_i}, \sigma_{2_i}))$ where
   - $\mathfrak{f}_1 = \mathfrak{f}_2 = 0$; and
   - $\text{loop}_c^{1_i}(n_1) = \text{loop}_c^{2_i}(n_2) = i$; and
   - $\forall x \in \text{TVar}(s), \sigma_{1_i}(x) = \sigma_{2_i}(x)$; and

- $\forall m_1' : (s_1, m_1) \xrightarrow{*} (S_1', m_1'(\text{loop}_c^{1'})) \xrightarrow{*} (s_1, m_{1_i})$, $\text{loop}_c^{1'}(n_1) \leq i$; and
- $\forall m_2' : (s_2, m_2) \xrightarrow{*} (S_2', m_2'(\text{loop}_c^{2'})) \xrightarrow{*} (s_2, m_{2_i})$, $\text{loop}_c^{2'}(n_2) \leq i$.

By similar argument in base case, evaluations of the predicate expression of s_1 and s_2 w.r.t value stores σ_{1_i} and σ_{2_i} produce same value. Then there are three possibilities:

(a) $\mathcal{E}'[\![e]\!]\sigma_{1_i} = \mathcal{E}'[\![e]\!]\sigma_{2_i} = (\text{error}, *)$.
Then the execution of s_1 proceeds as follows.
$(s_1, m_{1_i}(\mathfrak{f}_1, \sigma_{1_i}))$
$= (\text{while}_{\langle n_1 \rangle}(e)\ \{S_1\}, m_{1_i}(\mathfrak{f}_1, \sigma_{1_i}))$
$\rightarrow(\text{while}_{\langle n_1 \rangle}((\text{error}, *))\ \{S_1\}, m_{1_i}(\mathfrak{f}_1, \sigma_{1_i}))$ by the EEval' rule
$\rightarrow(\text{while}_{\langle n_1 \rangle}(0)\ \{S_1\}, m_{1_i}(1/\mathfrak{f}_1))$ by the ECrash rule
$\xrightarrow{k}(\text{while}_{\langle n_1 \rangle}(0)\ \{S_1\}, m_{1_i}(1/\mathfrak{f}_1))$, for any $k \geq 0$, by the Crash rule.
Similarly, the execution of s_2 started in the state $m_{2_i}(\sigma_{2_i})$ does not terminate.
The loop counters for s_1 and s_2 are less than $i+1$:
$\forall m_1'\, m_2'$ such that $(s_1, m_1) \xrightarrow{*} (S_1', m_1'(\text{loop}_c^{1'}))$ and $(s_2, m_2) \xrightarrow{*} (S_2', m_2'(\text{loop}_c^{2'}))$,
$\text{loop}_c^{1'}(n_1) < i+1$ and $\text{loop}_c^{2'}(n_2) < i+1$.
Besides, s_1 and s_2 both do not terminate when started in states m_1 and m_2,
$\forall k > 0, (s_1, m_1) \xrightarrow{k} (S_{1_k}, m_{1_k})$ and $(s_2, m_2) \xrightarrow{k} (S_{2_k}, m_{2_k})$ in which $S_{1_k} \neq \text{skip}$, $S_{2_k} \neq \text{skip}$.

(b) $\mathcal{E}'[\![e]\!]\sigma_{1_i} = \mathcal{E}'[\![e]\!]\sigma_{2_i} = (0, v_{\mathfrak{of}})$
The execution from $(s_1, m_{1_i}(\text{loop}_c^{1_i}, \sigma_{1_i}))$ proceeds as follows.
$(s_1, m_{1_i}(\text{loop}_c^{1_i}, \sigma_{1_i}))$
$= (\text{while}_{\langle n_1 \rangle}(e)\ \{S_1\}, m_{1_i}(\text{loop}_c^{1_i}, \sigma_{1_i}))$
$\rightarrow(\text{while}_{\langle n_1 \rangle}((0, v_{\mathfrak{of}}))\ \{S_1\}, m_{1_i}(\text{loop}_c^{1_i}, \sigma_{1_i}))$ by rule EEval'
$\rightarrow(\text{while}_{\langle n_1 \rangle}(0)\ \{S_1\}, m_{1_i}(\text{loop}_c^{1_i}, \sigma_{1_i}))$
by rule E-Oflow1 or E-Oflow2
$\rightarrow(\text{skip}, m_{1_i}(\text{loop}_c^{1_i}[0/(n_1)], \sigma_{1_i}))$ by the Wh-F2 rule.
By the hypothesis IH, the loop counter of s_1 and s_2 in any configuration in executions $(s_1, m_1) \xrightarrow{*} (s_1, m_{1_i}(\text{loop}_c^{1_i}, \sigma_{1_i}))$ and $(s_2, m_2) \xrightarrow{*} (s_2, m_{2_i}(\text{loop}_c^{2_i}, \sigma_{2_i}))$ respectively are less than or equal to i,
$\forall m_1' : (s_1, m_1) \xrightarrow{*} (S_1', m_1'(\text{loop}_c^{1'})) \xrightarrow{*} (s_1, m_{1_i}(\text{loop}_c^{1_i}, \sigma_{1_i}))$, $\text{loop}_c^{1'}(n_1) \leq i$; and
$\forall m_2' : (s_2, m_2) \xrightarrow{*} (S_2', m_2'(\text{loop}_c^{2'})) \xrightarrow{*} (s_2, m_{2_i}(\text{loop}_c^{2_i}, \sigma_{2_i}))$, $\text{loop}_c^{2'}(n_2) \leq i$.
Therefore, s_1 and s_2 both terminate and the loop counter of s_1 and s_2 in any state in executions respectively are less than $i+1$.

(c) $\mathcal{E}'[\![e]\!]\sigma_{1_i} = \mathcal{E}'[\![e]\!]\sigma_{2_i} = (v, v_{\mathfrak{of}})$ where $v \notin \{0, \text{error}\}$;
The execution from $(s_1, m_{1_i}(\text{loop}_c^{1_i}, \sigma_{1_i}))$ proceeds as follows.
$(s_1, m_{1_i}(\text{loop}_c^{1_i}, \sigma_{1_i}))$
$= (\text{while}_{\langle n_1 \rangle}(e)\ \{S_1\}, m_{1_i}(\text{loop}_c^{1_i}, \sigma_{1_i}))$
$\rightarrow(\text{while}_{\langle n_1 \rangle}((v, v_{\mathfrak{of}}))\ \{S_1\}, m_{1_i}(\text{loop}_c^{1_i}, \sigma_{1_i}))$ by rule EEval'
$\rightarrow(\text{while}_{\langle n_1 \rangle}((v, v_{\mathfrak{of}}))\ \{S_1\}, m_{1_i}(\text{loop}_c^{1_i}, \sigma_{1_i}))$ by rule EEval'
$\rightarrow(\text{while}_{\langle n_1 \rangle}(v)\ \{S_1\}, m_{1_i}(\text{loop}_c^{1_i}, \sigma_{1_i}))$
by rule E-Oflow1 or E-Oflow2
$\rightarrow(S_1; \text{while}_{\langle n_1 \rangle}(e)\ \{S_1\}, m_{1_i}(\text{loop}_c^{1_i}[(i+1)/(n_1)], \sigma_{1_i}))$ by rule Wh-T.
Similarly, $(s_2, m_{2_i}(\text{loop}_c^{2_i}, \sigma_{2_i})) \xrightarrow{2} (S_2; \text{while}_{\langle n_2 \rangle}(e)\{S_2\}, m_{2_i}(\text{loop}_c^{2_i}[(i+1)/(n_2)], \sigma_{2_i}))$.
By similar argument in base case, the executions of S_1 and S_2 terminate in the same way when started in states $m_{1_i}(\text{loop}_c^{1_i}[(i{+}1)/(n_1)], \sigma_{1_i})$ and $m_{2_i}(\text{loop}_c^{2_i}[(i{+}1)/(n_2)], \sigma_{2_i})$ respectively. Then there are two possibilities.

i. S_1 and S_2 terminate when started in states $m_{1_i}(\text{loop}_c^{1_i}[(i+1)/(n_1)], \sigma_{1_i})$ and $m_{2_i}(\text{loop}_c^{2_i}[(i+1)/(n_2)], \sigma_{2_i})$ respectively
$(S_1; s_1, m_{1_i}(\text{loop}_c^{1_i}[(i+1)/(n_1)], \sigma_{1_i})) \xrightarrow{*} (s_1, m_{1_{i+1}}(\mathfrak{f}_1, \text{loop}_c^{1_{i+1}}, \sigma_{1_{i+1}}))$ and
$(S_2; s_1, m_{2_i}(\text{loop}_c^{2_i}[(i+1)/(n_2)], \sigma_{2_i})) \xrightarrow{*} (s_2, m_{2_{i+1}}(\mathfrak{f}_2, \text{loop}_c^{2_{i+1}}, \sigma_{2_{i+1}}))$ such that all of the following holds:
- $\mathfrak{f}_1 = \mathfrak{f}_2 = 0$; and
- $\text{loop}_c^{1_{i+1}}(n_1) = \text{loop}_c^{2_{i+1}}(n_2) = i+1$; and
- $\forall y \in \text{TVar}(s), \sigma_{1_{i+1}}(y) = \sigma_{2_{i+1}}(y)$, and
- in any state in the execution $(s_1, m_{1_i}) \xrightarrow{*} (s_1, m_{1_{i+1}}(\text{loop}_c^{1_{i+1}}, \sigma_{1_{i+1}}))$, the loop counter of s_1 is less than or equal to $i+1$.
- in any state in the executions $(s_2, m_{2_i}) \xrightarrow{*} (s_2, m_{2_{i+1}}(\text{loop}_c^{2_{i+1}}, \sigma_{2_{i+1}}))$, the loop counter of s_2 is less than or equal to $i+1$.

With the hypothesis IH, there are two configurations $(s_1, m_{1_{i+1}})$ and $(s_2, m_{2_{i+1}})$ reachable from (s_1, m_1) and (s_2, m_2), respectively, in which crash flags are not set, the loop counters of s_1 and s_2 are equal to $i+1$ and value stores agree on the values of TVar(s) and, for every state in executions $(s_1, m_1) \xrightarrow{*} (s_1, m_{1_{i+1}})$ and $(s_2, m_2) \xrightarrow{*} (s_2, m_{2_{i+1}})$ the loop counters for s_1 and s_2 are less than or equal to $i+1$ respectively:
$\exists (s_1, m_{1_{i+1}})\,(s_2, m_{2_{i+1}})$:
$(s_1, m_1) \xrightarrow{*} (s_1, m_{1_{i+1}}(\mathfrak{f}_1, \text{loop}_c^{1_{i+1}}, \sigma_{1_{i+1}})) \wedge (s_2, m_2) \xrightarrow{*} (s_2, m_{2_{i+1}}(\mathfrak{f}_2, \text{loop}_c^{2_{i+1}}, \sigma_{2_{i+1}}))$ where
- $\mathfrak{f}_1 = \mathfrak{f}_2 = 0$; and
- $\text{loop}_c^{1_{i+1}}(n_1) = \text{loop}_c^{2_{i+1}}(n_2) = i+1$; and
- $\forall x \in \text{TVar}(s), \sigma_{1_{i+1}}(x) = \sigma_{2_{i+1}}(x)$; and
- $\forall m_1' : (s_1, m_1) \xrightarrow{*} (S_1', m_1'(\text{loop}_c^{1'})) \xrightarrow{*} (s_1, m_{1_{i+1}}(\text{loop}_c^{1_{i+1}}, \sigma_{1_{i+1}}))$, $\text{loop}_c^{1'}(n_1) \leq i+1$; and
- $\forall m_2' : (s_2, m_2) \xrightarrow{*} (S_2', m_2'(\text{loop}_c^{2'})) \xrightarrow{*} (s_2, m_{2_{i+1}}(\text{loop}_c^{2_{i+1}}, \sigma_{2_{i+1}}))$, $\text{loop}_c^{2'}(n_2) \leq i+1$.

ii. S_1 and S_2 do not terminate when started in states $m_{1_i}(\text{loop}_c^{1_i}[(i+1)/(n_1)], \sigma_{1_i})$ and $m_{2_i}(\text{loop}_c^{2_i}[(i+1)/(n_2)], \sigma_{2_i})$ respectively:
$\forall k > 0, (S_1, m_{1_i}(\text{loop}_c^{1_i}[(i+1)/(n_1)], \sigma_{1_i})) \xrightarrow{k} (S_{1_k}, m_{1_{1_k}}(\text{loop}_c^{1_{1_k}}, \sigma_{1_{1_k}}))$ and
$(S_2, m_{2_i}(\text{loop}_c^{2_i}[(i+1)/(n_2)], \sigma_{2_i})) \xrightarrow{k} (S_{2_k}, m_{2_{1_k}}(\text{loop}_c^{2_{1_k}}, \sigma_{2_{1_k}}))$ in which $S_{1_k} \neq \text{skip}$, $S_{2_k} \neq \text{skip}$.
By our assumption of unique loop labels, $s_1 \notin S_1$. Then, $\forall k > 0, \text{loop}_c^{1_{1_k}}(n_1) = \text{loop}_c^{1_i}[(i+1)/(n_1)](n_1) = i+1$. Similarly, $\forall k > 0, \text{loop}_c^{2_{1_k}}(n_2) = \text{loop}_c^{2_i}[(i+1)/(n_2)](n_2) = i+1$. In addition, by Lemma E.2,
$\forall k > 0, (S_1; s_1, m_{1_i}(\text{loop}_c^{1_i}[(i+1)/(n_1)], \sigma_{1_i})) \xrightarrow{k} (S_{1_k}; s_1, m_{1_{1_k}}(\text{loop}_c^{1_{1_k}}, \sigma_{1_{1_k}}))$ and $(S_2; s_2, m_2(\text{loop}_c^2[(i+1)/(n_2)], \sigma_2)) \xrightarrow{k} (S_{2_k}; s_2, m_{2_{1_k}}(\text{loop}_c^{2_{1_k}}, \sigma_{2_{1_k}}))$ in which $S_{1_k} \neq \text{skip}$, $S_{2_k} \neq \text{skip}$.
In summary, the loop counter of s_1 and s_2 are less than equal to $i+1$, and s_1 and s_2 do not terminate such that

there are no configurations $(s_1, m_{1_{i+1}})$ and $(s_2, m_{2_{i+1}})$ reachable from (s_1, m_1) and (s_2, m_2), respectively, in which crash flags are not set, the loop counters of s_1 and s_2 are equal to $i+1$ and value stores agree on values of variables in TVar(s).

□

Corollary 5.3. *Let s_1 = "while$_{\langle n_1\rangle}(e)\{S_1\}$" and s_2 = "while$_{\langle n_2\rangle}(e)\{S_2\}$" be two while statements respectively, with the same set of the termination deciding variables, TVar(s_1) = TVar(s_2) = TVar(s), whose bodies S_1 and S_2 satisfy the proof rule of equivalently computation of variables in TVar(s), $\forall x \in$ TVar(s) : $(S_1) \equiv_x^S (S_2)$, and whose bodies S_1 and S_2 terminate in the same way when started in states with crash flags not set and agreeing on values of variables in TVar$(S_1) \cup$ TVar(S_2):*
$\forall m_{S_1}(\mathfrak{f}_{S_1}, \sigma_{S_1}), m_{S_2}(\mathfrak{f}_{S_2}, \sigma_{S_2})$:
$(((\forall z \in \mathit{TVar}(S_1) \cup \mathit{TVar}(S_2)), \sigma_{S_1}(z) = \sigma_{S_2}(z)) \wedge (\mathfrak{f}_{S_1} = \mathfrak{f}_{S_2} = 0)) \Rightarrow (S_1, m_{S_1}(\mathfrak{f}_{S_1}, \sigma_{S_1})) \equiv_H (S_2, m_{S_2}(\mathfrak{f}_{S_2}, \sigma_{S_2}))$.

If s_1 and s_2 start in the state $m_1(\mathfrak{f}_1, loop_c^1, \sigma_1)$ and $m_2(\mathfrak{f}_2, loop_c^2, \sigma_2)$ respectively in which crash flags are not set, $\mathfrak{f}_1 = \mathfrak{f}_2 = 0$, s_1 and s_2 have not already executed, $loop_c^1(n_1) = loop_c^2(n_2) = 0$, value stores σ_1 and σ_2 agree on values of variables in TVar(s), $\forall x \in$ TVar$(s), \sigma_1(x) = \sigma_2(x)$, then s_1 and s_2 terminate in the same way:

1. *s_1 and s_2 both terminate, $(s_1, m_1) \xrightarrow{*} (skip, m_1')$, $(s_2, m_2) \xrightarrow{*} (skip, m_2')$.*
2. *s_1 and s_2 both do not terminate, $\forall k > 0$, $(s_1, m_1) \xrightarrow{k} (S_{1_k}, m_{1_k})$, $(s_2, m_2) \xrightarrow{k} (S_{2_k}, m_{2_k})$ where $S_{1_k} \neq$ skip, $S_{2_k} \neq$ skip.*

This is from Lemma 5.15 immediately.

Lemma5.16 is necessary only for showing the same I/O sequence in the next section.

Lemma 5.16. *Let s_1 = "while$_{\langle n_1\rangle}(e)\{S_1\}$" and s_2 = "while$_{\langle n_2\rangle}(e)\{S_2\}$" be two while statements in program P_1 and P_2 respectively with the same set of termination deciding variables, TVar(s_1) = TVar(s_2) = TVar(s), whose bodies S_1 and S_2 satisfy the proof rule of equivalently computation of variables in TVar(s), $\forall x \in$ TVar(s) : $S_1 \equiv_x^S S_2$ and whose bodies S_1 and S_2 terminate in the same way in executions when started in states with crash flags not set and agreeing on values of variables in TVar$(S_1) \cup$ TVar(S_2):*
$\forall m_{S_1}(\mathfrak{f}_{S_1}, \sigma_{S_1})\, m_{S_2}(\mathfrak{f}_{S_2}, \sigma_{S_2})$:
$(((\forall z \in \mathit{TVar}(S_1) \cup \mathit{TVar}(S_2)), \sigma_{S_1}(z) = \sigma_{S_2}(z)) \wedge (\mathfrak{f}_{S_1} = \mathfrak{f}_{S_2} = 0)$
$\Rightarrow (S_1, m_{S_1}(\mathfrak{f}_{S_1}, \sigma_{S_1})) \equiv_H (S_2, m_{S_2}(\mathfrak{f}_{S_2}, \sigma_{S_2}))$.

If s_1 and s_2 start in the state $m_1(\mathfrak{f}_1, loop_c^1, \sigma_1)$ and $m_2(\mathfrak{f}_2, loop_c^2, \sigma_2)$ respectively in which crash flags are not set, $\mathfrak{f}_1 = \mathfrak{f}_2 = 0$, s_1 and s_2 have not already executed, $loop_c^1(n_1) = loop_c^2(n_2) = 0$, value stores σ_1 and σ_2 agree on values of variables in TVar(s), $\forall x \in$ TVar$(s), \sigma_1(x) = \sigma_2(x)$, one of the following holds:

1. *s_1 and s_2 both terminate and the loop counters of s_1 and s_2 are less than a positive integer i and less than or equal to $i-1$: $(s_1, m_1) \xrightarrow{*} (skip, m_1')$, $(s_2, m_2) \xrightarrow{*} (skip, m_2')$ where both of the following hold:*
   - *The loop counters of s_1 and s_2 are less than a positive integer i:*
     $\exists i > 0\, \forall m_1'\, m_2'$:
     $(s_1, m_1) \xrightarrow{*} (S_1', m_1'(loop_c^{1'}))$, $loop_c^{1'}(n_1) < i$ *and*
     $(s_2, m_2) \xrightarrow{*} (S_2', m_2'(loop_c^{2'}))$, $loop_c^{2'}(n_2) < i$.
   - *$\forall 0 < j < i$, there are two configurations (s_1, m_{1_j}) and (s_2, m_{2_j}) reachable from (s_1, m_1) and (s_2, m_2), respectively, in which both crash flags are not set, the loop counters of s_1 and s_2 are equal to j and value stores agree on the values of variables in TVar(s), and for every state in execution $(s_1, m_1) \xrightarrow{*} (s_1, m_{1_j})$ or $(s_2, m_2) \xrightarrow{*} (s_2, m_{2_j})$, the loop counters for s_1 and s_2 are less than or equal to j respectively:*
     $\exists (s_1, m_{1_j})\, (s_2, m_{2_j})$:
     $(s_1, m_1) \xrightarrow{*} (s_1, m_{1_j}(\mathfrak{f}_1, loop_c^{1^j}, \sigma_{1_j})) \wedge$
     $(s_2, m_2) \xrightarrow{*} (s_2, m_{2_j}(\mathfrak{f}_2, loop_c^{2^j}, \sigma_{2_j}))$ *where*
     - $\mathfrak{f}_1 = \mathfrak{f}_2 = 0$*; and*
     - $loop_c^{1^j}(n_1) = loop_c^{2^j}(n_2) = j$*; and*
     - $\forall x \in$ *TVar*(s) : $\sigma_{1_j}(x) = \sigma_{2_j}(x)$*; and*
     - $\forall m_1'$: $(s_1, m_1) \xrightarrow{*} (S_1', m_1'(loop_c^{1'})) \xrightarrow{*} (s_1, m_{1_j})$, $loop_c^{1'}(n_1) \leq j$*; and*
     - $\forall m_2'$: $(s_2, m_2) \xrightarrow{*} (S_2', m_2'(loop_c^{2'})) \xrightarrow{*} (s_2, m_{2_j})$, $loop_c^{2'}(n_2) \leq j$.
2. *s_1 and s_2 both do not terminate, $\forall k > 0$, $(s_1, m_1) \xrightarrow{k} (S_{1_k}, m_{1_k})$, $(s_2, m_2) \xrightarrow{k} (S_{2_k}, m_{2_k})$ where $S_{1_k} \neq$ skip, $S_{2_k} \neq$ skip such that one of the following holds:*
   (a) *For any positive integer i, there are two configurations (s_1, m_{1_i}) and (s_2, m_{2_i}) reachable from (s_1, m_1) and (s_2, m_2), respectively, in which both crash flags are not set, the loop counters of s_1 and s_2 are equal to i and value stores agree on the values of variables in TVar(s), and for every state in execution $(s_1, m_1) \xrightarrow{*} (s_1, m_{1_i})$ or $(s_2, m_2) \xrightarrow{*} (s_2, m_{2_i})$, the loop counters for s_1 and s_2 are less than or equal to i respectively:*
   $\forall i > 0\, \exists (s_1, m_{1_i})\, (s_2, m_{2_i})$:
   $(s_1, m_1) \xrightarrow{*} (s_1, m_{1_i}(\mathfrak{f}_1, loop_c^{1^i}, \sigma_{1_i})) \wedge$
   $(s_2, m_2) \xrightarrow{*} (s_2, m_{2_i}(\mathfrak{f}_2, loop_c^{2^i}, \sigma_{2_i}))$ *where*
   - $\mathfrak{f}_1 = \mathfrak{f}_2 = 0$*; and*
   - $loop_c^{1^i}(n_1) = loop_c^{2^i}(n_2) = i$*; and*
   - $\forall x \in$ *TVar*(s) : $\sigma_{1_i}(x) = \sigma_{2_i}(x)$*; and*
   - $\forall m_1'$: $(s_1, m_1) \xrightarrow{*} (S_1', m_1'(loop_c^{1'})) \xrightarrow{*} (s_1, m_{1_i})$, $loop_c^{1'}(n_1) \leq i$*; and*
   - $\forall m_2'$: $(s_2, m_2) \xrightarrow{*} (S_2', m_2'(loop_c^{2'})) \xrightarrow{*} (s_2, m_{2_i})$, $loop_c^{2'}(n_2) \leq i$;

   (b) *The loop counters for s_1 and s_2 are less than a positive integer i and less than or equal to $i-1$ such that all of the following hold:*
   - $\exists i > 0, \forall m_1'\, m_2'$: $(s_1, m_1) \xrightarrow{*} (S_1, m_1'(loop_c^{1'}))$, $(s_2, m_2) \xrightarrow{*} (S_2, m_2'(loop_c^{2'}))$ *where* $loop_c^{1'}(n_1) < i$, $loop_c^{2'}(n_2) < i$;
   - *$\forall 0 < j < i$, there are two configurations (s_1, m_{1_j}) and (s_2, m_{2_j}) reachable from (s_1, m_1) and (s_2, m_2), respectively, in which both crash flags are not set, the loop counters of s_1 and s_2 are equal to j and value stores agree on the values of variables in TVar(s), and for every state in execution $(s_1, m_1) \xrightarrow{*} (s_1, m_{1_j})$ or $(s_2, m_2) \xrightarrow{*} (s_2, m_{2_j})$, the loop counters for s_1 and s_2 are less than or equal to j respectively:*
     $\exists (s_1, m_{1_j})\, (s_2, m_{2_j})$:
     $(s_1, m_1) \xrightarrow{*} (s_1, m_{1_j}(\mathfrak{f}_1, loop_c^{1^j}, \sigma_{1_j})) \wedge$
     $(s_2, m_2) \xrightarrow{*} (s_2, m_{2_j}(\mathfrak{f}_2, loop_c^{2^j}, \sigma_{2_j}))$ *where*
     - $\mathfrak{f}_1 = \mathfrak{f}_2 = 0$*; and*
     - $loop_c^{1^j}(n_1) = loop_c^{2^j}(n_2) = j$*; and*
     - $\forall x \in$ *TVar*(s) : $\sigma_{1_j}(x) = \sigma_{2_j}(x)$*; and*

- $\forall m_1' : (s_1, m_1) \xrightarrow{*} (S_1', m_1'(\textit{loop}_c^{1'})) \xrightarrow{*} (s_1, m_{1_j})$, $\textit{loop}_c^{1'}(n_1) \leq j$; *and*
- $\forall m_2' : (s_2, m_2) \xrightarrow{*} (S_2', m_2'(\textit{loop}_c^{2'})) \xrightarrow{*} (s_2, m_{2_j})$, $\textit{loop}_c^{2'}(n_2) \leq j$;

(c) *The loop counters for s_1 and s_2 are less than or equal to some positive integer i such that all of the following hold:*

- $\exists i > 0 \, \forall m_1' \, m_2' : (s_1, m_1) \xrightarrow{*} (S_1, m_1'(\textit{loop}_c^{1'}))$, $(s_2, m_2) \xrightarrow{*} (S_2, m_2'(\textit{loop}_c^{2'}))$ *where* $\textit{loop}_c^{1'}(n_1) \leq i$, $\textit{loop}_c^{2'}(n_2) \leq i$;
- $\forall 0 < j < i$, *there are two configurations (s_1, m_{1_j}) and (s_2, m_{2_j}) reachable from (s_1, m_1) and (s_2, m_2), respectively, in which both crash flags are not set, the loop counters of s_1 and s_2 are equal to j and value stores agree on the values of variables in TVar(s), and for every state in execution $(s_1, m_1) \xrightarrow{*} (s_1, m_{1_j})$ or $(s_2, m_2) \xrightarrow{*} (s_2, m_{2_j})$, the loop counters for s_1 and s_2 are less than or equal to j respectively:*
  $\exists (s_1, m_{1_j}) \, (s_2, m_{2_j})$:
  $(s_1, m_1) \xrightarrow{*} (s_1, m_{1_j}(\mathfrak{f}_1, \textit{loop}_c^{1_j}, \sigma_{1_j})) \wedge$
  $(s_2, m_2) \xrightarrow{*} (s_2, m_{2_j}(\mathfrak{f}_2, \textit{loop}_c^{2_j}, \sigma_{2_j}))$ *where*
  - $\mathfrak{f}_1 = \mathfrak{f}_2 = 0$; *and*
  - $\textit{loop}_c^{1_j}(n_1) = \textit{loop}_c^{2_j}(n_2) = j$; *and*
  - $\forall x \in \textit{TVar}(s) : \sigma_{1_j}(x) = \sigma_{2_j}(x)$; *and*
  - $\forall m_1' : (s_1, m_1) \xrightarrow{*} (S_1', m_1'(\textit{loop}_c^{1'})) \xrightarrow{*} (s_1, m_{1_j})$, $\textit{loop}_c^{1'}(n_1) \leq j$; *and*
  - $\forall m_2' : (s_2, m_2) \xrightarrow{*} (S_2', m_2'(\textit{loop}_c^{2'})) \xrightarrow{*} (s_2, m_{2_j})$, $\textit{loop}_c^{2'}(n_2) \leq j$;
- *There are no configurations (s_1, m_{1_i}) and (s_2, m_{2_i}) reachable from (s_1, m_1) and (s_2, m_2), respectively, in which crash flags are not set, the loop counters of s_1 and s_2 are equal to i, and value stores agree on the values of variables in TVar(s):*
  $\nexists (s_1, m_{1_i}) \, (s_2, m_{2_i})$:
  $(s_1, m_1) \xrightarrow{*} (s_1, m_{1_i}(\mathfrak{f}_1, \textit{loop}_c^{1_i}, \sigma_{1_i})) \wedge$
  $(s_2, m_2) \xrightarrow{*} (s_2, m_{2_i}(\mathfrak{f}_2, \textit{loop}_c^{2_i}, \sigma_{2_i}))$ *where*
  - $\mathfrak{f}_1 = \mathfrak{f}_2 = 0$; *and*
  - $\textit{loop}_c^{1_i}(n_1) = \textit{loop}_c^{2_i}(n_2) = i$; *and*
  - $\forall x \in \textit{TVar}(s) : \sigma_{1_i}(x) = \sigma_{2_i}(x)$.

Proof. From lemma 5.15, we have s_1 and s_2 terminate in the same way when started in states m_1 and m_2 respectively. Then there are two big cases.

1. s_1 and s_2 both terminate.
   Let i be the smallest integer such that the loop counters of s_1 and s_2 are less than i in the executions. Then there are two possibilities.
   (a) $i = 1$.
   In the proof of Lemma 5.15, the evaluation of the loop predicate of s_1 and s_2 produce zero w.r.t value stores, σ_1 and σ_2. Then the execution of s_1 proceeds as follows:
   $(s_1, m_1(\text{loop}_c^1, \sigma_1))$
   $= (\text{while}_{\langle n_1 \rangle}(e) \; \{S_1\}, m_1(\text{loop}_c^1))$
   $\rightarrow (\text{while}_{\langle n_1 \rangle}((0, v_{\mathfrak{of}})) \; \{S_1\}, m_1(\text{loop}_c^1))$ by rule EEval'
   $\rightarrow (\text{while}_{\langle n_1 \rangle}(0) \; \{S_1\}, m_1(\text{loop}_c^1))$
   by rule E-Oflow1 or E-Oflow2
   $\rightarrow (\text{skip}, m_1(\text{loop}_c^1 \setminus \{(n_1, *)\}))$
   by rule Wh-F1 or Wh-F2.
   Similarly, $(s_2, m_2(\text{loop}_c^2, \sigma_2)) \xrightarrow{2} (\text{skip}, m_2(\text{loop}_c^2 \setminus \{(n_2, *)\}))$. Then the lemma holds because of the initial configuration $(s_1, m_1(\text{loop}_c^1, \sigma_1))$ and $(s_2, m_2(\text{loop}_c^2, \sigma_2))$.
   (b) $i > 1$.
   Because s_1 and s_2 terminate, i is the smallest positive integer such that the loop counters of s_1 and s_2 are less than i, by Lemma 5.15, $\forall 0 < j < i$, there are two configurations (s_1, m_{1_j}) and (s_2, m_{2_j}) reachable from (s_1, m_1) and (s_2, m_2), respectively, in which both crash flags are not set, the loop counters of s_1 and s_2 are equal to j and value stores agree on the values of variables in TVar(s), and for every state in execution, $(s_1, m_1) \xrightarrow{*} (s_1, m_{1_j})$ or $(s_2, m_2) \xrightarrow{*} (s_2, m_{2_j})$ the loop counters for s_1 and s_2 are less than or equal to j respectively. With the initial configuration (s_1, m_1) and (s_2, m_2), the lemma holds.
2. s_1 and s_2 both do not terminate. There are three possibilities.
   (a) $\forall i > 0$, there are two configurations (s_1, m_{1_i}) and (s_2, m_{2_i}) reachable from (s_1, m_1) and (s_2, m_2), respectively, in which both crash flags are not set, the loop counters of s_1 and s_2 are equal to i and value stores agree on the values of variables in TVar(s), and for every state in execution, $(s_1, m_1) \xrightarrow{*} (s_1, m_{1_i})$ or $(s_2, m_2) \xrightarrow{*} (s_2, m_{2_i})$ the loop counters for s_1 and s_2 are less than or equal to i respectively.
   (b) The loop counters of s_1 and s_2 are less than a positive integer i.
   Let i be the smallest positive integer such that there is no positive integer $j < i$ that the loop counters of s_1 and s_2 are less than the positive integer j. This case occurs when s_1 and s_2 finish the full $(i-1)$th iterations and both executions raise an exception in the evaluation of the loop predicate of s_1 and s_2 for the ith time. There are further two possibilities.
      i. $i = 1$.
      In proof of Lemma 5.15, evaluations of the predicate expression of s_1 and s_2 raise an exception w.r.t value stores σ_1 and σ_2. The lemma holds.
      ii. $i > 1$. By the assumption of initial states (s_1, m_1) and (s_2, m_2), when $j = 0$, initial states (s_1, m_1) and (s_2, m_2) have crash flags not set, the loop counters of s_1 and s_2 are zero and value stores agree on values of variables of s_1 and s_2.
      By Lemma 5.15, $\forall 0 < j < i$, there are two configurations (s_1, m_{1_j}) and (s_2, m_{2_j}) reachable from (s_1, m_1) and (s_2, m_2), respectively, in which both crash flags are not set, the loop counters of s_1 and s_2 are equal to j and value stores agree on the values of variables in TVar(s), and for every state in execution, $(s_1, m_1) \xrightarrow{*} (s_1, m_{1_j})$ or $(s_2, m_2) \xrightarrow{*} (s_2, m_{2_j})$ the loop counters for s_1 and s_2 are less than or equal to j respectively. With the initial configuration (s_1, m_1) and (s_2, m_2), the lemma holds.
   (c) The loop counters of s_1 and s_2 are less than or equal to a positive integer i.
   Let i be the smallest positive integer such that the loop counters of s_1 and s_2 are less than or equal to the positive integer i. There are two possibilities.
      i. $i = 1$.
      In the proof of Lemma 5.15, the execution of s_1 proceeds as follows:
      $(s_1, m_1(\text{loop}_c^1, \sigma_1))$
      $= (\text{while}_{\langle n_1 \rangle}(e) \; \{S_1\}, m_1(\text{loop}_c^1, \sigma_1))$
      $\rightarrow (\text{while}_{\langle n_1 \rangle}((v, v_{\mathfrak{of}})) \; \{S_1\}, m_1(\text{loop}_c^1, \sigma_1))$ where $v \neq 0$
      by rule EEval'
      $\rightarrow (\text{while}_{\langle n_1 \rangle}(v) \; \{S_1\}, m_1(\text{loop}_c^1, \sigma_1))$
      by rule E-Oflow1 or E-Oflow2

$\rightarrow(S_1; \text{while}_{\langle n_1\rangle}(e)\ \{S_1\}, m_1(\text{loop}_c^1[1/(n_1)], \sigma_1))$
by rule Wh-T .

The execution of s_2 proceeds to $(S_2; \text{while}_{\langle n_2\rangle}(e)\quad \{S_2\}, m_2(\text{loop}_c^2[1/(n_2)], \sigma_2))$. In addition, executions of S_1 and S_2 do not terminate when started in states $m_1(\text{loop}_c^1[1/(n_1)], \sigma_1)$ and $m_2(\text{loop}_c^2[1/(n_2)], \sigma_2)$. By Lemma E.2, executions of s_1 and s_2 do not terminate.

ii. $i > 1$.
In the proof of Lemma 5.15, when started in the state $(s_1, m_{1_{i-1}}(\mathfrak{f}_1, \text{loop}_c^{1_{i-1}}, \sigma_{1_{i-1}}))$ the execution of s_1 proceeds as follows:

$(s_1, m_{1_{i-1}}(\mathfrak{f}_1, \text{loop}_c^{1_{i-1}}, \sigma_{1_{i-1}}))$
$= (\text{while}_{\langle n_1\rangle}(e)\ \{S_1\}, m_{1_{i-1}}(\mathfrak{f}_1, \text{loop}_c^{1_{i-1}}, \sigma_{1_{i-1}}))$
$\rightarrow(\text{while}_{\langle n_1\rangle}((v, v_{\mathfrak{of}}))\ \{S_1\}, m_{1_{i-1}}(\mathfrak{f}_1, \text{loop}_c^{1_{i-1}}, \sigma_{1_{i-1}}))$
by rule EEval'
$\rightarrow(\text{while}_{\langle n_1\rangle}(v)\ \{S_1\}, m_{1_{i-1}}(\mathfrak{f}_1, \text{loop}_c^{1_{i-1}}, \sigma_{1_{i-1}}))$
by rule E-Oflow1 or E-Oflow2
$\rightarrow(S_1; \text{while}_{\langle n_1\rangle}(e)\ \{S_1\},$
$m_{1_{i-1}}(\mathfrak{f}_1, \text{loop}_c^{1_{i-1}}[i/(n_1)], \sigma_{1_{i-1}}))$
by rule Wh-T1 or Wh-T2.

The execution of s_2 proceeds to $(S_2; \text{while}_{\langle n_2\rangle}(e)\ \{S_2\}, m_{2_{i-1}}(\mathfrak{f}_2, \text{loop}_c^{2_{i-1}}[i/(n_2)], \sigma_{2_{i-1}}))$. In addition, executions of S_1 and S_2 do not terminate when started in states $m_{1_{i-1}}(\mathfrak{f}_1, \text{loop}_c^{1_{i-1}}[i/(n_1)], \sigma_{1_{i-1}})$ and $m_{2_{i-1}}(\mathfrak{f}_2, \text{loop}_c^{2_{i-1}}[i/(n_2)], \sigma_{2_{i-1}})$. By Lemma E.2, executions of s_1 and s_2 do not terminate.

□

5.4 Behavioral equivalence

We now propose a proof rule under which two programs produce the same *output sequence*, namely the same I/O sequence till any ith output value. We care about the I/O sequence due to the possible crash from the lack of input. We start by giving the definition of the same output sequence, then we describe the proof rule under which two programs produce the same output sequence, finally we show that our proof rule ensures same output together with the necessary auxiliary lemmas. We use the notation "Out(σ)" to represent the output sequence in value store σ, the I/O sequence $\sigma(id_{IO})$ till the rightmost output value. Particularly, when there is no output value in the I/O sequence $\sigma(id_{IO})$, Out$(\sigma) = \varnothing$.

Definition 20. (Same output sequence) *Two statement sequences S_1 and S_2 produce the same output sequence when started in states m_1 and m_2 respectively, written $(S_1, m_1) \equiv_O (S_2, m_2)$, iff $\forall m_1'\, m_2'$ such that $(S_1, m_1) \xrightarrow{*} (S_1', m_1'(\sigma_1'))$ and $(S_2, m_2) \xrightarrow{*} (S_2', m_2'(\sigma_2'))$, there are states $m_1''\, m_2''$ reachable from initial states m_1 and m_2, $(S_1, m_1) \xrightarrow{*} (S_1'', m_1''(\sigma_1''))$ and $(S_2, m_2) \xrightarrow{*} (S_2'', m_2''(\sigma_2''))$ so that Out$(\sigma_2'') =$ Out(σ_1') and Out$(\sigma_1'') =$ Out(σ_2').*

5.4.1 Proof rule for behavioral equivalence

We show the proof rules of the behavioral equivalence. The output sequence produced in executions of a statement sequence S depends on values of a set of variables in the program, the output deciding variables OVar(S). The output deciding variables are of two parts: TVar$_o(S)$ are variables affecting the termination of executions of a statement sequence; Imp$_o(S)$ are variables affecting values of the I/O sequence produced in executions of a statement sequence. The definitions of TVar$_o(S)$ and Imp$_o(S)$ are shown in Definition 21 and 22.

Definition 21. (Imported variables relative to output) *The imported variables in one program S relative to output, written Imp$_o(S)$, are listed as follows:*

1. *Imp$_o(S) = \{id_{IO}\}$, if $(\forall e$: "output e" $\notin S)$;*
2. *Imp$_o$("output e") $= \{id_{IO}\} \cup$ Use(e);*
3. *Imp$_o$("If (e) then $\{S_t\}$ else $\{S_f\}$") $=$ Use$(e) \cup$ Imp$_o(S_t) \cup$ Imp$_o(S_f)$ if $(\exists e$: "output e" $\in S)$;*
4. *Imp$_o$("while$_{\langle n\rangle}(e)\{S''\}$") $=$ Imp("while$_{\langle n\rangle}(e)\{S''\}$", $\{id_{IO}\}$) if $(\exists e$: "output e" $\in S'')$;*
5. *For $k > 0$, Imp$_o(s_1; ...; s_k; s_{k+1}) =$ Imp$(s_1; ...; s_k,$ Imp$_o(s_{k+1}))$ if $(\exists e$: "output e" $\in s_{k+1})$;*
6. *For $k > 0$, Imp$_o(s_1; ...; s_k; s_{k+1}) =$ Imp$_o(s_1; ...; s_k)$ if $(\forall e$: "output e" $\notin s_{k+1})$;*

Definition 22. (Termination deciding variables relative to output) *The termination deciding variables in a statement sequence S relative to output, written TVar$_o(S)$, are listed as follows:*

1. *TVar$_o(S) = \emptyset$ if $(\forall e$: "output e" $\notin S)$;*
2. *TVar$_o$("output e") $=$ Err(e);*
3. *TVar$_o$("If (e) then $\{S_t\}$ else $\{S_f\}$") $=$ Use$(e) \cup$ TVar$_o(S_t) \cup$ TVar$_o(S_f)$ if $(\exists e$: "output e" $\in S)$;*
4. *TVar$_o$("while$_{\langle n\rangle}(e)\{S''\}$") $=$ TVar("while$_{\langle n\rangle}(e)\{S''\}$") if $(\exists e$: "output e" $\in S'')$;*
5. *For $k > 0$, TVar$_o(s_1; ...; s_k; s_{k+1}) =$ TVar$(s_1; ...; s_k)$ $\cup$Imp$(s_1; ...; s_k,$ TVar$_o(s_{k+1}))$ if $(\exists e$: "output e" $\in s_{k+1})$;*
6. *For $k > 0$, TVar$_o(s_1; ...; s_k; s_{k+1}) =$ TVar$_o(s_1; ...; s_k)$ if $(\forall e$: "output e" $\notin s_{k+1})$;*

Definition 23. (Output deciding variables) *The output deciding variables in a statement sequence S are Imp$_o(S) \cup$ TVar$_o(S)$, written OVar(S).*

The condition of the behavioral equivalence is defined recursively. The base case is for two same output statements or two statements where the output sequence variable is not defined. The inductive cases are syntax directed considering the syntax of compound statements and statement sequences.

Definition 24. (proof rule of behavioral equivalence) *Two statement sequences S_1 and S_2 satisfy the proof rule of behavioral equivalence, written $S_1 \equiv_O^S S_2$, iff one of the following holds:*

1. *S_1 and S_2 are one statement and one of the following holds:*
   (a) *S_1 and S_2 are simple statement and one of the following holds:*
      i. *S_1 and S_2 are not output statement, $\forall e_1\, e_2$: ("output e_1" $\neq S_1) \wedge$ ("output e_2" $\neq S_2)$; or*
      ii. *$S_1 = S_2 =$ "output e".*
   (b) *$S_1 =$ "If (e) then $\{S_1^t\}$ else $\{S_1^f\}$", $S_2 =$ "If (e) then $\{S_2^t\}$ else $\{S_2^f\}$" and all of the following hold:*
      - *There is an output statement in S_1 and S_2, $\exists e_1\, e_2$: ("output e_1" $\in S_1) \wedge$ ("output e_2" $\in S_2)$;*
      - *$(S_1^t \equiv_O^S S_2^t) \wedge (S_1^f \equiv_O^S S_2^f)$;*
   (c) *$S_1 =$ "while$_{\langle n_1\rangle}(e)\ \{S_1''\}$" and $S_2 =$ "while$_{\langle n_2\rangle}(e)\ \{S_2''\}$" and all of the following hold:*
      - *There is an output statement in S_1 and S_2, $\exists e_1\, e_2$: ("output e_1" $\in S_1) \wedge$ ("output e_2" $\in S_2)$;*
      - *$S_1'' \equiv_O^S S_2''$;*
      - *S_1'' and S_2'' have equivalent computation of OVar$(S_1) \cup$ OVar(S_2);*
      - *S_1'' and S_2'' satisfy the proof rule of termination in the same way, $S_1'' \equiv_H^S S_2''$;*
   (d) *Output statements are not in both S_1 and S_2, $\forall e_1\, e_2$: ("output e_1" $\notin S_1) \wedge$ ("output e_2" $\notin S_2)$.*
2. *S_1 and S_2 are not both one statement and one of the following holds:*

(a) $S_1 = S_1'; s_1$ and $S_2 = S_2'; s_2$, and all of the following hold:

- *$S_1' \equiv_O^S S_2'$;*
- *S_1' and S_2' have equivalent computation of OVar$(s_1) \cup$ OVar(s_2);*
- *S_1' and S_2' satisfy the proof rule of termination in the same way: $S_1' \equiv_H^S S_2'$;*
- *There is an output statement in both s_1 and s_2, $\exists e_1\, e_2\ :\ (\text{"output } e_1\text{"} \in s_1) \wedge (\text{"output } e_2\text{"} \in s_2)$;*
- *$s_1 \equiv_O^S s_2$;*

(b) There is no output statement in the last statement in S_1 or S_2:
$\big((S_1 = S_1'; s_1) \wedge (S_1' \equiv_O^S S_2) \wedge (\forall e\ :\ \text{"output } e\text{"} \notin s_1)\big)$
$\vee \big((S_2 = S_2'; s_2) \wedge (S_1 \equiv_O^S S_2') \wedge (\forall e\ :\ \text{"output } e\text{"} \notin s_2)\big)$;

5.4.2 Soundness of the proof rule for behavioral equivalence

We show that two statement sequences satisfy the proof rule of the behavioral equivalence and their initial states agree on values of their output deciding variables, then the two statement sequences produce the same output sequence when started in their initial states.

Theorem 5. *Two statement sequences S_1 and S_2 satisfy the proof rule of the behavioral equivalence, $S_1 \equiv_O^S S_2$. If S_1 and S_2 start in states $m_1(\mathfrak{f}_1, \sigma_1)$ and $m_2(\mathfrak{f}_2, \sigma_2)$ where both of the following hold:*

- *Crash flags are not set, $\mathfrak{f}_1 = \mathfrak{f}_2 = 0$;*
- *Value stores σ_1 and σ_2 agree on values of the output deciding variables of S_1 and S_2, $\forall id \in$ OVar$(S_1) \cup$ OVar(S_2) : $\sigma_1(id) = \sigma_2(id)$;*

then S_1 and S_2 produce the same output sequence,
$(S_1, m_1) \equiv_O (S_2, m_2)$.

The proof is by induction on the sum of program size of S_1 and S_2, size(S_1) + size(S_2) and is a case analysis based on $S_1 \equiv_O^S S_2$.

Proof. The proof is by induction on the sum of program size of S_1 and S_2, size(S_1) + size(S_2) and is a case analysis based on $S_1 \equiv_O^S S_2$.
Base case.

S_1 and S_2 are simple statement. There are two cases according to the proof rule of behavioral equivalence because stacks are not changed in executions of S_1 and S_2.

1. S_1 and S_2 are not output statement, $\forall e_1\, e_2\ :\ (\text{"output } e_1\text{"} \neq S_1) \wedge (\text{"output } e_2\text{"} \neq S_2)$;
   By the definition of imported variables relative to output, Imp$_o(S_1)$ = Imp$_o(S_2) = \{id_{IO}\}$. By assumption, initial value stores σ_1 and σ_2 agree on the value of the I/O sequence variable, $\sigma_1(id_{IO}) = \sigma_2(id_{IO})$. By definition, Out$(\sigma_1)$ = Out(σ_2).
   By Lemma 5.23, in any state m_1' reachable from m_1, the output sequence in m_1' is same as that in m_1, $\forall m_1'\ :\ ((S_1, m_1(\sigma_1)) \xrightarrow{*} (S_1', m_1'(\sigma_1'))) \Rightarrow (\text{Out}(\sigma_1') = \text{Out}(\sigma_1))$. Similarly, for any state m_2' reachable from m_2, the output sequence in m_2' is same as that in m_2. The theorem holds.
2. $S_1 = S_2 =$ "output e".
   We show that the expression e evaluates to the same value w.r.t value stores, σ_1, σ_2. By the definition of imported variables relative to output, Imp$_o(S_1)$ = Imp$_o(S_2)$ = Use$(e) \cup \{id_{IO}\}$. Then $\forall x \in$ Use$(e) \cup \{id_{IO}\}$: $\sigma_1(x) = \sigma_2(x)$ by assumption. By Lemma D.1, $\mathcal{E}[\![e]\!]\sigma_1 = \mathcal{E}[\![e]\!]\sigma_2$. Then, there are two possibilities.
   (a) $\mathcal{E}[\![e]\!]\sigma_1 = \mathcal{E}[\![e]\!]\sigma_2 = (\text{error}, v_{\mathfrak{of}})$.
   The execution of S_1 proceeds as follows.
   $(\text{output } e, m_1(\sigma_1))$
   $= (\text{output } (\text{error}, v_{\mathfrak{of}}), m_1(\sigma_1))$ by the rule EEval'
   $\to (\text{output } 0, m_1(1/\mathfrak{f}))$ by the ECrash rule
   $\xrightarrow{i} (\text{output } 0, m_1(1/\mathfrak{f}))$ for any $i > 0$ by the Crash rule.
   Similarly, the execution of S_2 does not terminate and there is no change to I/O sequence in execution. Because $\sigma_1(id_{IO}) = \sigma_2(id_{IO})$, then the output sequence in value stores σ_1 and σ_2 are same, Out(σ_1) = Out(σ_2), the theorem holds.
   (b) $\mathcal{E}[\![e]\!]\sigma_1 = \mathcal{E}[\![e]\!]\sigma_2 \neq (\text{error}, v_{\mathfrak{of}})$
   S_1 and S_2 satisfy the proof rule of equivalent computation of I/O sequence variable and their initial states agree on the values of the imported variables relative to I/O sequence variable. By Theorem 2, S_1 and S_2 produce the same output sequence after terminating execution when started in state $m_1(\sigma_1)$ and $m_2(\sigma_2)$ respectively. The theorem holds.

Induction step.

The hypothesis IH is that Theorem 5 holds when size(S_1) + size$(S_2) = k \geq 2$.

We show Theorem 5 holds when size(S_1) + size$(S_2) = k + 1$. The proof is a case analysis according to the cases in the definition of the proof rule of behavioral equivalence.

1. S_1 and S_2 are one statement and one of the following holds:
   (a) S_1 = "If(e) then $\{S_1^t\}$ else $\{S_1^f\}$" and S_2 = "If(e) then $\{S_2^t\}$ else $\{S_2^f\}$" and all of the following hold:
   - There is an output statement in S_1 and S_2: $\exists e_1\, e_2\ :\ (\text{"output } e_1\text{"} \in S_1) \wedge (\text{"output } e_2\text{"} \in S_2)$;
   - $S_1^t \equiv_O^S S_2^t$;
   - $S_1^f \equiv_O^S S_2^f$;

   By Lemma 5.17, $\{id_{IO}\} \in$ Imp$_o(S_1)$. By assumption, value stores σ_1 and σ_2 agree on the value of the I/O sequence variable and the I/O sequence variable, $\sigma_1(id_{IO}) = \sigma_2(id_{IO})$.
   We show that the evaluations of the predicate expression of S_1 and S_2 w.r.t. initial value store σ_1 and σ_2 produce the same value. We need to show that value stores σ_1 and σ_2 agree on values of variables used in the predicate expression e of S_1 and S_2. Because the output sequence is defined in S_1, by the definition of imported variables relative to output, Imp$_o(S_1)$ = Use$(e) \cup$ Imp$_o(S_1^t) \cup$ Imp$_o(S_1^f)$. Thus, Use$(e) \subseteq$ OVar(S_1). By assumption, value stores σ_1 and σ_2 agree on values of variables used in the predicate expression e of S_1 and S_2, $\forall x \in$ Use(e) : $\sigma_1(x) = \sigma_2(x)$. By Lemma D.1, the evaluations of the predicate expression of S_1 and S_2 w.r.t. pairs value stores, σ_1 and σ_2 generate the same value, $\mathcal{E}'[\![e]\!]\sigma_1 = \mathcal{E}'[\![e]\!]\sigma_2$. Then there are two possibilities.
   i. $\mathcal{E}'[\![e]\!]\sigma_1 = \mathcal{E}'[\![e]\!]\sigma_2 = (\text{error}, v_{\mathfrak{of}})$.
   Then the execution of S_1 proceeds as follows:
   $(\text{If}(e) \text{ then } \{S_1^t\} \text{ else } \{S_1^f\}, m_1(\sigma_1))$
   $\to (\text{If}((\text{error}, v_{\mathfrak{of}})) \text{ then } \{S_1^t\} \text{ else } \{S_1^f\}, m_1(\sigma_1))$
   by the EEval' rule
   $\to (\text{If}(0) \text{ then } \{S_1^t\} \text{ else } \{S_1^f\}, m_1(1/\mathfrak{f}))$
   by the ECrash rule
   $\xrightarrow{i} (\text{If}(0) \text{ then } \{S_1^t\} \text{ else } \{S_1^f\}, m_1(1/\mathfrak{f}))$
   for any $i > 0$, by the Crash rule.

   Similarly, the execution of S_2 does not terminate and does not redefine I/O sequence. Because $\sigma_1(id_{IO}) = \sigma_2(id_{IO})$, the theorem holds.
   ii. $\mathcal{E}'[\![e]\!]\sigma_1 = \mathcal{E}'[\![e]\!]\sigma_2 \neq (\text{error}, v_{\mathfrak{of}})$.
   W.l.o.g., $\mathcal{E}'[\![e]\!]\sigma_1 = \mathcal{E}'[\![e]\!]\sigma_2 = (0, v_{\mathfrak{of}})$. The execution of S_1 proceeds as follows.

(If(e) then $\{S_1^t\}$ else $\{S_1^f\}, m_1(\sigma_1)$)
$\rightarrow$(If($(0, v_{\mathfrak{of}})$) then $\{S_1^t\}$ else $\{S_1^f\}, m_1(\sigma_1)$)
by the EEval rule
$\rightarrow$(If(0) then $\{S_1^t\}$ else $\{S_1^f\}, m_1(\sigma_1)$)
by the E-Oflow1 or E-Oflow2 rule
$\rightarrow(S_1^f, m_1(\sigma_1))$ by the If-F rule.

Similarly, the execution of S_2 proceeds to $(S_2^f, m_2(\sigma_2))$ after two steps. By the hypothesis IH, we show that S_1^f and S_2^f produce the same output sequence when started in states $m_1(\sigma_1)$ and $m_2(\sigma_2)$. We need to show that all required conditions are satisfied.

- size(S_1^f) + size$(S_2^f) \leq k$.
  By definition, size$(S_1) = 1 +$ size(S_1^t) + size(S_1^f). Therefore, size(S_1^f) + size$(S_2^f) < k$.
- Value stores σ_1 and σ_2 agree on values of the out-deciding variables of S_1^f and S_2^f, $\forall x \in$ OVar$(S_1^f) \cup$ OVar(S_2^f) : $\sigma_1(x) = \sigma_2(x)$.
  By the definition of imported variables relative to output, Imp$_o(S_1^f) \subseteq$ Imp$_o(S_1)$. Besides, by the definition of TVar$_o(S_1)$, TVar$_o(S_1^f) \subseteq$ TVar$_o(S_1)$. Then OVar$(S_1^f) \subseteq$ OVar(S_1). Similarly, OVar$(S_2^f) \subseteq$ OVar(S_2).By assumption, the value stores σ_1 and σ_2 agree on the values of the out-deciding variables of S_1^f and S_2^f.

By the hypothesis IH, S_1^f and S_2^f produce the same output sequence when started from state $m_1(\sigma_1)$ and $m_2(\sigma_2)$ respectively. The theorem holds.

(b) S_1 = "while$_{\langle n_1 \rangle}(e)$ $\{S_1''\}$" and S_2 = "while$_{\langle n_2 \rangle}(e)$ $\{S_2''\}$" and all of the following hold:

- There is an output statement in S_1 and S_2: $\exists e_1\, e_2$: ("output e_1" $\in S_1$) $\wedge$ ("output e_2" $\in S_2$);
- $S_1'' \equiv_O^S S_2''$;
- Both loop bodies satisfy the proof rule of termination in the same way: $S_1'' \equiv_H^S S_2''$;
- S_1'' and S_2'' have equivalent computation of OVar$(S_1) \cup$ OVar(S_2);

By Corollary 5.5, we show that S_1 and S_2 produce the same output sequence when started in states $m_1(\sigma_1)$ and $m_2(\sigma_2)$ respectively. We need to show that the required conditions are satisfied.

- Crash flags are not set, $\mathfrak{f}_1 = \mathfrak{f}_2 = 0$.
- Value stores σ_1 and σ_2 agree on the values of the out-deciding variables of S_1 and S_2, $\forall x \in$ OVar$(S_1) \cup$ OVar(S_2) : $\sigma_1(x) = \sigma_2(x)$.
- The loop counter value of S_1 and S_2 are zero in initial loop counter, loop$_c^1(n_1)$ = loop$_c^2(n_2) = 0$.
- The loop body of S_1 and S_2 satisfy the proof rule of termination in the same way, $S_1'' \equiv_H^S S_2''$.
- The loop body of S_1 and S_2 satisfy the proof rule of equivalent computation of OVar$(S_1) \cup$ OVar(S_2).
  The above five conditions are from assumption.
- S_1 and S_2 have same set of termination deciding variables, TVar(S_1) = TVar(S_2).
  By the definition of TVar$_o(S_1)$, TVar$_o(S_1)$ = TVar(S_1) and TVar$_o(S_2)$ = TVar(S_2). By Lemma 5.21, TVar$_o(S_1)$ = TVar$_o(S_2)$. Thus, TVar(S_1) = TVar(S_2).
- S_1 and S_2 have same set of imported variables relative to the I/O sequence variable,
  Imp$(S_1, \{id_{IO}\})$ = Imp$(S_2, \{id_{IO}\})$.
  By Lemma 5.19, Imp$_o(S_1)$ = Imp$_o(S_2)$. By definition, Imp$_o(S_1)$ = Imp$(S_1, \{id_{IO}\})$ and Imp$_o(S_2)$ = Imp$(S_2, \{id_{IO}\})$. Thus, Imp$(S_1, \{id_{IO}\})$ = Imp$(S_2, \{id_{IO}\})$.
- The loop body of S_1 and S_2 produce the same output sequence when started in states with crash flags not set and whose value stores agree on values of the out-deciding variables of S_1'' and S_2'', $\forall m_{S_1''}(\mathfrak{f}_1'', \sigma_1'')\, m_{S_2''}(\mathfrak{f}_2'', \sigma_2'')$: $((\forall x \in \text{OVar}(S_1'') \cup \text{OVar}(S_2'') : \sigma_1''(x) = \sigma_2''(x)) \wedge (\mathfrak{f}_1'' = \mathfrak{f}_2'' = 0)) \Rightarrow (S_1'', m_{S_1''}(\mathfrak{f}_1'', \sigma_1'')) \equiv_O (S_2'', m_{S_2''}(\mathfrak{f}_2'', \sigma_2''))$.
  Because size$(S_1) = 1 +$ size(S_1''), size$(S_2) = 1 +$ size(S_2''), then size(S_1'') + size$(S_2'') < k$. By the hypothesis IH, the condition is satisfied.

By Corollary 5.5, S_1 and S_2 produce the same output sequence when started in states $m_1(\sigma_1)$ and $m_2(\sigma_2)$ respectively. The theorem holds.

(c) Output statements are not in both S_1 and S_2, $\forall e_1\, e_2$: ("output e_1" $\notin S_1$) $\wedge$ ("output e_2" $\notin S_2$).
By Lemma 5.17, $\{id_{IO}\} \subseteq$ Imp$_o(S_1)$. By assumption, value stores in initial states m_1, m_2 agree on values of the I/O sequence variable, $\sigma_1(id_{IO}) = \sigma_2(id_{IO})$. By Lemma 5.23, the value of output sequence is same in m_1 and any state reachable from m_1, $\forall m_1'\, m_2'$: $(S_1, m_1(\sigma_1)) \xrightarrow{*} (S_1', m_1'(\sigma_1'))$ and $(S_2, m_2(\sigma_2)) \xrightarrow{*} (S_2', m_2'(\sigma_2'))$, Out$(\sigma_1')$ = Out(σ_1) = Out(σ_2) = Out(σ_2'). The theorem holds.

2. $S_1 = S_1'; s_1$ and $S_2 = S_2'; s_2$ are not both one statement and one of the following holds:

(a) There is an output statement in both s_1 and s_2, $\exists e_1\, e_2$: ("output e_1" $\in s_1$) $\wedge$ ("output e_2" $\in s_2$), and all of the following hold:

- $S_1' \equiv_O^S S_2'$;
- S_1' and S_2' satisfy the proof rule of termination in the same way: $S_1' \equiv_H^S S_2'$;
- S_1' and S_2' have equivalent computation of OVar$(s_1) \cup$ OVar(s_2);
- $s_1 \equiv_O^S s_2$;

By the hypothesis IH, we show S_1' and S_2' produce the same output sequence when started in states $m_1(\sigma_1)$ and $m_2(\sigma_2)$ respectively. We need to show that all required conditions are satisfied.

- size(S_1') + size$(S_2') < k$.
  By the definition of program size, size$(s_1) \geq 1$, size$(s_2) \geq 1$. Then size(S_1') + size$(S_2') < k$.
- Value stores σ_1 and σ_2 agree on values of the out-deciding variables of S_1' and S_2', $\forall x \in$ OVar$(S_1') \cup$ OVar(S_2') : $\sigma_1(x) = \sigma_2(x)$.
  We show that TVar$_o(S_1') \subseteq$ TVar$_o(S_1)$.
  TVar$_o(S_1')$
  $\subseteq$ TVar(S_1') by Lemma 5.20
  $\subseteq$ TVar$_o(S_1)$ by the definition of TVar$_o(S_1)$

  We show that Imp$_o(S_1') \subseteq$ Imp$_o(S_1)$.
  Imp$_o(S_1')$
  $\subseteq$ Imp$(S_1', \{id_{IO}\})$ (1) by Lemma 5.18

  $\{id_{IO}\} \subseteq$ Imp$_o(s_{k+1})$ (2) by Lemma 5.17

  Combining (1) + (2)
  Imp$(S_1', \{id_{IO}\})$
  $\subseteq$ Imp$(S_1'$, Imp$_o(s_1))$ by Lemma C.2
  = Imp$_o(S_1)$ by the definition of Imp$_o(S)$.

Similarly, OVar(S_2') $\subseteq$ OVar(S_2). By assumption, value stores σ_1 and σ_2 agree on values of out-deciding variables of S_1' and S_2'.

By the hypothesis IH, S_1' and S_2' produce the same output sequence when started in state $m_1(\sigma_1)$ and $m_2(\sigma_2)$ respectively.

We show that S_1 and S_2 produce the same output sequence if s_1 and s_2 execute. We need to show that S_1' and S_2' terminate in the same way when started in states $m_1(\sigma_1)$ and $m_2(\sigma_2)$ respectively. Specifically, we prove that the value stores σ_1 and σ_2 agree on the values of termination deciding variables of S_1' and S_2'. By definition, the termination deciding variables in S_1' are a subset of the termination deciding variables relative to output, TVar(S_1') $\subseteq$ TVar$_o$(S_1). Similarly, TVar(S_2') $\subseteq$ TVar$_o$(S_2). By assumption, the value stores σ_1 and σ_2 agree on the values of the termination deciding variables of S_1' and S_2', $\forall x \in$ TVar(S_1')$\cup$TVar(S_2') $:$ $\sigma_1(x) = \sigma_2(x)$. By Theorem 4, S_1' and S_2' terminate in the same way when started in state $m_1(\sigma_1)$ and $m_2(\sigma_2)$ respectively.

If S_1' and S_2' terminate when started in states $m_1(\sigma_1)$ and $m_2(\sigma_2)$, by Lemma 5.14, S_1' and S_2' consume same amount of input values. In addition, we show that value stores agree on values of the out-deciding variables of s_1 and s_2 by Theorem 2. We need to show that S_1' and S_2' start execution in states agreeing on values of the imported variables in S_1' and S_2' relative to the out-deciding variables of s_1 and s_2.

- Imp(TVar$_o$(s_1), S_1') $\subseteq$ TVar$_o$(S_1).
  This is by the definition of TVar$_o$(S_1).
- Imp(Imp$_o$(s_1), S_1') $=$ Imp$_o$(S_1).
  This is by the definition of Imp$_o$(S_1).

Thus, the imported variables in S_1' relative to the out-deciding variables of s_1 are a subset of the out-deciding variables of S_1, Imp(S_1', OVar(s_1)) $\subseteq$ OVar(S_1). Similarly, Imp(S_2', OVar(s_2)) $\subseteq$ OVar(S_2). By Corollary 5.4, s_1 and s_2 have same out-deciding variables, OVar(s_1) $=$ OVar(s_2). By assumption, S_1' and S_2' terminate when started in states $m_1(\sigma_1)$ and $m_2(\sigma_2)$, $(S_1', m_1(\sigma_1)) \xrightarrow{*}$ (skip, $m_1'(\sigma_1')$), $(S_2', m_1(\sigma_2)) \xrightarrow{*}$ (skip, $m_2'(\sigma_2')$). By Theorem 2, value stores σ_1' and σ_2' agree on values of the out-deciding variables of s_1 and s_2.

By the hypothesis IH again, s_1 and s_2 produce the same output sequence when started in states $m_1'(\sigma_1')$ and $m_2'(\sigma_2')$ respectively. The theorem holds.

(b) There is no output statement in the last statement in S_1 or S_2: W.l.o.g., ($\forall e$: "output e" $\notin s_1$) $\wedge$ ((S_1') $\equiv_O^S$ (S_2)).
By the hypothesis IH, we show that S_1' and S_2 produce the same output sequence when started in states $m_1(\sigma_1)$ and $m_2(\sigma_2)$ respectively. We need to show that two required conditions are satisfied.
- size(S_1') + size(S_2) $\leq k$.
  size(s_1) ≥ 1 by definition. Then size(S_1') + size(S_2) $\leq$ k.
- $\forall x \in$ OVar(S_1') $\cup$ OVar(S_2) $:$ $\sigma_1(x) = \sigma_2(x)$.
  By definition of TVar$_o$(S)/Imp$_o$(S), TVar$_o$(S_1') $=$ TVar$_o$(S_1), and Imp$_o$(S_1') $=$ Imp$_o$(S_1) Hence, $\forall x \in$ OVar(S_1') $\cup$ OVar(S_2) $:$ $\sigma_1(x) = \sigma_2(x)$.

Therefore, S_1' and S_2 produce the same output sequence when started in state $m_1(\sigma_1)$ and $m_2(\sigma_2)$ respectively, $(S_1', m_1) \equiv_O (S_2, m_2)$ by the hypothesis IH.

When the execution of S_1' terminates, then the output sequence is not changed in the execution of s_1 by Lemma 5.23. The theorem holds.

□

5.4.3 Supporting lemmas for the soundness proof of behavioral equivalence

We listed the lemmas and corollaries used in the proof of Theorem 5 below. The supporting lemmas are of two parts. One part is various properties related to the out-deciding variables. The other part is the proof that two loop statements produce the same output sequence.

Lemma 5.17. *For any statement sequence S, the I/O sequence variable is in imported variable in S relative to output, $id_{IO} \in$ Imp$_o$(S).*

Proof. By structure induction on abstract syntax of S. □

Lemma 5.18. *For any statement sequence S, the imported variables in S relative to output are a subset of the imported variables in S relative to the I/O sequence variable, Imp$_o$(S) $\subseteq$ Imp(S, $\{id_{IO}\}$).*

Proof. By induction on abstract syntax of S. In every case, there are subcases based on if there is output statement in the statement sequence S or not if necessary. □

Lemma 5.19. *If two statement sequences S_1 and S_2 satisfy the proof rule of behavioral equivalence, then S_1 and S_2 have the same set of imported variables relative to output, ($S_1 \equiv_O^S S_2$) $\Rightarrow$ (Imp$_o$(S_1) $=$ Imp$_o$(S_2)).*

Proof. By induction on size(S_1) + size(S_2). □

Lemma 5.20. *For any statement sequence S and any variable x, the termination deciding variables in S relative to output is a subset of the termination deciding variables in S, TVar$_o$(S) $\subseteq$ TVar(S).*

Proof. By induction on abstract syntax of S. In every case, there are subcases based on if there is output statement in the statement sequence S or not if necessary. □

Lemma 5.21. *If two statement sequences S_1 and S_2 satisfy the proof rule of behavioral equivalence, then S_1 and S_2 have the same set of termination deciding variables relative to output, ($S_1 \equiv_O^S$ S_2) $\Rightarrow$ (TVar$_o$(S_1) $=$ TVar$_o$(S_2)).*

Proof. By induction on size(S_1) + size(S_2). □

Corollary 5.4. *If two statement sequences S_1 and S_2 satisfy the proof rule of behavioral equivalence, then S_1 and S_2 have the same set of out-deciding variables, ($S_1 \equiv_O^S S_2$) $\Rightarrow$ OVar(S_1) $=$ OVar(S_2).*

Proof. By Lemma 5.19, Imp$_o$(S_1) $=$ Imp$_o$(S_2). By Lemma 5.21, TVar$_o$(S_1) $=$ TVar$_o$(S_2). □

Lemma 5.22. *In one step execution $(S, m(\sigma)) \rightarrow (S', m'(\sigma'))$, if there is no output statement in S, then the output sequence is same in value store σ and σ', Out(σ') $=$ Out(σ).*

Proof. By induction on abstract syntax of S and crash flag $\mathfrak{f}$ in state m. □

Lemma 5.23. *If there is no output statement in S, then, after the execution $(S, m(\sigma)) \xrightarrow{*} (S', m'(\sigma'))$, the output sequence is same in value store σ and σ', Out(σ') $=$ Out(σ).*

Proof. By induction on number k of execution steps in the execution $(S, m(\sigma)) \xrightarrow{k} (S', m'(\sigma'))$. The proof also relies on the fact that if $s \notin S$, then $s \notin S'$. □

Lemma 5.24. *One while statement $s =$ "while$_{\langle n\rangle}(e)\{S\}$" starts in a state $m(\mathfrak{f}, loop_c)$ in which the loop counter of s is zero, $loop_c(n) = 0$ and the crash flag is not set, $\mathfrak{f} = 0$. For any positive integer i, if there is a state $m'(m'_c)$ reachable from m in which the loop counter is i, $loop'_c(n) = i$, then there is a configuration $(S; s, m''(\mathfrak{f}'', loop''_c))$ reachable from the configuration (s, m) in which loop counter of s is i, $loop''_c(n) = i$ and the crash flag is not set, $\mathfrak{f}'' = 0$:*

$\forall i > 0 : (((s, m(\mathfrak{f}, m_c)) \xrightarrow{*} (S', m'(\mathfrak{f}', loop'_c))) \wedge (loop_c(n) = 0) \wedge (\mathfrak{f} = 0) \wedge (loop'_c(n) = i)) \Rightarrow$

$(s, m(\mathfrak{f}, loop_c)) \xrightarrow{*} (S; s, m''(\mathfrak{f}, loop''_c))$ *where* $\mathfrak{f} = 0$ *and* $loop''_c(n) = i$.

Proof. The proof is by induction on i.

Base case $i = 1$.

We show that the evaluation of the loop predicate of s w.r.t the value store σ in the state $m(\text{loop}_c, \sigma)$ produces an nonzero integer value. By our semantic rule, if the evaluation of the predicate expression of s raises an exception, the execution of s proceeds as follows:

$(s, m(\mathfrak{f}, \text{loop}_c, \sigma))$
$= (\text{while}_{\langle n\rangle}(e)\ \{S\}, m(\text{loop}_c, \sigma))$
$\rightarrow(\text{while}_{\langle n\rangle}(\text{error})\ \{S\}, m(\text{loop}_c, \sigma))$ by the EEval rule
$\rightarrow(\text{while}_{\langle n\rangle}(0)\ \{S\}, m(1/\mathfrak{f}, \text{loop}_c, \sigma))$ by the ECrash rule
$\xrightarrow{k}(\text{while}_{\langle n\rangle}(0)\ \{S\}, m(1/\mathfrak{f}, \text{loop}_c))$ for any $k > 0$, by the Crash rule.

Hence, we have a contradiction that there is no configuration in which the loop counter of s is 1.

When the evaluation of the loop predicate expression of s produce zero, the execution of s proceeds as follows:

$(s, m(\mathfrak{f}, \text{loop}_c, \sigma))$
$= (\text{while}_{\langle n\rangle}(e)\ \{S\}, m(\text{loop}_c, \sigma))$
$\rightarrow(\text{while}_{\langle n\rangle}(0)\ \{S\}, m(\text{loop}_c, \sigma))$ by the EEval rule
$\rightarrow(\text{skip}, m(\text{loop}_c[0/n_1]))$ by the Wh-F rule.

Hence, we have a contradiction that there is no configuration in which the loop counter of s is 1. The evaluation of the predicate expression of s w.r.t value store σ produce nonzero value. The execution of s proceeds as follows:

$(s, m(\mathfrak{f}, \text{loop}_c, \sigma))$
$= (\text{while}_{\langle n\rangle}(e)\ \{S\}, m(\mathfrak{f}, \text{loop}_c, \sigma))$
$\rightarrow(\text{while}_{\langle n\rangle}(\mathcal{E}[\![e]\!]\sigma)\ \{S\}, m(\mathfrak{f}, \text{loop}_c, \sigma))$ by the EEval rule
$\rightarrow(S; \text{while}_{\langle n\rangle}(e)\ \{S\}, m(\mathfrak{f}, \text{loop}_c[1/n_1], \sigma))$ by the Wh-T rule.

The lemma holds.

Induction step.

The hypothesis IH is that, if there is a configuration $(S', m_i(\text{loop}^i_c))$ reachable from (s, m) in which the loop counter of s is i, $\text{loop}^i_c(n) = i > 0$, then there is a reachable configuration $(S; s, m_i(\mathfrak{f}, \text{loop}^i_c))$ from (s, m) where the loop counter of s is i and the crash flag is not set.

Then we show that, if there is a configuration $(S', m_{i+1}(\text{loop}^{i+1}_c))$ reachable from (s, m) in which the loop counter of s is $i + 1$, then there is a reachable configuration $(S; s, m_{i+1}(\mathfrak{f}, \text{loop}^{i+1}_c))$ from (s, m) where the loop counter of s is $i + 1$ and the crash flag $\mathfrak{f}$ is not set.

By Lemma E.8, the loop counter of s is increasing by one in one step. Hence, there must be one configuration reachable from (s, m) in which the loop counter of s is i. By hypothesis, there is the configuration $(S; s, m_i(\mathfrak{f}, \text{loop}^i_c, \sigma_i))$ reachable from (s, m) in which the loop counter is i, $\text{loop}^i_c(n) = i$, and the crash flag is not set, $\mathfrak{f} = 0$. By the assumption of unique loop labels, $s \notin S$. Then the loop counter of s is not redefined in the execution of S started in state $m_i(\mathfrak{f}, \text{loop}^i_c, \sigma_i)$. Because there is a configuration in which the loop counter of s is $i + 1$, then the execution of S when started in the state $m_i(\mathfrak{f}, \text{loop}^i_c, \sigma_i)$ terminates, $(S, m_i(\mathfrak{f}, \text{loop}^i_c, \sigma_i)) \xrightarrow{*} (\text{skip}, m_{i+1}(\mathfrak{f}, \text{loop}^{i+1}_c, \sigma_{i+1}))$ where $\mathfrak{f} = 0$ and $\text{loop}^{i+1}_c(n) = i$. By Corollary E.1, $(S; s, m_i(\mathfrak{f}, \text{loop}^i_c, \sigma_i)) \xrightarrow{*} (s, m_{i+1}(\mathfrak{f}, \text{loop}^{i+1}_c, \sigma_{i+1}))$. By similar argument in base case, the evaluation of the predicate expression w.r.t the value store σ_{i+1} produce nonzero integer value. The execution of s proceeds as follows:

$(s, m_{i+1}(\mathfrak{f}, \text{loop}^{i+1}_c, \sigma_{i+1}))$
$= (\text{while}_{\langle n\rangle}(e)\ \{S\}, m_{i+1}(\mathfrak{f}, \text{loop}^{i+1}_c, \sigma_{i+1}))$
$\rightarrow(\text{while}_{\langle n\rangle}(\mathcal{E}[\![e]\!]\sigma_{i+1})\ \{S\}, m_{i+1}(\mathfrak{f}, \text{loop}^{i+1}_c, \sigma_{i+1}))$
by the EEval rule
$\rightarrow(S; \text{while}_{\langle n\rangle}(e)\ \{S\}, m_{i+1}(\mathfrak{f}, \text{loop}^{i+1}_c[(i+1)/n], \sigma_{i+1}))$
by the Wh-T rule.

The lemma holds. □

Lemma 5.25. *Let $s_1 =$ "while$_{\langle n_1\rangle}(e)\ \{S_1\}$" and $s_2 =$ "while$_{\langle n_2\rangle}(e)\ \{S_2\}$" be two while statements and all of the followings hold:*

- *There are output statements in s_1 and s_2, $\exists e_1\, e_2$: ("output e_1" $\in s_1) \wedge$ ("output e_2" $\in s_2)$;*
- *s_1 and s_2 have the same set of termination deciding variables relative to output, and the same set of imported variables relative to output, $(TVar_o(s_1) = TVar_o(s_2) = TVar(s)) \wedge (Imp_o(s_1) = Imp_o(s_2) = Imp(io))$;*
- *Loop bodies S_1 and S_2 satisfy the proof rule of equivalent computation of the out-deciding variables of s_1 and s_2, $\forall x \in OVar(s) = TVar(s) \cup Imp(io) : S_1 \equiv^S_x S_2$;*
- *Loop bodies S_1 and S_2 satisfy the proof rule of termination in the same way, $S_1 \equiv^S_H S_2$;*
- *Loop bodies S_1 and S_2 produce the same output sequence when started in states with crash flags not set and whose value stores agree on values of variables in $OVar(S_1) \cup OVar(S_2)$, $\forall m_{S_1}(\mathfrak{f}_1, \sigma_{S_1})\, m_{S_2}(\mathfrak{f}_2, \sigma_{S_2})$:*
  $((\mathfrak{f}_1 = \mathfrak{f}_2 = 0) \wedge (\forall x \in OVar(S_1) \cup OVar(S_2) : \sigma_{S_1}(x) = \sigma_{S_2}(x))) \Rightarrow$
  $((S_1, m_{S_1}(\mathfrak{f}_1, \sigma_{S_1})) \equiv_O (S_2, m_{S_2}(\mathfrak{f}_2, \sigma_{S_2})))$.

If s_1 and s_2 start in states $m_1(\mathfrak{f}_1, loop^1_c, \sigma_1), m_2(\mathfrak{f}_2, loop^2_c, \sigma_2)$ respectively with crash flags not set $\mathfrak{f}_1 = \mathfrak{f}_2 = 0$ and in which s_1 and s_2 have not started execution $(loop^1_c(n_1) = loop^2_c(n_2) = 0)$, value stores σ_1 and σ_2 agree on values of variables in $OVar(s)$, $\forall x \in OVar(s) : \sigma_1(x) = \sigma_2(x)$, then one of the followings holds:

1. *s_1 and s_2 both terminate and produce the same output sequence:*
   $(s_1, m_1) \xrightarrow{*} (skip, m'_1(\sigma'_1))$, $(s_2, m_2) \xrightarrow{*} (skip, m'_2(\sigma'_2))$ where $\sigma'_1(id_{IO}) = \sigma'_2(id_{IO})$.
2. *s_1 and s_2 both do not terminate, $\forall k > 0$, $(s_1, m_1) \xrightarrow{k} (S_{1_k}, m_{1_k})$, $(s_2, m_2) \xrightarrow{k} (S_{2_k}, m_{2_k})$ where $S_{1_k} \neq skip$, $S_{2_k} \neq skip$ and one of the followings holds:*
   (a) *For any positive integer i, there are two configurations (s_1, m_{1_i}) and (s_2, m_{2_i}) reachable from (s_1, m_1) and (s_2, m_2), respectively, in which both crash flags are not set, the loop counters of s_1 and s_2 are equal to i and value stores agree on values of variables in $OVar(s)$, and for every state in execution, $(s_1, m_1) \xrightarrow{*} (s_1, m_{1_i})$ or $(s_2, m_2) \xrightarrow{*} (s_2, m_{2_i})$, loop counters for s_1 and s_2 are less than or equal to i respectively:*

$\forall i > 0\, \exists (s_1, m_{1_i})\, (s_2, m_{2_i}) : (s_1, m_1) \xrightarrow{*} (s_1, m_{1_i}(\mathfrak{f}_1, loop_c^{1_i}, \sigma_{1_i})) \wedge (s_2, m_2) \xrightarrow{*} (s_2, m_{2_i}(\mathfrak{f}_2, loop_c^{2_i}, \sigma_{2_i}))$ where

- *$\mathfrak{f}_1 = \mathfrak{f}_2 = 0$; and*
- *$loop_c^{1_i}(n_1) = loop_c^{2_i}(n_2) = i$; and*
- *$\forall x \in OVar(s) : \sigma_{1_i}(x) = \sigma_{2_i}(x)$.*
- *$\forall m_1' : (s_1, m_1) \xrightarrow{*} (S_1', m_1'(loop_c^{1'})) \xrightarrow{*} (s_1, m_{1_i}(loop_c^{1_i}, \sigma_{1_i}))$, $loop_c^{1'}(n_1) \leq i$; and*
- *$\forall m_2' : (s_2, m_2) \xrightarrow{*} (S_2', m_2'(loop_c^{2'})) \xrightarrow{*} (s_2, m_{2_i}(loop_c^{2_i}, \sigma_{2_i}))$, $loop_c^{2'}(n_2) \leq i$;*

(b) *The loop counters for s_1 and s_2 are less than a smallest positive integer i and all of the followings hold:*

- *$\exists i > 0\, \forall m_1', m_2' : (s_1, m_1) \xrightarrow{*} (S_1', m_1'(loop_c^{1'}))$, $(s_2, m_2) \xrightarrow{*} (S_2', m_2'(loop_c^{2'}))$ where $loop_c^{1'}(n_1) < i$, $loop_c^{2'}(n_2) < i$;*
- *$\forall 0 < j < i$, there are two configurations (s_1, m_{1_j}) and (s_2, m_{2_j}) reachable from (s_1, m_1) and (s_2, m_2), respectively, in which both crash flags are not set, loop counters of s_1 and s_2 are equal to j and value stores agree on values of variables in OVar(s):*
  $\exists (s_1, m_{1_j}), (s_2, m_{2_j}) : (s_1, m_1) \xrightarrow{*} (s_1, m_{1_j}(\mathfrak{f}_1, loop_c^{1_j}, \sigma_{1_j})) \wedge (s_2, m_2) \xrightarrow{*} (s_2, m_{2_j}(\mathfrak{f}_2, loop_c^{2_j}, \sigma_{2_j}))$ where
  - *$\mathfrak{f}_1 = \mathfrak{f}_2 = 0$; and*
  - *$loop_c^{1_j}(n_1) = loop_c^{2_j}(n_2) = j$; and*
  - *$\forall x \in OVar(s) : \sigma_{1_j}(x) = \sigma_{2_j}(x)$.*
- *If $i = 1$, then the I/O sequence is not redefined in any states reachable from (s_1, m_1) and (s_2, m_2).*
  - *$\forall m_1'' : (s_1, m_1(loop_c^1, \sigma_1)) \xrightarrow{*} (S_1'', m_1''(\sigma_1''))$ where $\sigma_1''(id_{IO}) = \sigma_1(id_{IO})$.*
  - *$\forall m_2'' : (s_2, m_2(loop_c^2, \sigma_2)) \xrightarrow{*} (S_2'', m_2''(\sigma_2''))$ where $\sigma_2''(id_{IO}) = \sigma_2(id_{IO})$.*
- *If $i > 1$, then the I/O sequence is not redefined in any states reachable from $(s_1, m_{1_{i-1}})$ and $(s_2, m_{2_{i-1}})$.*
  - *$\forall m_1'' : (s_1, m_{1_{i-1}}(loop_c^{1_{i-1}}, \sigma_{1_{i-1}})) \xrightarrow{*} (S_1'', m_1''(\sigma_1''))$ where $\sigma_1''(id_{IO}) = \sigma_{1_{i-1}}(id_{IO})$.*
  - *$\forall m_2'' : (s_2, m_{2_{i-1}}(loop_c^{2_{i-1}}, \sigma_{2_{i-1}})) \xrightarrow{*} (S_2'', m_2''(\sigma_2''))$ where $\sigma_2''(id_{IO}) = \sigma_{2_{i-1}}(id_{IO})$.*

(c) *The loop counters for s_1 and s_2 are less than or equal to a smallest positive integer i and all of the followings hold:*

- *$\exists i > 0\, \forall m_1', m_2' : (s_1, m_1) \xrightarrow{*} (S_1', m_1'(loop_c^{1'}))$, $(s_2, m_2) \xrightarrow{*} (S_2', m_2'(loop_c^{2'}))$ where $loop_c^{1'}(n_1) \leq i$, $loop_c^{2'}(n_2) \leq i$;*
- *There are no configurations (s_1, m_{1_i}) and (s_2, m_{2_i}) reachable from (s_1, m_1) and (s_2, m_2), respectively, in which crash flags are not set, the loop counters of s_1 and s_2 are equal to i, and value stores agree on values of variables in OVar(s):*
  $\nexists (s_1, m_{1_i}), (s_2, m_{2_i}) : (s_1, m_1) \xrightarrow{*} (s_1, m_{1_i}(\mathfrak{f}_1, loop_c^{1_i}, \sigma_{1_i})) \wedge (s_2, m_2) \xrightarrow{*} (s_2, m_{2_i}(\mathfrak{f}_2, loop_c^{2_i}, \sigma_{2_i}))$ where
  - *$\mathfrak{f}_1 = \mathfrak{f}_2 = 0$; and*
  - *$loop_c^{1_i}(n_1) = loop_c^{2_i}(n_2) = i$; and*
  - *$\forall x \in OVar(s) : \sigma_{1_i}(x) = \sigma_{2_i}(x)$.*
- *$\forall 0 < j < i$, there are two configurations (s_1, m_{1_j}) and (s_2, m_{2_j}) reachable from (s_1, m_1) and (s_2, m_2), respectively, in which crash flags are not set, the loop counters of s_1 and s_2 are equal to j and value stores agree on values of variables in OVar(s):*
  $\exists (s_1, m_{1_j}), (s_2, m_{2_j}) : (s_1, m_1) \xrightarrow{*} (s_1, m_{1_j}(\mathfrak{f}_1, loop_c^{1_j}, \sigma_{1_j})) \wedge (s_2, m_2) \xrightarrow{*} (s_2, m_{2_j}(\mathfrak{f}_2, loop_c^{2_j}, \sigma_{2_j}))$ where
  - *$\mathfrak{f}_1 = \mathfrak{f}_2 = 0$; and*
  - *$loop_c^{1_j}(n_1) = loop_c^{2_j}(n_2) = j$; and*
  - *$\forall x \in OVar(s) : \sigma_{1_j}(x) = \sigma_{2_j}(x)$.*
- *If $i = 1$, then executions from (s_1, m_1) and (s_2, m_2) produce the same output sequence:*
  $(s_1, m_1(loop_c^1, \sigma_1)) \equiv_O (s_2, m_2(loop_c^2, \sigma_2))$.
- *If $i > 1$, then executions from $(s_1, m_{1_{i-1}})$ and $(s_2, m_{2_{i-1}})$ produce the same output sequence:*
  $(s_1, m_{1_{i-1}}(loop_c^{1_{i-1}}, \sigma_{1_{i-1}})) \equiv_O (s_2, m_{2_{i-1}}(loop_c^{2_{i-1}}, \sigma_{2_{i-1}}))$.

Proof. We show that s_1 and s_2 terminate in the same way when started in states $m_1(\mathfrak{f}_1, \text{loop}_c^1, \sigma_1)$ and $m_2(\mathfrak{f}_2, \text{loop}_c^2, \sigma_2)$ respectively, $(s_1, m_1) \equiv_H (s_2, m_2)$. In addition, we show that s_1 and s_2 produce the same output sequence in every possibilities of termination in the same way, $(s_1, m_1) \equiv_O (s_2, m_2)$.

By definition, s_1 and s_2 satisfy the proof rule of termination in the same way because

- Loop bodies S_1 and S_2 satisfy the proof rule of termination in the same way;
  By assumption.
- Loop bodies S_1 and S_2 satisfy the proof rule of equivalent computation of those in the termination deciding variables of s_1 and s_2, $\forall x \in \text{TVar}(s_1) \cup \text{TVar}(s_2) : S_1 \equiv_x^S S_2$;
  By the definition of OVar(s), $\text{TVar}_o(s_1) \subseteq \text{OVar}(s_1)$ and $\text{TVar}_o(s_2) \subseteq \text{OVar}(s_2)$. By the definition of TVar$_o$, $\text{TVar}_o(s_1) = \text{TVar}(s_1)$ and $\text{TVar}_o(s_2) = \text{TVar}(s_2)$.

By Lemma 5.16, we show s_1 and s_2 terminate in the same way when started in states $m_1(\mathfrak{f}_1, \text{loop}_c^1, \sigma_1)$ and $m_2(\mathfrak{f}_2, \text{loop}_c^2, \sigma_2)$. We need to show that all the required conditions are satisfied.

- Crash flags are not set, $\mathfrak{f}_1 = \mathfrak{f}_2 = 0$;
- Loop counters of s_1 and s_2 are initially zero, $\text{loop}_c^1(n_1) = \text{loop}_c^2(n_2) = 0$;
- s_1 and s_2 have same set of termination deciding variables, $\text{TVar}(s_1) = \text{TVar}(s_2) = \text{TVar}(s)$;
- Value stores σ_1 and σ_2 agree on values of variables in $\text{TVar}(s_1) = \text{TVar}(s_2)$, $\forall x \in \text{TVar}(s) : \sigma_1(x) = \sigma_2(x)$;
  The above four conditions are from assumption.
- Loop bodies S_1 and S_2 terminate in the same way when started in states with crash flags not set and whose value stores agree on values of variables in $\text{TVar}(S_1) \cup \text{TVar}(S_2)$;
  By Theorem 4.

Therefore, by Lemma 5.16, we have one of the followings holds:

1. s_1 and s_2 both terminate and the loop counters of s_1 and s_2 are less than a positive integer i such that the loop counters of s_1 and s_2 are less than or equal to $i - 1$:
   $(s_1, m_1) \xrightarrow{*} (\text{skip}, m_1')$, $(s_2, m_2) \xrightarrow{*} (\text{skip}, m_2')$.
   We show that, when s_1 and s_2 terminate, value stores of s_1 and s_2 agree on the value of the I/O sequence variable by Lemma 5.2. We need to show all the required conditions hold.
   - $\forall x \in \text{Imp}(io) : \sigma_1(x) = \sigma_2(x)$;
   - $\text{loop}_c^1(n_1) = \text{loop}_c^2(n_2) = 0$;
     The above two conditions are from assumption.
   - $id_{IO} \in \text{Def}(s_1) \cap \text{Def}(s_2)$;
     Because there are output statements in s_1 and s_2. By the definition of $\text{Def}(\cdot)$, the I/O sequence variable is defined in s_1 and s_2.

- Imp$(s_1, \{id_{IO}\})$ = Imp$(s_2, \{id_{IO}\})$ = Imp(io);
  By the definition of Imp$_o(\cdot)$, Imp$_o(s_1)$ = Imp$(s_1, \{id_{IO}\})$, Imp$_o(s_2)$ = Imp$(s_2, \{id_{IO}\})$.
- $\forall y \in$ Imp$(io), \forall m_{S_1}(\sigma_{S_1})\, m_{S_2}(\sigma_{S_2})$:
  $((\forall z \in$ Imp$(S_1,$ Imp$(io)) \cup$ Imp$(S_2,$ Imp$(io)), \sigma_{S_1}(z) = \sigma_{S_2}(z)) \Rightarrow (S_1, m_{S_1}(\sigma_{S_1})) \equiv_y (S_2, m_{S_2}(\sigma_{S_2})))$.
  By Theorem 2.

In addition, by the semantic rules, the I/O sequence is appended at most by one value in one step. Hence, s_1 and s_2 produce the same output sequence when started in states m_1 and m_2 respectively.

2. s_1 and s_2 both do not terminate, $\forall k > 0$, $(s_1, m_1) \xrightarrow{k} (S_{1_k}, m_{1_k})$, $(s_2, m_2) \xrightarrow{k} (S_{2_k}, m_{2_k})$ where $S_{1_k} \neq$ skip, $S_{2_k} \neq$ skip and one of the followings holds:
   (a) $\forall i > 0$, there are two configurations (s_1, m_{1_i}) and (s_2, m_{2_i}) reachable from (s_1, m_1) and (s_2, m_2), respectively, in which both crash flags are not set, the loop counters of s_1 and s_2 are equal to i and value stores agree on the values of variables in TVar(s):
   $\forall i > 0\, \exists (s_1, m_{1_i})\, (s_2, m_{2_i})$: $(s_1, m_1) \xrightarrow{*} (s_1, m_{1_i}(\mathfrak{f}_1,$ loop$_c^{1_i}, \sigma_{1_i})) \wedge (s_2, m_2) \xrightarrow{*} (s_2, m_{2_i}(\mathfrak{f}_2,$ loop$_c^{2_i}, \sigma_{2_i}))$ where
   - $\mathfrak{f}_1 = \mathfrak{f}_2 = 0$; and
   - loop$_c^{1_i}(n_1)$ = loop$_c^{2_i}(n_2) = i$; and
   - $\forall x \in$ TVar(s) : $\sigma_{1_i}(x) = \sigma_{2_i}(x)$.
   - $\forall m_1'$: $(s_1, m_1) \xrightarrow{*} (S_1', m_1'($loop$_c^{1'})) \xrightarrow{*} (s_1, m_{1_i}($loop$_c^{1_i}, \sigma_{1_i}))$, loop$_c^{1'}(n_1) \leq i$; and
   - $\forall m_2'$: $(s_2, m_2) \xrightarrow{*} (S_2', m_2'($loop$_c^{2'})) \xrightarrow{*} (s_2, m_{2_i}($loop$_c^{2_i}, \sigma_{2_i}))$, loop$_c^{2'}(n_2) \leq i$;

   We show that, for any positive integer i, value stores σ_{1_i} and σ_{2_i} agree on values of variables in Imp(io) by the proof of Lemma 5.1. We need to show that all the required conditions are satisfied.
   - $\forall x \in$ Imp(io) : $\sigma_1(x) = \sigma_2(x)$;
   - loop$_c^1(n_1)$ = loop$_c^2(n_2) = 0$;
     The above two conditions are by assumption.
   - $id_{IO} \in$ Def$(s_1) \cap$ Def(s_2);
   - Imp$(s_1, \{id_{IO}\})$ = Imp$(s_2, \{id_{IO}\})$ = Imp(io);
     The above two conditions are obtained by similar argument in the case that s_1 and s_2 both terminate.
   - $\forall y \in$ Imp$(io), \forall m_{S_1}(\sigma_{S_1})\, m_{S_2}(\sigma_{S_2})$:
     $((\forall z \in$ Imp$(S_1,$ Imp$(io))\cup$Imp$(S_2,$ Imp$(io)), \sigma_{S_1}(z) = \sigma_{S_2}(z)) \Rightarrow$
     $(S_1, m_{S_1}(\sigma_{S_1})) \equiv_y (S_2, m_{S_2}(\sigma_{S_2})))$.
     By Theorem 2.

   We cannot apply Lemma 5.1 directly because s_1 and s_2 do not terminate. But we can still have the proof closely similar to that of Lemma 5.1 by using the fact that there exists a configuration of arbitrarily large loop counters of s_1 and s_2 and in which crash flags are not set.
   Then, $\forall i > 0, \forall x \in$ Imp(io) : $\sigma_{1_i}(x) = \sigma_{2_i}(x)$. In addition, by the semantic rules, the I/O sequence is appended at most by one value in one step. The lemma holds.

   (b) The loop counters for s_1 and s_2 are less than a smallest positive integer i and all of the followings hold:
   - $\exists i > 0\, \forall m_1', m_2'$: $(s_1, m_1) \xrightarrow{*} (S_1', m_1'($loop$_c^{1'})), (s_2, m_2) \xrightarrow{*} (S_2', m_2'($loop$_c^{2'}))$ where loop$_c^{1'}(n_1) < i$, loop$_c^{2'}(n_2) < i$;
   - $\forall 0 < j < i$, there are two configurations (s_1, m_{1_j}) and (s_2, m_{2_j}) reachable from (s_1, m_1) and (s_2, m_2), respectively, in which both crash flags are not set, the loop counters of s_1 and s_2 are equal to j and value stores agree on the values of variables in TVar(s):
     $\exists (s_1, m_{1_j}), (s_2, m_{2_j})$: $(s_1, m_1) \xrightarrow{*} (s_1, m_{1_j}(\mathfrak{f}_1,$ loop$_c^{1_j}, \sigma_{1_j})) \wedge (s_2, m_2) \xrightarrow{*} (s_2, m_{2_j}(\mathfrak{f}_2,$ loop$_c^{2_j}, \sigma_{2_j}))$ where
     - $\mathfrak{f}_1 = \mathfrak{f}_2 = 0$; and
     - loop$_c^{1_j}(n_1)$ = loop$_c^{2_j}(n_2) = j$; and
     - $\forall x \in$ TVar(s) : $\sigma_{1_j}(x) = \sigma_{2_j}(x)$.

   This case corresponds to the situation that the ith evaluations of the predicate expression of s_1 and s_2 raise an exception. There are two possibilities regarding the value of i.
   i. $i = 1$.
      s_1 and s_2 raise an exception in the 1st evaluation of their predicate expression because loop counters of s_1 and s_2 are less than 1. In the proof of Lemma 5.16, value stores σ_1 and σ_2 in states m_1 and m_2 respectively are not modified in the 1st evaluation of the predicate expression of s_1 and s_2. In addition, value stores σ_1 and σ_2 are not modified after s_1 and s_2 both crash according to the rule Crash. We have the corresponding initial state in which value stores σ_1 and σ_2 agree on values of variables in OVar(s). Thus, $\sigma_1(id_{IO}) = \sigma_2(id_{IO})$. The lemma holds.
   ii. $i > 1$.
      We show that, for any positive integer $0 < j < i$, value stores σ_{1_j} and σ_{2_j} agree on values of variables in Imp(s) by the proof of Lemma 5.1. We need to show that all the required conditions are satisfied.
      - $\forall x \in$ Imp(io) : $\sigma_1(x) = \sigma_2(x)$;
      - loop$_c^1(n_1)$ = loop$_c^2(n_2) = 0$;
        The above two conditions are from assumption.
      - $id_{IO} \in$ Def$(s_1) \cap$ Def(s_2);
      - Imp$(s_1, \{id_{IO}\})$ = Imp$(s_2, \{id_{IO}\})$ = Imp(io);
        The above two conditions are obtained by the same argument in the case that s_1 and s_2 terminate.
      - $\forall y \in$ Imp$(io), \forall m_{S_1}(\sigma_{S_1})\, m_{S_2}(\sigma_{S_2})$:
        $((\forall z \in$ Imp$(S_1,$ Imp$(io))\cup$Imp$(S_2,$ Imp$(io)), \sigma_{S_1}(z) = \sigma_{S_2}(z)) \Rightarrow$
        $(S_1, m_{S_1}(\sigma_{S_1})) \equiv_y (S_2, m_{S_2}(\sigma_{S_2})))$.
        By Theorem 2.

      We cannot apply Lemma 5.1 directly because s_1 and s_2 do not terminate. But we can still have the proof closely similar to that of Lemma 5.1 by using the fact that there are reachable configurations $(s_1, m_{1_{i-1}})$ and $(s_2, m_{2_{i-1}})$ with the loop counters of s_1 and s_2 of value $i-1$ and crash flags not set.
      By assumption, there is configuration $(s_1, m_{1_{i-1}}(\mathfrak{f}_1,$ loop$_c^{1_{i-1}}, \sigma_{1_{i-1}}))$ reachable from (s_1, m_1) in which the loop counter of s_1 is $i-1$ and the crash flag is not set; there is also a configuration $(s_2, m_{2_{i-1}}(\mathfrak{f}_2,$ loop$_c^{2_{i-1}}, \sigma_{2_{i-1}}))$ of s_2 reachable from (s_2, m_2) in which the loop counter is $i-1$ and the crash flag is not set. In addition, value stores $\sigma_{1_{i-1}}$ and $\sigma_{2_{i-1}}$ agree on values of variables in Imp(io). In the proof of Lemma 5.15, the ith evaluations of the predicate expression of s_1 and s_2 must raise an exception because loop counters of s_1 and s_2 are less than i. Then the I/O sequence is not redefined in any state reachable from $(s_1, m_{1_{i-1}}(\mathfrak{f}_1,$ loop$_c^{1_{i-1}}, \sigma_{1_{i-1}}))$ and $(s_2, m_{2_{i-1}}(\mathfrak{f}_2,$ loop$_c^{2_{i-1}}, \sigma_{2_{i-1}}))$ respectively. In addition, by the semantic rules, the I/O sequence is appended at most by one value in one step. The lemma holds.

(c) The loop counters for s_1 and s_2 are less than or equal to a smallest positive integer i and all of the followings hold:

- $\exists i > 0\, \forall m'_1, m'_2 \,:\, (s_1, m_1) \xrightarrow{*} (S'_1, m'_1(\text{loop}_c^{1'})), (s_2, m_2) \xrightarrow{*} (S'_2, m'_2(\text{loop}_c^{2'}))$ where $\text{loop}_c^{1'}(n_1) \le i$, $\text{loop}_c^{2'}(n_2) \le i$;
- There are no configurations (s_1, m_{1_i}) and (s_2, m_{2_i}) reachable from (s_1, m_1) and (s_2, m_2), respectively, in which crash flags are not set, the loop counters of s_1 and s_2 are equal to i, and value stores agree on values of variables in TVar(s):
  $\nexists (s_1, m_{1_i}), (s_2, m_{2_i}) \,:\, (s_1, m_1) \xrightarrow{*} (s_1, m_{1_i}(\mathfrak{f}_1, \text{loop}_c^{1_i}, \sigma_{1_i})) \wedge (s_2, m_2) \xrightarrow{*} (s_2, m_{2_i}(\mathfrak{f}_2, \text{loop}_c^{2_i}, \sigma_{2_i}))$ where
  - $\mathfrak{f}_1 = \mathfrak{f}_2 = 0$; and
  - $\text{loop}_c^{1_i}(n_1) = \text{loop}_c^{2_i}(n_2) = i$; and
  - $\forall x \in \text{TVar}(s) \,:\, \sigma_{1_i}(x) = \sigma_{2_i}(x)$.
- $\forall 0 < j < i$, there are two configurations (s_1, m_{1_j}) and (s_2, m_{2_j}) reachable from (s_1, m_1) and (s_2, m_2), respectively, in which both crash flags are not set, the loop counters of s_1 and s_2 are equal to j and value stores agree on values of variables in TVar(s):
  $\exists (s_1, m_{1_j}), (s_2, m_{2_j}) \,:\, (s_1, m_1) \xrightarrow{*} (s_1, m_{1_j}(\mathfrak{f}_1, \text{loop}_c^{1_j}, \sigma_{1_j})) \wedge (s_2, m_2) \xrightarrow{*} (s_2, m_{2_j}(\mathfrak{f}_2, \text{loop}_c^{2_j}, \sigma_{2_j}))$ where
  - $\mathfrak{f}_1 = \mathfrak{f}_2 = 0$; and
  - $\text{loop}_c^{1_j}(n_1) = \text{loop}_c^{2_j}(n_2) = j$; and
  - $\forall x \in \text{TVar}(s) \,:\, \sigma_{1_j}(x) = \sigma_{2_j}(x)$.

This case corresponds to the situation that, the ith evaluation of the predicate expression of s_1 and s_2 produce same nonzero integer value and loop bodies S_1 and S_2 do not terminate after the ith evaluation of the predicate expression of s_1 and s_2. There are two possibilities regarding the value of i.

i. $i = 1$.
By assumption, we have the initial value stores σ_1 and σ_2 agree on values of variables in OVar(s). In the proof of Lemma 5.15, the execution of s_1 proceeds as follows:

$$\begin{aligned}
&(s_1, m_1(\text{loop}_c^1, \sigma_1))\\
=\ &(\text{while}_{\langle n_1\rangle}(e)\ \{S_1\}, m_1(\text{loop}_c^1, \sigma_1))\\
\rightarrow &(\text{while}_{\langle n_1\rangle}(v)\ \{S_1\}, m_1(\text{loop}_c^1, \sigma_1)) \text{ by the EEval rule}\\
\rightarrow &(S_1; \text{while}_{\langle n_1\rangle}(e)\ \{S_1\}, m_1(\text{loop}_c^1[1/n_1], \sigma_1))\\
&\text{by the Wh-T rule.}
\end{aligned}$$

The execution of s_2 proceeds to $(S_2; \text{while}_{\langle n_2\rangle}(e)\ \{S_2\}, m_2(\text{loop}_c^2[1/n_2], \sigma_2))$. Then the execution of S_1 and S_2 do not terminate when started in states $m_1(\text{loop}_c^1[1/n_1], \sigma_1)$ and $m_2(\text{loop}_c^2[1/n_2], \sigma_2)$. By assumption, value stores σ_1 and σ_2 agree on values of the out-deciding variables of s_1 and s_2, $\forall x \in \text{OVar}(s_1) \cup \text{OVar}(s_2) \,:\, \sigma_1(x) = \sigma_2(x)$. By definition, $\text{Imp}(S_1, \{id_{IO}\}) \subseteq \text{Imp}(s_1, \{id_{IO}\})$, $\text{CVar}(S_1) \subseteq \text{CVar}(s_1)$ and $\text{LVar}(S_1) \subseteq \text{LVar}(s_1)$. Thus, $\text{TVar}(S_1) \subseteq \text{TVar}(s_1)$. By Lemma 5.20, $\text{TVar}_o(S_1) \subseteq \text{TVar}(S_1)$. By Lemma 5.18, $\text{Imp}_o(S_1) \subseteq \text{Imp}(S_1, id_{IO})$. In conclusion, $\text{OVar}(S_1) \subseteq \text{OVar}(s_1)$. Similarly, $\text{OVar}(S_2) \subseteq \text{OVar}(s_2)$. Thus, $\forall x \in \text{OVar}(S_1) \cup \text{OVar}(S_2) \,:\, \sigma_1(x) = \sigma_2(x)$. Then executions of S_1 and S_2 when started from states $m_1(\text{loop}_c^1[1/n_1], \sigma_1)$ and $m_2(\text{loop}_c^2[1/n_2], \sigma_2)$ produce the same output sequence:
$(S_1, m_1(\text{loop}_c^1[1/n_1], \sigma_1)) \equiv_O (S_2, m_2(\text{loop}_c^2[1/n_2], \sigma_2))$.
In addition, by the semantic rules, the I/O sequence is appended at most by one value in one step. The lemma holds.

ii. $i > 1$.
We show that, for any positive integer $0 < j < i$, value stores σ_{1_j} and σ_{2_j} agree on values of variables in Imp(io) by the proof of Lemma 5.1. We need to show that all the required conditions are satisfied.

- $\forall x \in \text{Imp}(io) \,:\, \sigma_1(x) = \sigma_2(x)$;
- $\text{loop}_c^1(n_1) = \text{loop}_c^2(n_2) = 0$;
  The above two conditions are from assumption.
- $id_{IO} \in \text{Def}(s_1) \cap \text{Def}(s_2)$;
- $\text{Imp}(s_1, \{id_{IO}\}) = \text{Imp}(s_2, \{id_{IO}\}) = \text{Imp}(io)$;
  The above two conditions are obtained by the same argument in the case that s_1 and s_2 terminate.
- $\forall y \in \text{Imp}(io), \forall m_{S_1}(\sigma_{S_1})\, m_{S_2}(\sigma_{S_2}) :$
  $((\forall z \in \text{Imp}(S_1, \text{Imp}(io)) \cup \text{Imp}(S_2, \text{Imp}(io)), \sigma_{S_1}(z) = \sigma_{S_2}(z)) \Rightarrow$
  $(S_1, m_{S_1}(\sigma_{S_1})) \equiv_y (S_2, m_{S_2}(\sigma_{S_2})))$.
  By Theorem 2.

We cannot apply Lemma 5.1 directly because s_1 and s_2 do not terminate. But we can still have the proof closely similar to that of Lemma 5.1. The reason is that there are reachable configurations $(S_1; s_1, m'_1)$ and $(S_2; s_2, m'_2)$ with loop counters of s_1 and s_2 of value i and crash flags not set. This is by Lemma 5.24 because there are configurations reachable from (s_1, m_1) and (s_2, m_2) respectively with loop counters of s_1 and s_2 of i.
There are configurations $(s_1, m_{1_{i-1}})$ reachable from (s_1, m_1) and $(s_2, m_{2_{i-1}})$ reachable from (s_2, m_2) in which loop counters of s_1 and s_2 are $i-1$ and crash flags are not set and value stores agree on values of variables in Imp(io). Because loop counters of s_1 and s_2 are less than or equal to i. Then the execution of s_1 proceeds as follows:

$$\begin{aligned}
&(s_1, m_{1_{i-1}}(\text{loop}_c^{1_{i-1}}, \sigma_{1_{i-1}}))\\
=\ &(\text{while}_{\langle n_1\rangle}(e)\ \{S_1\}, m_{1_{i-1}}(\text{loop}_c^{1_{i-1}}, \sigma_{1_{i-1}}))\\
\rightarrow &(\text{while}_{\langle n_1\rangle}(v)\ \{S_1\}, m_{1_{i-1}}(\text{loop}_c^{1_{i-1}}, \sigma_{1_{i-1}}))\\
&\text{by the EEval rule}\\
\rightarrow &(S_1; \text{while}_{\langle n_1\rangle}(e)\ \{S_1\}, m_{1_{i-1}}(\text{loop}_c^{1_{i-1}}[i/n_1],\\
&\sigma_{1_{i-1}})) \text{ by the Wh-T rule.}
\end{aligned}$$

The execution of s_2 proceeds to $(S_2; \text{while}_{\langle n_2\rangle}(e)\ \{S_2\}, m_{2_{i-1}}(\text{loop}_c^{2_{i-1}}[i/n_2], \sigma_{2_{i-1}}))$. By similar argument in the case $i = 1$, S_2 and S_1 produce the same output sequence when started in states $m_{1_{i-1}}(\text{loop}_c^{1_{i-1}}[i/n_1], \sigma_{1_{i-1}})$ and $m_{2_{i-1}}(\text{loop}_c^{2_{i-1}}[i/n_2], \sigma_{2_{i-1}})$ respectively. In addition, by the semantic rules, the I/O sequence is appended at most by one value in one step. The lemma holds.

□

Corollary 5.5. *Let s_1 = "while$_{\langle n_1\rangle}(e)$ $\{S_1\}$" and s_2 = "while$_{\langle n_2\rangle}(e)$ $\{S_2\}$" be two while statements such that all of the followings hold*

- *There are output statements in s_1 and s_2, $\exists e_1\, e_2 \,:\, (\text{"output } e_1\text{"} \in s_1) \wedge (\text{"output } e_2\text{"} \in s_2)$;*
- *s_1 and s_2 have same set of termination deciding variables and same set of imported variables relative to the I/O sequence variable, $(TVar(s_1) = TVar(s_2) = TVar(s)) \wedge (Imp(s_1, \{id_{IO}\}) = Imp(s_2, \{id_{IO}\}) = Imp(io))$;*
- *Loop bodies S_1 and S_2 satisfy the proof rule of equivalent computation of those in out-deciding variables of s_1 and s_2, $\forall x \in OVar(s) = TVar(s) \cup Imp(io) \,:\, S_1 \equiv_x^S S_2$;*
- *Loop bodies S_1 and S_2 satisfy the proof rule of termination in the same way, $S_1 \equiv_H^S S_2$;*

- *Loop bodies S_1 and S_2 produce the same output sequence when started in states with crash flags not set and agreeing on values of variables in OVar$(S_1) \cup$ OVar(S_2), $\forall m_{S_1}(\mathfrak{f}_1, \sigma_{S_1})\, m_{S_2}(\mathfrak{f}_2, \sigma_{S_2})$:*
  $((\mathfrak{f}_1 = \mathfrak{f}_2 = 0) \wedge (\forall x \in \mathit{OVar}(S_1) \cup \mathit{OVar}(S_2)\ :\ \sigma_{S_1}(x) = \sigma_{S_2}(x))) \Rightarrow ((S_1, m_{S_1}(\mathfrak{f}_1, \sigma_{S_1})) \equiv_O (S_2, m_{S_2}(\mathfrak{f}_2, \sigma_{S_2})))$.

If s_1 and s_2 start in states $m_1(\mathfrak{f}_1, loop_c^1, \sigma_1), m_2(\mathfrak{f}_2, loop_c^2, \sigma_2)$ respectively with crash flags not set $\mathfrak{f}_1 = \mathfrak{f}_2 = 0$ and in which s_1 and s_2 have not started execution ($loop_c^1(n_1) = loop_c^2(n_2) = 0$), value stores σ_1 and σ_2 agree on values of variables in OVar(s), $\forall x \in \mathit{OVar}(s)\ :\ \sigma_1(x) = \sigma_2(x)$, then s_1 and s_2 produce the same output sequence: $(s_1, m_1) \equiv_O (s_2, m_2)$.

This is from lemma 5.25.

5.5 Backward compatible DSU based on program equivalence

Based on the equivalence result above, we show that there exists backward compatible DSU. We need to show there exists a mapping of old program configurations and new program configurations and the hybrid execution obtained from the configuration mapping is backward compatible. We do not provide a practical algorithm to calculate the state mapping. Instead we only show that there exists new program configurations corresponding to some old program configurations via a simulation. The treatment in this section is informal.

The idea is to map a configuration just before an output is produced to a corresponding configuration. Based on the proof rule of same output sequences, not every statement of the old program can correspond to a statement of the new program, but every output statemet of the old program should correspond to an output statement of the new program. Consider configuration C_1 of the old program where the leftmost statement (next statement to execute) is an output statement. We can define a corresponding statement of the new program by *simulating* the execution of the new program on the input consumed so far in C_1. There are two cases. When the leftmost statement in C_1 is not included in a loop statement, then it is easy to know when to stop simulation. Otherwise, we have the bijection of loop statements including output statements based on the condition of same output sequences. Therefore, it is easy to know how many iterations of the loop statements including the output statement shall be carried out based on the loop counters in the old program configuration C_1. Based on Theorem 5, there must be a configuration C_2 corresponding to C_1. Moreover, the executions starting from configurations C_1 and C_2 produce the same output sequence based on Theorem 5. In conclusion, we obtain a backward compatible hybrid execution where the state mapping is from C_1 to C_2.

6. Real world backward compatible update classes: proof rules

We propose our formal treatment for real world update classes. For each update class, we show how the old program and new program produce the same I/O sequence which guarantees backward compatible DSU.

6.1 Proof rule for specializing new configuration variables

New configuration variables can be introduced to generalize functionality. Figure 15 shows an example of how a new configuration variable introduces new code. The two statement sequences in Figure 15 are equivalent when the new variable b is specialized to 0.

Our generalized formal definition of "specializing new configuration variables" is defined as follows.

```
1:                       1':  If (b) then
2:                       2':      output a * 2
3:                       3':  else
4:  output a + 2         4':      output a + 2

    old                       new
```

Figure 15: Specializing new configuration variables

Definition 25. (Specializing new configuration variables) *A statement sequence S_2 includes updates of specializing new configuration variables compared with S_1 w.r.t a mapping ρ of new configuration variables in S_2, $\rho : \{id\} \mapsto \{0, 1\}$, denoted $S_2 \approx_\rho^S S_1$, iff one of the following holds:*

1. *$S_2 =$ "If(id) then$\{S_2^t\}$ else$\{S_2^f\}$" where one of the following holds:*
   (a) $(\rho(id) = 0) \wedge (S_2^f \approx_\rho^S S_1)$;
   (b) $(\rho(id) = 1) \wedge (S_2^t \approx_\rho^S S_1)$;
2. *S_1 and S_2 produce the same output sequence, $S_1 \approx_O^S S_2$;*
3. *$S_1 =$ "If(e) then$\{S_1^t\}$ else$\{S_1^f\}$", $S_2 =$ "If(e) then$\{S_2^t\}$ else$\{S_2^f\}$" where $(S_2^t \approx_\rho^S S_1^t) \wedge (S_2^f \approx_\rho^S S_1^f)$;*
4. *$S_1 =$ "while$_{\langle n_1 \rangle}(e)\ \{S_1'\}$", $S_2 =$ "while$_{\langle n_2 \rangle}(e)\ \{S_2'\}$" where $S_2' \approx_\rho^S S_1'$;*
5. *$S_1 = S_1'; s_1$ and $S_2 = S_2'; s_2$ where $(S_2' \approx_\rho^S S_1') \wedge (S_2' \approx_H^S S_1') \wedge \big(\forall x \in \mathit{Imp}(s_1, id_{IO}) \cup \mathit{Imp}(s_1, id_{IO})\ :\ (S_2' \approx_x^S S_1')\big) \wedge (s_2 \approx_\rho^S s_1)$.*

Then we show that executions of two statement sequences produce the same I/O sequence if there are updates of specializing new configuration variables between the two.

Lemma 6.1. *Let S_1 and S_2 be two different statement sequences where there are updates of "specializing new configuration variables" in S_2 compared with S_1 w.r.t a mapping of new configuration variables ρ, $S_2 \approx_\rho^S S_1$. If executions of S_2 and S_1 start in states $m_2(\mathfrak{f}_2, \sigma_2)$ and $m_1(\mathfrak{f}_1, \sigma_1)$ respectively where all of the following hold:*

- *Crash flags $\mathfrak{f}_2, \mathfrak{f}_1$ are not set, $\mathfrak{f}_2 = \mathfrak{f}_1 = 0$;*
- *Value stores σ_1 and σ_2 agree on output deciding variables in both S_1 and S_2 including the input and I/O sequence variable, $\forall id \in (\mathit{OVar}(S_1) \cap \mathit{OVar}(S_2)) \cup \{id_I, id_{IO}\}$: $\sigma_1(id) = \sigma_2(id)$;*
- *Values of new configuration variables in the value store σ_2 are matching those in ρ, $\forall id \in \mathit{Dom}(\rho)\ :\ \rho(id) = \sigma_2(id)$;*
- *Values of new configuration variables are not defined in the statement sequence S_2, $\mathit{Dom}(\rho) \cap \mathit{Def}(S_2) = \emptyset$;*

then S_2 and S_1 satisfy all of the following:

- *$(S_1, m_1) \equiv_H (S_2, m_2)$;*
- *$(S_1, m_1) \equiv_O (S_2, m_2)$;*
- *$\forall x \in \{id_I, id_{IO}\}$: $(S_1, m_1) \equiv_x (S_2, m_2)$;*

Proof. The proof of Lemma 6.1 is by induction on the sum of program sizes of S_1 and S_2 and is a case analysis based on Definition 25.
Base case.
S_1 is a simple statement s, $S_2 =$ "If(id) then$\{s_2^t\}$ else$\{s_2^f\}$" where s_2^t, s_2^f are simple statement and one of the following holds:

1. $(\rho(id) = 0) \wedge (s_2^f = s)$;
2. $(\rho(id) = 1) \wedge (s_2^t = s)$;

W.l.o.g., we assume that $\rho(id) = 0$. By assumption, $\sigma_2(id) = \rho(id) = 0$. Then the execution of S_2 proceeds as follows:

$$\begin{aligned}
&(\text{If}(id)\text{ then}\{s_2^t\}\text{ else}\{s_2^f\}, m_2(\sigma_2))\\
\rightarrow&(\text{If}(0)\text{ then}\{s_2^t\}\text{ else}\{s_2^f\}, m_2(\sigma_2))\\
&\text{by the rule Var}\\
\rightarrow&(s_2^f, m_2(\sigma_2)) \text{ by the If-F rule.}
\end{aligned}$$

By Theorem 5 and Theorem 4, this lemma holds.
Induction step.

The induction hypothesis (IH) is that Lemma 6.1 holds when the sum of the program size of S_1 and S_2 is at least 4, size(S_1) + size(S_2) $= k \geq 4$.

Then we show that the lemma holds when size(S_1)+size(S_2) $= k + 1$. There are cases to consider.

1. S_1 and S_2 satisfy the condition of same output sequence, $S_1 \equiv_O^S S_2$.
   By Theorem 5, the lemma 6.1 holds.
2. S_1 and S_2 are both "If" statement:
   S_1 = "If(e) then$\{S_1^t\}$ else$\{S_1^f\}$", S_2 = "If(e) then$\{S_2^t\}$ else$\{S_2^f\}$" where both of the following hold
   - $S_2^t \approx_\rho^S S_1^t$;
   - $S_2^f \approx_\rho^S S_1^f$;

   By the definition of Use(S_1), variables used in the predicate expression e are a subset of used variables in S_1 and S_2, Use(e) $\subseteq$ Use(S_1) $\cap$ Use(S_2). By assumption, corresponding variables used in e are of same value in value stores σ_1 and σ_2. By Lemma D.1, the expression evaluates to the same value w.r.t value stores σ_1 and σ_2. There are three possibilities.
   (a) The evaluation of e crashes, $\mathcal{E}'[\![e]\!]\sigma_1 = \mathcal{E}'[\![e]\!]\sigma_2 = (\text{error}, v_{\mathfrak{of}})$.
   The execution of S_1 continues as follows:
   $$\begin{aligned}
   &(\text{If}(e)\text{ then}\{S_1^t\}\text{ else}\{S_1^f\}, m_1(\sigma_1))\\
   \rightarrow&(\text{If}((\text{error}, v_{\mathfrak{of}}))\text{ then}\{S_1^t\}\text{ else}\{S_1^f\}, m_1(\sigma_1))\\
   &\text{by the rule EEval'}\\
   \rightarrow&(\text{If}(0)\text{ then}\{S_1^t\}\text{ else}\{S_1^f\}, m_1(1/\mathfrak{f}))\\
   &\text{by the ECrash rule}\\
   \xrightarrow{i}&(\text{If}(0)\text{ then}\{S_1^t\}\text{ else}\{S_1^f\}, m_1(1/\mathfrak{f})) \text{ for any } i > 0\\
   &\text{by the Crash rule.}
   \end{aligned}$$
   Similarly, the execution of S_2 started from the state $m_2(\sigma_2)$ crashes. The lemma holds.
   (b) The evaluation of e reduces to zero, $\mathcal{E}'[\![e]\!]\sigma_1 = \mathcal{E}'[\![e]\!]\sigma_2 = (0, v_{\mathfrak{of}})$.
   The execution of S_1 continues as follows.
   $$\begin{aligned}
   &(\text{If}(e)\text{ then}\{S_1^t\}\text{ else}\{S_1^f\}, m_1(\sigma_1))\\
   =\ &(\text{If}((0, v_{\mathfrak{of}}))\text{ then}\{S_1^t\}\text{ else}\{S_1^f\}, m_1(\sigma_1))\\
   &\text{by the rule EEval'}\\
   \rightarrow&(\text{If}(0)\text{ then}\{S_1^t\}\text{ else}\{S_1^f\}, m_1(\sigma_1))\\
   &\text{by the E-Oflow1 or E-Oflow2 rule}\\
   \rightarrow&(S_1^f, m_1(\sigma_1)) \text{ by the If-F rule.}
   \end{aligned}$$
   Similarly, the execution of S_2 gets to the configuration $(S_2^f, m_2(\sigma_2))$.
   By the hypothesis IH, we show the lemma holds. We need to show that all conditions are satisfied for the application of the hypothesis IH.
   - $(S_2^f \approx_\rho^S S_1^f)$
     By assumption.
   - The sum of the program size of S_1^f and S_2^f is less than k, size(S_1^f) + size(S_2^f) $< k$.
     By definition, size(S_1) = 1+size(S_1^t)+size(S_1^f). Then, size(S_1^f) + size(S_2^f) $< k + 1 - 2 = k - 1$.
   - Value stores σ_1 and σ_2 agree on values of used variables in S_1^f and S_2^f as well as the input, I/O sequence variable.
     By definition, Use(S_1^f) $\subseteq$ Use(S_1). So are the cases to S_2^f and S_2. In addition, value stores σ_1 and σ_2 are not changed in the evaluation of the predicate expression e. The condition holds.
   - Values of new configuration variables are consistent in the value store σ_2 and the specialization ρ, $\forall id \in$ Dom(ρ) : $\sigma_2(id) = \rho(id)$.
     By assumption.

   By the hypothesis IH, the lemma holds.
   (c) The evaluation of e reduces to the same nonzero integer value, $\mathcal{E}'[\![e]\!]\sigma_1 = \mathcal{E}'[\![e]\!]\sigma_2 = (v, v_{\mathfrak{of}})$ where $v \neq 0$.
   By arguments similar to the second subcase above.
3. S_1 and S_2 are both "while" statements:
   S_1 = "while$_{\langle n\rangle}(e)\ \{S_1'\}$", S_2 = "while$_{\langle n\rangle}(e)\ \{S_2'\}$" where $S_2' \approx_\rho^S S_1'$;
   By Lemma 6.3, we show this lemma holds. We need to show that all required conditions are satisfied for the application of Lemma 6.3.
   - S_1 and S_2 have same set of output deciding variables, OVar(S_1) = OVar(S_2) = OVar(S);
     By Lemma 6.2 and Corollary 5.1.
   - When started in states $m_1'(\sigma_1'), m_2'(\sigma_1')$ where value stores σ_1' and σ_2' agree on values of output deciding variables in both S_1 and S_2 as well as the input sequence variable and the I/O sequence variable, then S_1' and S_2' terminate in the same way, produce the same output sequence, and have equivalent computation of defined variables in both S_1 and S_2.
     By the induction hypothesis IH. This is because the sum of the program size of S_1' and S_2' is less than k. By definition, size(S_1) = 1 + size(S_1').

   By Lemma 6.3, this lemma holds.
4. S_2 = "If(id) then $\{S_2^t\}$ else $\{S_2^f\}$" where one of the following holds:
   (a) $(\rho(id) = 0) \wedge (S_2^f \approx_\rho^S S_1)$;
   (b) $(\rho(id) = 1) \wedge (S_2^t \approx_\rho^S S_1)$;
   W.l.o.g, we assume $(\rho(id) = 0) \wedge (S_2^f \approx_\rho^S S_1)$;
   Then the execution of S_2 proceeds as follows:
   $$\begin{aligned}
   &(\text{If}(id)\text{ then}\{S_2^t\}\text{ else}\{S_2^f\}, m_2(\sigma_2))\\
   =\ &(\text{If}(0)\text{ then}\{S_2^t\}\text{ else}\{S_2^f\}, m_2(\sigma_2))\\
   &\text{by the Var rule}\\
   \rightarrow&(S_2^f, m_2(\sigma_2))\\
   &\text{by the If-F rule}
   \end{aligned}$$
   By the induction hypothesis, we show that the lemma holds. We need to show the required conditions are satisfied for the application of the hypothesis.
   - $S_2^f \approx_\rho^S S_1$
     By assumption.
   - The sum of the program size of S_1^f and S_2^f is less than k, size(S_1^f) + size(S_2^f) $< k$.
     By definition, size(S_2) = 1 + size(S_2^t) + size(S_2^f). Then, size(S_2^f) + size(S_1) $< k + 1 - 1 -$ size(S_2^t) $< k$.
   - Value stores σ_1 and σ_2 agree on values of used variables in S_2^f and S_1 as well as the input, I/O sequence variable.
     By definition, Use(S_2^f) $\subseteq$ Use(S_2). In addition, the value store σ_2 is not changed in the evaluation of the predicate expression e. The condition holds.

   By the hypothesis IH, the lemma holds.

5. S_1 and S_2 are same, $S_1 = S_2$;
By definition, used variables in S_1 and S_2 are same; defined variables in S_1 and S_2 are same. By semantic rules, S_1 and S_2 terminate in the same way, produce the same output sequence and have equivalent computation of defined variables in S_1 and S_2. This lemma holds.
6. $S_1 = S_1'; s_1$ and $S_2 = S_2'; s_2$ where both of the following hold:
   - $S_2' \approx_\rho^S S_1'$;
   - $s_2 \approx_\rho^S s_1$;

   By Theorem 4 and the hypothesis IH, we show S_2' and S_1' terminate in the same way and produce the same output sequence and when S_2' and S_1' both terminate, S_2' and S_1' have equivalent terminating computation of variables used or defined in S_2' and S_1'.
   We show all the required conditions are satisfied for the application of the hypothesis IH.
   - $S_2' \approx_\rho^S S_1'$.
     By assumption.
   - The sum of the program size of S_1' and S_2' is less than k, $\text{size}(S_1') + \text{size}(S_2') < k$.
     By definition, $\text{size}(S_2) = \text{size}(s_2) + \text{size}(S_2')$ where $\text{size}(s_2) < 1$. Then, $\text{size}(S_2') + \text{size}(S_1') < k + 1 - \text{size}(s_2) - \text{size}(s_1) < k$.
   - Value stores σ_1 and σ_2 agree on values of output deciding variables in S_2' and S_1' including the input, I/O sequence variable.
     By definition of TVar_o and Imp_o, $\text{OVar}(S_2') \subseteq \text{OVar}(S_2)$. The condition holds.
   - Values of new configuration variables are consistent in the value store σ_2 and the specialization ρ, $\forall id \in \text{Dom}(\rho) : \sigma_2(id) = \rho(id)$.
     By assumption.

   By the hypothesis IH, one of the following holds:
   (a) S_1' and S_2' both do not terminate.
   By Lemma E.2, executions of $S_1 = S_1'; s_1$ and $S_2 = S_2'; s_2$ both do not terminate and produce the same output sequence.
   (b) S_1' and S_2' both terminate.
   By assumption, $(S_2', m_2(\sigma_2)) \xrightarrow{*} (\text{skip}, m_2'(\sigma_2'))$, $(S_1', m_1(\sigma_1)) \xrightarrow{*} (\text{skip}, m_1'(\sigma_1'))$.
   By Corollary E.1, $(S_2'; s_2, m_2(\sigma_2)) \xrightarrow{*} (s_2, m_2'(\sigma_2'))$, $(S_1'; s_1, m_1(\sigma_1)) \xrightarrow{*} (s_1, m_1'(\sigma_1'))$.
   By the hypothesis IH, we show that s_2 and s_1 terminate in the same way, produce the same output sequence and when s_2 and s_1 both terminate, s_2 and s_1 have equivalent computation of variables used or defined in s_1 and s_2 and the input, and I/O sequence variables.
   We need to show that all conditions are satisfied for the application of the hypothesis IH.
   - There are updates of "new configuration variables" between s_2 and s_1;
     By assumption, $s_2 \approx_\rho^S s_1$.
   - The sum of the program size s_2 and s_1 is less than or equals to k;
     By definition, $\text{size}(S_2') \geq 1, \text{size}(S_1') \geq 1$. Therefore, $\text{size}(s_2) + \text{size}(s_1) < k + 1 - \text{size}(S_2') - \text{size}(S_1') \leq k$.
   - Value stores σ_1' and σ_2' agree on values of output deciding variables in s_2 and s_1 as well as the input, I/O sequence variable.
     By induction hypothesis IH, $\text{OVar}(s_1) \subseteq \text{OVar}(s_2)$, then $\text{Use}(s_2) \cap \text{Use}(s_1) = \text{Use}(s_1)$. For any variable id in $\text{OVar}(s_1)$, if id is in $\text{OVar}(S_1')$, then the value of id is same after the execution of S_1' and S_2', $\sigma_1'(id) = \sigma_2(id) = \sigma_2'(id)$. Otherwise, the variable id is defined in the execution of S_1' and S_2', by assumption, $\sigma_1'(id) = \sigma_2'(id)$. The condition holds.
   - Values of new configuration variables are consistent in the value store σ_2' and the specialization ρ, $\forall id \in \text{Dom}(\rho) : \sigma_2'(id) = \rho(id)$.
     By assumption, $\text{Dom}(\rho) \cap \text{Def}(S_2)$. By Corollary E.2, values of new configuration variables are not changed in the execution of S_2', $\forall id \in \text{Dom}(\rho) : \sigma_2'(id) = \sigma_2(id) = \rho(id)$.

   By the hypothesis IH, the lemma holds.

□

We list properties of the update of new configuration variables and the proof of backward compatibility for the case of loop statement as follows. We present one auxiliary lemma used in the proof of Lemma 6.1.

Lemma 6.2. *Let S_2 be a statement sequence and S_1 where there are updates of "specializing new configuration variables" w.r.t a mapping of new configuration variables ρ, $S_2 \approx_\rho^S S_1$. Then the output deciding variables in S_1 are a subset of the union of those in S_2, $OVar(S_1) \subseteq OVar(S_2)$.*

Proof. By induction on the sum of the program size of S_1 and S_2. □

Lemma 6.3. *Let $S_1 = while_{\langle n_1 \rangle}(e)\, \{S_1'\}$ and $S_2 = while_{\langle n_2 \rangle}(e)\, \{S_2'\}$ be two loop statements where all of the following hold:*

- *S_2' includes updates of "specializing new configuration variables" compared to S_1', $S_2' \approx_\rho^S S_1'$ where $Dom(\rho) \cap Def(S_2') = \emptyset$.*
- *the output deciding variables in S_1 are a subset of those in S_2, $OVar(S_1) \subseteq OVar(S_2)$;*
- *When started in states agreeing on values of output deciding variables in S_1 and S_2 including the input sequence variable and the I/O sequence variable, $\forall x \in OVar(S_1) \cup OVar(S_2) \cup \{id_I, id_{IO}\}\, \forall m_1'(\sigma_1')\, m_2'(\sigma_2') : (\sigma_1'(x) = \sigma_2'(x)), S_1'$ and S_2' terminate in the same way, produce the same output sequence, and have equivalent computation of defined variables in S_1' and S_2' as well as the input sequence variable and the I/O sequence variable $((S_1', m_1) \equiv_H (S_2', m_2)) \wedge ((S_1', m_1) \equiv_O (S_2', m_2)) \wedge (\forall x \in OVar(S_1) \cup OVar(S_2) \cup \{id_I, id_{IO}\} : (S_1', m_1) \equiv_x (S_2', m_2))$;*

If S_1 and S_2 start in states $m_1(loop_c^1, \sigma_1), m_2(loop_c^2, \sigma_2)$ respectively, with loop counters of S_1 and S_2 not initialized (S_1, S_2 have not executed yet), value stores agree on values of output deciding variables in S_1 and S_2, then, for any positive integer i, one of the following holds:

1. *Loop counters for S_1 and S_2 are always less than i if any is present, $\forall m_1'(loop_c^{1'})\, m_2'(loop_c^{2'}) : (S_1, m_1(loop_c^1, \sigma_1)) \xrightarrow{*} (S_1'', m_1'(loop_c^{1'})), loop_c^{1'}(n_1) < i, (S_2, m_2(loop_c^2, \sigma_2)) \xrightarrow{*} (S_2'', m_2'(loop_c^{2'})), loop_c^{2'}(n_2) < i$, S_1 and S_2 terminate in the same way, produce the same output sequence, and have equivalent computation of output deciding variables in both S_1 and S_2 and the input sequence variable, the I/O sequence variable, $(S_1, m_1) \equiv_H (S_2, m_2)$ and $(S_1, m_1) \equiv_O (S_2, m_2)$ and $\forall x \in (OVar(S_1) \cap OVar(S_2)) \cup \{id_I, id_{IO}\} : (S_1, m_1) \equiv_x (S_2, m_2)$;*
2. *The loop counter of S_1 and S_2 are of value less than or equal to i, and there are no reachable configurations $(S_1, m_1(loop_c^{1^i}, \sigma_{1_i}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(loop_c^{2^i}, \sigma_{2_i}))$ from $(S_2, m_2(\sigma_2))$ where all of the following hold:*

- *The loop counters of S_1 and S_2 are of value i, $loop_c^{1_i}(n_1) = loop_c^{2_i}(n_2) = i$.*
- *Value stores σ_{1_i} and σ_{2_i} agree on values of output deciding variables in both S_1 and S_2 as well as the input sequence variable and the I/O sequence variable, $\forall x \in (OVar(S_1) \cap OVar(S_2)) \cup \{id_I, id_{IO}\} : \sigma_{1_i}(x) = \sigma_{2_i}(x)$.*

3. *There are reachable configurations $(S_1, m_1(loop_c^{1_i}, \sigma_{1_i}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(loop_c^{2_i}, \sigma_{2_i}))$ from $(S_2, m_2(\sigma_2))$ where all of the following hold:*
   - *The loop counter of S_1 and S_2 are of value i, $loop_c^{1_i}(n_1) = loop_c^{2_i}(n_2) = i$.*
   - *Value stores σ_{1_i} and σ_{2_i} agree on values of output deciding variables in both S_1 and S_2 including the input sequence variable and the I/O sequence variable, $\forall x \in (OVar(S_1) \cap OVar(S_2)) \cup \{id_I, id_{IO}\} : \sigma_{1_i}(x) = \sigma_{2_i}(x)$.*

Proof. By induction on i.
Base case.

We show that, when $i = 1$, one of the following holds:

1. Loop counters for S_1 and S_2 are always less than 1 if any is present, $\forall m_1'(\text{loop}_c^{1'})\, m_2'(\text{loop}_c^{2'}) : (S_1, m_1(\text{loop}_c^1, \sigma_1)) \xrightarrow{*} (S_1'', m_1'(\text{loop}_c^{1'})), \text{loop}_c^{1'}(n_1) < i, (S_2, m_2(\text{loop}_c^2, \sigma_2)) \xrightarrow{*} (S_2'', m_2'(\text{loop}_c^{2'})), \text{loop}_c^{2'}(n_2) < i$, S_1 and S_2 terminate in the same way, produce the same output sequence, and have equivalent computation of used/defined variables in both S_1 and S_2 and the input sequence variable, the I/O sequence variable, $(S_1, m_1) \equiv_H (S_2, m_2)$ and $(S_1, m_1) \equiv_O (S_2, m_2)$ and $\forall x \in (\text{OVar}(S_1) \cap \text{OVar}(S_2)) \cup \{id_I, id_{IO}\} : (S_1, m_1) \equiv_x (S_2, m_2)$;
2. Loop counters of S_1 and S_2 are of value less than or equal to 1 but there are no reachable configurations $(S_1, m_1(\text{loop}_c^{1_1}, \sigma_{1_i}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(\text{loop}_c^{2_1}, \sigma_{2_i}))$ from $(S_2, m_2(\sigma_2))$ where all of the following hold:
   - The loop counter of S_1 and S_2 are of value 1, $\text{loop}_c^{1_1}(n_1) = \text{loop}_c^{2_1}(n_2) = 1$.
   - Value stores σ_{1_1} and σ_{2_1} agree on values of used variables in both S_1 and S_2 as well as the input sequence variable and the I/O sequence variable, $\forall x \in (\text{OVar}(S_1) \cap \text{OVar}(S_2)) \cup \{id_I, id_{IO}\} : \sigma_{1_1}(x) = \sigma_{2_1}(x)$.
3. There are reachable configuration $(S_1, m_1(\text{loop}_c^{1_1}, \sigma_{1_i}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(\text{loop}_c^{2_1}, \sigma_{2_i}))$ from $(S_2, m_2(\sigma_2))$ where all of the following hold:
   - The loop counter of S_1 and S_2 are of value 1, $\text{loop}_c^{1_1}(n_1) = \text{loop}_c^{2_1}(n_2) = 1$.
   - Value stores σ_{1_1} and σ_{2_1} agree on values of used variables in both S_1 and S_2 as well as the input sequence variable and the I/O sequence variable, $\forall x \in (\text{OVar}(S_1) \cap \text{OVar}(S_2)) \cup \{id_I, id_{IO}\} : \sigma_{1_1}(x) = \sigma_{2_1}(x)$.

By definition, variables used in the predicate expression e of S_1 and S_2 are used in S_1 and S_2, $\text{Use}(e) \subseteq \text{OVar}(S_1) \cap \text{OVar}(S_2)$. By assumption, value stores σ_1 and σ_2 agree on values of variables in $\text{Use}(e)$, the predicate expression e evaluates to the same value w.r.t value stores σ_1 and σ_2. There are three possibilities.

1. The evaluation of e crashes,
   $\mathcal{E}'[\![e]\!]\sigma_1 = \mathcal{E}'[\![e]\!]\sigma_2 = (\text{error}, v_{\mathfrak{of}})$.
   The execution of S_1 continues as follows.

   $(\text{while}_{\langle n_1\rangle}(e)\,\{S_1'\}, m_1(\sigma_1))$
   $\rightarrow(\text{while}_{\langle n_1\rangle}((\text{error}, v_{\mathfrak{of}}))\,\{S_1'\}, m_1(\sigma_1))$
   by the rule EEval'
   $\rightarrow(\text{while}_{\langle n_1\rangle}(0)\,\{S_1'\}, m_1(1/\mathfrak{f}))$
   by the ECrash rule
   $\xrightarrow{i}(\text{while}_{\langle n_1\rangle}(0)\,\{S_1'\}, m_1(1/\mathfrak{f}))$ for any $i > 0$
   by the Crash rule.

   Similarly, the execution of S_2 started from the state $m_2(\sigma_2)$ crashes. Therefore S_1 and S_2 terminate in the same way when started from m_1 and m_2 respectively. Because $\sigma_1(id_{IO}) = \sigma_2(id_{IO})$, the lemma holds.
2. The evaluation of e reduces to zero, $\mathcal{E}'[\![e]\!]\sigma_1 = \mathcal{E}'[\![e]\!]\sigma_2 = (0, v_{\mathfrak{of}})$.
   The execution of S_1 continues as follows.

   $(\text{while}_{\langle n_1\rangle}(e)\,\{S_1'\}, m_1(\sigma_1))$
   $= (\text{while}_{\langle n_1\rangle}((0, v_{\mathfrak{of}}))\,\{S_1'\}, m_1(\sigma_1))$
   by the rule EEval'
   $\rightarrow(\text{while}_{\langle n_1\rangle}(0)\,\{S_1'\}, m_1(\sigma_1))$
   by the E-Oflow1 or E-Oflow2 rule
   $\rightarrow(\text{skip}, m_1(\sigma_1))$ by the Wh-F rule.

   Similarly, the execution of S_2 gets to the configuration $(\text{skip}, m_2(\sigma_2))$. Loop counters of S_1 and S_2 are less than 1 and value stores agree on values of used/defined variables in both S_1 and S_2 as well as the input sequence variable and the I/O sequence variable.
3. The evaluation of e reduces to the same nonzero integer value, $\mathcal{E}'[\![e]\!]\sigma_1 = \mathcal{E}'[\![e]\!]\sigma_2 = (0, v_{\mathfrak{of}})$.
   Then the execution of S_1 proceeds as follows:

   $(\text{while}_{\langle n_1\rangle}(e)\,\{S_1'\}, m_1(\sigma_1))$
   $= (\text{while}_{\langle n_1\rangle}((v, v_{\mathfrak{of}}))\,\{S_1'\}, m_1(\sigma_1))$
   by the rule EEval'
   $\rightarrow(\text{while}_{\langle n_1\rangle}(v)\,\{S_1'\}, m_1(\sigma_1))$
   by the E-Oflow1 or E-Oflow2 rule
   $\rightarrow(S_1'; \text{while}_{\langle n_1\rangle}(e)\,\{S_1'\}, m_1($
   $\text{loop}_c^1 \cup \{(n_1) \mapsto 1\}, \sigma_1))$ by the Wh-T rule.

   Similarly, the execution of S_2 proceeds to the configuration $(S_2'; \text{while}_{\langle n_2\rangle}(e)\,\{S_2'\}, m_2(\text{loop}_c^2 \cup \{n_2 \mapsto 1\}, \sigma_2))$.
   By the hypothesis IH, we show that S_1' and S_2' terminate in the same way and produce the same output sequence when started in the state $m_1(\text{loop}_c^{1_1}, \sigma_1)$ and $m_2(\text{loop}_c^{2_1}, \sigma_2)$, and S_1' and S_2' have equivalent computation of variables used or defined in both statement sequences if both terminate. We need to show that all conditions are satisfied for the application of the hypothesis IH.
   - variables in the domain of ρ are not redefined in the execution of S_2'.
     The above three conditions are by assumption.
     By definition, $\text{size}(S_1) = 1 + \text{size}(S_1')$. Then, $\text{size}(S_1') + \text{size}(S_2') = k + 1 - 2 = k - 1$.
   - Value stores σ_1 and σ_2 agree on values of used variables in S_1' and S_2' as well as the input, I/O sequence variable.
     By definition, $\text{OVar}(S_1') \subseteq \text{OVar}(S_1)$. So are the cases to S_2' and S_2. In addition, value stores σ_1 and σ_2 are not changed in the evaluation of the predicate expression e. The condition holds.
   - Values of new configuration variables are consistent in the value store σ_2 and the specialization ρ, $\forall id \in \text{Dom}(\rho) : \sigma_2(id) = \rho(id)$.
     By assumption.

   By assumption, S_1' and S_2' terminate in the same way and produce the same output sequence when started in states $m_1(\text{loop}_c', \sigma_1)$ and $m_2(\text{loop}_c', \sigma_2)$. In addition, S_1' and S_2' have equivalent computation of variables used or defined in S_1' and S_2' when started in states $m_1(\text{loop}_c', \sigma_1)$ and $m_2(\text{loop}_c', \sigma_2)$.
   Then there are two cases.

(a) S_1' and S_2' both do not terminate and produce the same output sequence.
By Lemma E.2, $S_1'; S_1$ and $S_2'; S_2$ both do not terminate and produce the same output sequence.

(b) S_1' and S_2' both terminate and have equivalent computation of variables used or defined in S_1' and S_2'.
By assumption, $(S_1', m_1(\text{loop}_c', \sigma_1)) \xrightarrow{*} (\text{skip}, m_1'(\text{loop}_c'', \sigma_1'))$; $(S_2', m_2(\text{loop}_c', \sigma_2)) \xrightarrow{*} (\text{skip}, m_2'(\text{loop}_c'', \sigma_2'))$ where $\forall x \in (\text{OVar}(S_1') \cap \text{OVar}(S_2')) \cup \{id_I, id_{IO}\}, \sigma_1'(x) = \sigma_2'(x)$.
Because S_1 and S_2 have the same predicate expression, variables used in the predicate expression of S_1 and S_2 are not in the domain of ρ. By assumption, $\text{OVar}(S_1') \subseteq \text{OVar}(S_2') \subseteq \text{OVar}(S_2') \cup \text{Dom}(\rho)$ and $\text{OVar}(S_1') \subseteq \text{OVar}(S_2')$. Then variables used in the predicate expression of S_1 and S_2 are either in variables used or defined in both S_1' and S_2' or not. Therefore value stores σ_2' and σ_1' agree on values of variables used in the expression e and even variables used or defined in S_1 and S_2.

Induction step on iterations

The induction hypothesis (IH) is that, when $i \geq 1$, one of the following holds:

1. Loop counters for S_1 and S_2 are always less than i if any is present, $\forall m_1'(\text{loop}_c^{1'})\, m_2'(\text{loop}_c^{2'}) : (S_1, m_1(\text{loop}_c^1, \sigma_1)) \xrightarrow{*} (S_1'', m_1'(\text{loop}_c^{1'})), \text{loop}_c^{1'}(n_1) < i, (S_2, m_2(\text{loop}_c^2, \sigma_2)) \xrightarrow{*} (S_2'', m_2'(\text{loop}_c^{2'})), \text{loop}_c^{2'}(n_2) < i$, S_1 and S_2 terminate in the same way, produce the same output sequence, and have equivalent computation of used/defined variables in both S_1 and S_2 and the input sequence variable, the I/O sequence variable, $(S_1, m_1) \equiv_H (S_2, m_2)$ and $(S_1, m_1) \equiv_O (S_2, m_2)$ and $\forall x \in (\text{OVar}(S_1) \cap \text{OVar}(S_2)) \cup \{id_I, id_{IO}\} : (S_1, m_1) \equiv_x (S_2, m_2)$;
2. The loop counter of S_1 and S_2 are of value less than or equal to i, and there are no reachable configurations $(S_1, m_1(\text{loop}_c^{1_i}, \sigma_{1_i}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(\text{loop}_c^{2_i}, \sigma_{2_i}))$ from $(S_2,\ m_2(\sigma_2))$ where all of the following hold:
   - The loop counters of S_1 and S_2 are of value i, $\text{loop}_c^{1_i}(n_1) = \text{loop}_c^{2_i}(n_2) = i$.
   - Value stores σ_{1_i} and σ_{2_i} agree on values of used variables in both S_1 and S_2 as well as the input sequence variable and the I/O sequence variable, $\forall x \in (\text{OVar}(S_1) \cap \text{OVar}(S_2)) \cup \{id_I, id_{IO}\} : \sigma_{1_i}(x) = \sigma_{2_i}(x)$.
3. There are reachable configurations $(S_1, m_1(\text{loop}_c^{1_i}, \sigma_{1_i}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(\text{loop}_c^{2_i}, \sigma_{2_i}))$ from $(S_2,\ m_2(\sigma_2))$ where all of the following hold:
   - The loop counter of S_1 and S_2 are of value i, $\text{loop}_c^{1_i}(n_1) = \text{loop}_c^{2_i}(n_2) = i$.
   - Value stores σ_{1_i} and σ_{2_i} agree on values of used variables in both S_1 and S_2 as well as the input sequence variable and the I/O sequence variable, $\forall x \in (\text{OVar}(S_1) \cap \text{OVar}(S_2)) \cup \{id_I, id_{IO}\} : \sigma_{1_i}(x) = \sigma_{2_i}(x)$.

Then we show that, when $i+1$, one of the following holds: The induction hypothesis (IH) is that, when $i \geq 1$, one of the following holds:

1. Loop counters for S_1 and S_2 are always less than $i+1$ if any is present, $\forall m_1'(\text{loop}_c^{1'})\, m_2'(\text{loop}_c^{2'}) : (S_1, m_1(\text{loop}_c^1, \sigma_1)) \xrightarrow{*} (S_1'', m_1'(\text{loop}_c^{1'})), \text{loop}_c^{1'}(n_1) < i+1, (S_2, m_2(\text{loop}_c^2, \sigma_2)) \xrightarrow{*} (S_2'', m_2'(\text{loop}_c^{2'})), \text{loop}_c^{2'}(n_2) < i+1$, S_1 and S_2 terminate in the same way, produce the same output sequence, and have equivalent computation of used/defined variables in both S_1 and S_2 and the input sequence variable, the I/O sequence variable, $(S_1, m_1) \equiv_H (S_2, m_2)$ and $(S_1, m_1) \equiv_O (S_2, m_2)$ and $\forall x \in (\text{OVar}(S_1) \cap \text{OVar}(S_2)) \cup \{id_I, id_{IO}\} : (S_1, m_1) \equiv_x (S_2, m_2)$;
2. The loop counter of S_1 and S_2 are of value less than or equal to $i+1$, and there are no reachable configurations $(S_1, m_1(\text{loop}_c^{1_{i+1}}, \sigma_{1_{i+1}}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(\text{loop}_c^{2_{i+1}}, \sigma_{2_{i+1}}))$ from $(S_2,\ m_2(\sigma_2))$ where all of the following hold:
   - The loop counters of S_1 and S_2 are of value $i+1$, $\text{loop}_c^{1_{i+1}}(n_1) = \text{loop}_c^{2_{i+1}}(n_2) = i+1$.
   - Value stores $\sigma_{1_{i+1}}$ and $\sigma_{2_{i+1}}$ agree on values of used variables in both S_1 and S_2 as well as the input sequence variable and the I/O sequence variable, $\forall x \in (\text{OVar}(S_1) \cap \text{OVar}(S_2)) \cup \{id_I, id_{IO}\} : \sigma_{1_{i+1}}(x) = \sigma_{2_{i+1}}(x)$.
3. There are reachable configurations $(S_1, m_1(\text{loop}_c^{1_{i+1}}, \sigma_{1_i}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(\text{loop}_c^{2_{i+1}}, \sigma_{2_i}))$ from $(S_2,\ m_2(\sigma_2))$ where all of the following hold:
   - The loop counter of S_1 and S_2 are of value i, $\text{loop}_c^{1_{i+1}}(n_1) = \text{loop}_c^{2_{i+1}}(n_2) = i+1$.
   - Value stores $\sigma_{1_{i+1}}$ and $\sigma_{2_{i+1}}$ agree on values of used variables in both S_1 and S_2 as well as the input sequence variable and the I/O sequence variable, $\forall x \in (\text{OVar}(S_1) \cap \text{OVar}(S_2)) \cup \{id_I, id_{IO}\} : \sigma_{1_{i+1}}(x) = \sigma_{2_{i+1}}(x)$.

By hypothesis IH, there is no configuration where loop counters of S_1 and S_2 are of value $i+1$ when any of the following holds:

1. Loop counters for S_1 and S_2 are always less than i if any is present, $\forall m_1'(\text{loop}_c^{1'})\, m_2'(\text{loop}_c^{2'}) : (S_1, m_1(\text{loop}_c^1, \sigma_1)) \xrightarrow{*} (S_1'', m_1'(\text{loop}_c^{1'})), \text{loop}_c^{1'}(n_1) < i, (S_2, m_2(\text{loop}_c^2, \sigma_2)) \xrightarrow{*} (S_2'', m_2'(\text{loop}_c^{2'})), \text{loop}_c^{2'}(n_2) < i$, S_1 and S_2 terminate in the same way, produce the same output sequence, and have equivalent computation of used/defined variables in both S_1 and S_2 and the input sequence variable, the I/O sequence variable, $(S_1, m_1) \equiv_H (S_2, m_2)$ and $(S_1, m_1) \equiv_O (S_2, m_2)$ and $\forall x \in (\text{OVar}(S_1) \cap \text{OVar}(S_2)) \cup \{id_I, id_{IO}\} : (S_1, m_1) \equiv_x (S_2, m_2)$;
2. The loop counter of S_1 and S_2 are of value less than or equal to i, and there are no reachable configurations $(S_1, m_1(\text{loop}_c^{1_i}, \sigma_{1_i}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(\text{loop}_c^{2_i}, \sigma_{2_i}))$ from $(S_2,\ m_2(\sigma_2))$ where all of the following hold:
   - The loop counters of S_1 and S_2 are of value i, $\text{loop}_c^{1_i}(n_1) = \text{loop}_c^{2_i}(n_2) = i$.
   - Value stores σ_{1_i} and σ_{2_i} agree on values of used variables in both S_1 and S_2 as well as the input sequence variable, and the I/O sequence variable, $\forall x \in (\text{OVar}(S_1) \cap \text{OVar}(S_2)) \cup \{id_I, id_{IO}\} : \sigma_{1_i}(x) = \sigma_{2_i}(x)$.

When there are reachable configurations $(S_1, m_1(\text{loop}_c^{1_i}, \sigma_{1_i}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(\text{loop}_c^{2_i}, \sigma_{2_i}))$ from $(S_2,\ m_2(\sigma_2))$ where all of the following hold:

- The loop counter of S_1 and S_2 are of value i, $\text{loop}_c^{1_i}(n_1) = \text{loop}_c^{2_i}(n_2) = i$.
- The loop counter of S_1 and S_2 are of value i, $\text{loop}_c^{1_i}(n_1) = \text{loop}_c^{2_i}(n_2) = i$.
- Value stores σ_{1_i} and σ_{2_i} agree on values of used variables in both S_1 and S_2 as well as the input sequence variable and the I/O sequence variable, $\forall x \in (\text{OVar}(S_1) \cap \text{OVar}(S_2)) \cup \{id_I, id_{IO}\} : \sigma_{1_i}(x) = \sigma_{2_i}(x)$.

By similar argument in base case, we have one of the following holds:

1. Loop counters for S_1 and S_2 are always less than $i+1$ if any is present, $\forall m_1'(\text{loop}_c^{1'})\, m_2'(\text{loop}_c^{2'}) : (S_1, m_1(\text{loop}_c^1, \sigma_1)) \xrightarrow{*}$

```
1:  enum id {o_1}                1':  enum id {o_1, o_2}
2:  a : enum id                  2':  a : enum id
3:  If (a == o_1) then           3':  If (a == o_1) then
4:      output 2 + c             4':      output 2 + c
5:                               5':  If (a == o_2) then
6:                               6':      output 3 + c

    old                              new
```

Figure 16: Enumeration type extension

$(S_1'', m_1'(\text{loop}_c^{1'})), \text{loop}_c^{1'}(n_1) < i+1, (S_2, m_2(\text{loop}_c^2, \sigma_2)) \xrightarrow{*} (S_2'', m_2'(\text{loop}_c^{2'})), \text{loop}_c^{2'}(n_2) < i+1$, S_1 and S_2 terminate in the same way, produce the same output sequence, and have equivalent computation of used/defined variables in both S_1 and S_2 and the input sequence variable, the I/O sequence variable, $(S_1, m_1) \equiv_H (S_2, m_2)$ and $(S_1, m_1) \equiv_O (S_2, m_2)$ and $\forall x \in (\text{OVar}(S_1) \cap \text{OVar}(S_2)) \cup \{id_I, id_{IO}\} : (S_1, m_1) \equiv_x (S_2, m_2)$;

2. The loop counter of S_1 and S_2 are of value less than or equal to $i+1$, and there are no reachable configurations $(S_1, m_1(\text{loop}_c^{1_{i+1}}, \sigma_{1_{i+1}}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(\text{loop}_c^{2_{i+1}}, \sigma_{2_{i+1}}))$ from $(S_2, m_2(\sigma_2))$ where all of the following hold:
   - The loop counters of S_1 and S_2 are of value i, $\text{loop}_c^{1_{i+1}}(n_1) = \text{loop}_c^{2_{i+1}}(n_2) = i$.
   - Value stores $\sigma_{1_{i+1}}$ and $\sigma_{2_{i+1}}$ agree on values of used variables in both S_1 and S_2 as well as the input sequence variable and the I/O sequence variable, $\forall x \in (\text{OVar}(S_1) \cap \text{OVar}(S_2)) \cup \{id_{I+1}, id_{IO}\} : \sigma_{1_{i+1}}(x) = \sigma_{2_{i+1}}(x)$.
3. There are reachable configurations $(S_1, m_1(\text{loop}_c^{1_{i+1}}, \sigma_{1_{i+1}}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(\text{loop}_c^{2_{i+1}}, \sigma_{2_{i+1}}))$ from $(S_2, m_2(\sigma_2))$ where all of the following hold:
   - The loop counter of S_1 and S_2 are of value i, $\text{loop}_c^{1_{i+1}}(n_1) = \text{loop}_c^{2_{i+1}}(n_2) = i+1$.
   - Value stores $\sigma_{1_{i+1}}$ and $\sigma_{2_{i+1}}$ agree on values of used variables in both S_1 and S_2 as well as the input sequence variable and the I/O sequence variable, $\forall x \in (\text{OVar}(S_1) \cap \text{OVar}(S_2)) \cup \{id_{I+1}, id_{IO}\} : \sigma_{1_{i+1}}(x) = \sigma_{2_{i+1}}(x)$.

□

6.2 Proof rule for enumeration type extension

Enumeration types allow developers to list similar items. New code is usually accompanied with the introduction of new enumeration labels. Figure 16 shows an example of the update. The new enum label o_2 gives a new option for matching the value of the variable a, which introduce the new code $b := 3 + c$. To show updates "enumeration type extension" to be backward compatible, we assume that values of enum variables, used in the If-predicate introducing the new code, are only from inputs that cannot be translated to new enum labels.

In order to have a general definition of the update class, we show a relation between two sequences of enumeration type definitions, called proper subset.

Definition 26. (Extension relation of enumeration types) *Let EN_1, EN_2 be two different sequences of enumeration type definitions. EN_1 is a subset of EN_2, written $EN_1 \subset EN_2$, iff one of the following holds:*

1. *EN_1 = "enum $id\,\{el_1\}$", EN_2 = "enum $id\,\{el_2\}$" where labels in type "enum id" in EN_1 are a subset of those in EN_2, $el_2 = el_1, el$ and $el \neq \varnothing$;*
2. *EN_1, EN_2 include more than one enumeration type definitions EN_1 = "enum $id\,\{el_1\}, EN_1'$", EN_2 = "enum $id\,\{el_2\}, EN_2'$" where one of the following holds:*
   (a) $(EN_1' \subset EN_2')$ and $(el_1 = el_2) \vee (el_2 = el_1, el)$;
   (b) $(EN_1' \subset EN_2') \vee (EN_1' = EN_2')$ and "enum $id\,\{el_1\}$" $\subset$ "enum $id\,\{el_2\}$".

Definition 27. (Enumeration type extension) *Let P_1, P_2 be two programs where enumeration type definitions EN_1 in P_1 are a subset of EN_2 in P_2, $EN_1 \subset EN_2$ and E are new enum labels in P_2. A statement sequence S_2 in a program P_2 includes updates of enumeration type extension compared with a statement sequence S_1 in P_1, written $S_2 \approx_E^S S_1$, iff one of the following holds:*

1. *S_2 = "If(id==l) then$\{S_2^t\}$ else$\{S_2^f\}$" and all of the following hold:*
   - *$l \in E$;*
   - *The variable id is not lvalue in an assignment statement, "$id := e$" $\notin P_2$;*
   - *$S_2^f \approx_E^S S_1$;*
2. *S_1 = "If(e) then$\{S_1^t\}$ else$\{S_1^f\}$", S_2 = "If(e) then$\{S_2^t\}$ else$\{S_2^f\}$" where $(S_2^t \approx_E^S S_1^t) \wedge (S_2^f \approx_E^S S_1^f)$;*
3. *S_1 = "while$_{\langle n_1 \rangle}(e)\,\{S_1'\}$", S_2 = "while$_{\langle n_2 \rangle}(e)\,\{S_2'\}$" where $S_2' \approx_E^S S_1'$;*
4. *$S_1 \approx_O^S S_2$;*
5. *$S_1 = S_1'; s_1$ and $S_2 = S_2'; s_2$ where $(S_2' \approx_E^S S_1') \wedge (S_2' \approx_H^S S_1') \wedge \big(\forall x \in Imp(s_1, id_{IO}) \cup Imp(s_1, id_{IO}) : (S_2' \approx_x^S S_1')\big) \wedge (s_2 \approx_E^S s_1)$.*

We show that two programs terminate in the same way, produce the same output sequence, and have equivalent computation of variables defined in both of them in executions if there are updates of enumeration type extension between them.

Lemma 6.4. *Let S_1 and S_2 be two statement sequences in programs P_1 and P_2 respectively where there are updates of enumeration type extensions in S_2 of P_2 compared with S_1 of P_1, $S_2 \approx_E^S S_1$. If S_1 and S_2 start in states $m_1(\sigma_1)$ and $m_2(\sigma_2)$ such that both of the following hold:*

- *Value stores σ_1 and σ_2 agree on values of output deciding variables in both S_1 and S_2 including the input sequence variable and the I/O sequence variable, $\forall x \in (OVar(S_1) \cup OVar(S_2)) \cup \{id_I, id_{IO}\} : \sigma_1(x) = \sigma_2(x)$;*
- *No variables used in S_2 are of initial value of enum labels in E, $\forall x \in Use(S_2) : (\sigma_2(x) \notin E)$;*
- *No inputs are translated to any label in E during the execution of S_2;*

then S_1 and S_2 terminate in the same way, produce the same output sequence, and when S_1 and S_2 both terminate, they have equivalent computation of used variables and defined variables,

- *$(S_1, m_1) \equiv_H (S_2, m_2)$;*
- *$(S_1, m_1) \equiv_O (S_2, m_2)$;*
- *$\forall x \in OVar(S_1) \cup OVar(S_2) : (S_1, m_1) \equiv_x (S_2, m_2)$;*

Proof. By induction on the sum of the program size of S_1 and S_2, $\text{size}(S_1) + \text{size}(S_1)$.
Base case. S_1 is a simple statement s, and S_2 = "If(id==l) then$\{s_2^t\}$ else$\{s_2^f\}$" where all of the following hold:

- $l \in E$;
- s_2^t, s_2^f are two simple statements;
- $s_2^f = s$;

We informally argue that the value of the variable id in the predicate expression of S_2 only coming from an input value or the initial value. There are three ways a scalar variable is defined: the execution of an assignment statement, the execution of an input statement or the initial value. Because id is not lvalue in an assignment statement, then the value of id is only from the execution of an input statement or the initial value.

In addition, by assumption, any output deciding variable is not of the initial value of enum label in E; no input values are translated into an enum label in E. Then the execution of S_2 proceeds as follows:

$(\text{If}(id{=}{=}l)\ \text{then}\{s_2^t\}\ \text{else}\{s_2^f\}, m_2(\sigma_2))$
$\rightarrow(\text{If}(0)\ \text{then}\{s_2^t\}\ \text{else}\{s_2^f\}, m_2(\sigma_2))$
by the rule Eq-F
$\rightarrow(s_2^f, m_2(\sigma_2))$ by the If-F rule.

The value store σ_2 is not updated in the execution of S_2 so far. By assumption, value stores σ_1 and σ_2 agree on values of output deciding variables in both S_1 and S_2.

By Theorem 2 and 4, S_1 and S_2 terminate in the same way, produce the same I/O sequence. The lemma holds.
Induction step.

The hypothesis is that this lemma holds when the sum k of the program size of S_1 and S_2 are great than or equal to 4, $k \geq 4$.

We then show that this lemma holds when the sum of the program size of S_1 and S_2 is $k+1$. There are cases regarding $S_2 \approx_E^S S_1$.

1. S_1 and S_2 are both "If" statement:
   $S_1 =$ "If(e) then$\{S_1^t\}$ else$\{S_1^f\}$", $S_2 =$ "If(e) then$\{S_2^t\}$ else$\{S_2^f\}$" where both of the following hold
   - $S_2^t \approx_E^S S_1^t$;
   - $S_2^f \approx_E^S S_1^f$;

   By the definition of $\text{Imp}_o(S_1)$, variables used in the predicate expression e are a subset of output deciding variables in S_1 and S_2, Use$(e) \subseteq$ OVar$(S_1) \cap$ OVar(S_2). By assumption, corresponding variables used in e are of same value in value stores σ_1 and σ_2. By Lemma D.1, the expression evaluates to the same value w.r.t value stores σ_1 and σ_2. There are three possibilities.
   (a) The evaluation of e crashes, $\mathcal{E}'[\![e]\!]\sigma_1 = \mathcal{E}'[\![e]\!]\sigma_2 = (\text{error}, v_{\mathfrak{of}})$.
   The execution of S_1 continues as follows:
   $(\text{If}(e)\ \text{then}\{S_1^t\}\ \text{else}\{S_1^f\}, m_1(\sigma_1))$
   $\rightarrow(\text{If}((\text{error}, v_{\mathfrak{of}}))\ \text{then}\{S_1^t\}\ \text{else}\{S_1^f\}, m_1(\sigma_1))$
   by the rule EEval'
   $\rightarrow(\text{If}(0)\ \text{then}\{S_1^t\}\ \text{else}\{S_1^f\}, m_1(1/\mathfrak{f}))$
   by the ECrash rule
   $\xrightarrow{i}(\text{If}(0)\ \text{then}\{S_1^t\}\ \text{else}\{S_1^f\}, m_1(1/\mathfrak{f}))$ for any $i > 0$
   by the Crash rule.
   Similarly, the execution of S_2 started from the state $m_2(\sigma_2)$ crashes. The lemma holds.
   (b) The evaluation of e reduces to zero, $\mathcal{E}'[\![e]\!]\sigma_1 = \mathcal{E}'[\![e]\!]\sigma_2 = (0, v_{\mathfrak{of}})$.
   The execution of S_1 continues as follows.
   $(\text{If}(e)\ \text{then}\{S_1^t\}\ \text{else}\{S_1^f\}, m_1(\sigma_1))$
   $= (\text{If}((0, v_{\mathfrak{of}}))\ \text{then}\{S_1^t\}\ \text{else}\{S_1^f\}, m_1(\sigma_1))$
   by the rule EEval'
   $\rightarrow(\text{If}(0)\ \text{then}\{S_1^t\}\ \text{else}\{S_1^f\}, m_1(\sigma_1))$
   by the E-Oflow1 or E-Oflow2 rule
   $\rightarrow(S_1^f, m_1(\sigma_1))$ by the If-F rule.
   Similarly, the execution of S_2 gets to the configuration $(S_2^f, m_2(\sigma_2))$.
   By the hypothesis IH, we show the lemma holds. We need to show that all conditions are satisfied for the application of the hypothesis IH.
   - $S_2^f \approx_E^S S_1^f$
     By assumption.
   - The sum of the program size of S_1^f and S_2^f is less than k, size(S_1^f) + size$(S_2^f) < k$.
     By definition, size$(S_1) = 1+$size$(S_1^t)+$size(S_1^f). Then, size(S_1^f) + size$(S_2^f) < k + 1 - 2 = k - 1$.
   - Value stores σ_1 and σ_2 agree on values of output deciding variables in S_1^f and S_2^f including the input, I/O sequence variable.
     By definition, OVar$(S_1^f) \subseteq$ OVar(S_1). So are the cases to S_2^f and S_2. In addition, value stores σ_1 and σ_2 are not changed in the evaluation of the predicate expression e. The condition holds.
   - There are no inputs translated to enum labels in E in S_2^f's execution.
     By assumption.

   By the hypothesis IH, the lemma holds.
   (c) The evaluation of e reduces to the same nonzero integer value, $\mathcal{E}'[\![e]\!]\sigma_1 = \mathcal{E}'[\![e]\!]\sigma_2 = (0, v_{\mathfrak{of}})$.
   By similar to the second subcase above.
2. S_1 and S_2 are both "while" statements:
   $S_1 =$ "while$_{\langle n_1 \rangle}(e)\ \{S_1'\}$", $S_2 =$ "while$_{\langle n_2 \rangle}(e)\ \{S_2'\}$" where $S_2' \approx_E^S S_1'$;
   By Lemma 6.6, we show the lemma holds. We need to show all the required conditions for the application of Lemma 6.6 holds
   (a) No variables are of initial values as new enum labels in E;
   (b) Value stores σ_1 and σ_2 agree on values of variables used in both S_1 and S_2;
   (c) Enumeration types in P_1 are a subset of those in P_2;
   The above three conditions are by assumption.
   (d) The output deciding variables in S_1' are a subset of those in S_2';
   The above condition is by Lemma 6.5.
   (e) S_1' and S_2' produce the same output sequence, terminate in the same way and have equivalent computation of defined variables in both S_1' and S_2' when started in states agreeing on values of variables used in both S_1' and S_2';
   Because size(S_1) = size$(S_1') + 1$, then the condition holds by the induction hypothesis.

   By Lemma 6.6, the lemma holds.
3. $S_2 =$ "If$(id{=}{=}l)$ then $\{S_2^t\}$ else $\{S_2^f\}$" such that both of the following hold:
   - The label l is in E, $l \in E$;
   - The variable id is not lvalue in an assignment statement, $\nexists$"$id :=$ e" in S_2;
   - There are updates of enumeration type extension from S_2^f to S_1, $S_2^f \approx_E^S S_1$;

   By Lemma 6.6, we show this lemma holds. We need to show all the conditions are satisfied for the application of Lemma 6.6.
   - S_1' and S_2' have same set of output deciding variables, OVar(S_1') = OVar(S_2') = OVar(S);
   - The output deciding variables in S_1' are a subset of those in S_2', OVar$(S_1') \subseteq$ OVar(S_2');
     By Lemma 6.5.
   - There are no inputs translated to enum labels in the set E.
     By assumption.
   - When started in states $m_1'(\sigma_1'), m_2'(\sigma_1')$ where value stores σ_1' and σ_2' agree on values of output deciding variables in both S_1' and S_2' as well as the input sequence variable and

the I/O sequence variable, and there are no inputs translated to enum labels in E, then S_1' and S_2' produce the same output sequence.
By the induction hypothesis IH. This is because the sum of the program size of S_1' and S_2' is less than k. By definition, size$(S_1) = 1 + \text{size}(S_1')$.

4. S_2 = "If(id==l) then $\{S_2^t\}$ else $\{S_2^f\}$" such that both of the following hold:
   - The label l is in E, $l \in E$;
   - The variable id is not lvalue in an assignment statement, "id := e" $\notin S_2$;
   - There are updates of enumeration type extension from S_2^f to S_1, $S_2^f \approx_E^S S_1$;

   We informally argue that the value of the variable id in the predicate expression of S_2 only coming from an input value or the initial value. There are several ways a scalar variable is defined: the execution of an assignment statement, the execution of an input statement or the initial value. Because id is not lvalue in an assignment statement, then the value of id is only from the execution of an input statement or initial value. In addition, by assumption, any used variable is not of initial value of enum label in E; no input values are translated into an enum label in E.
   Then the expression id==l evaluates to 0. The execution of S_2 proceeds as follows.

   (If(id==l) then$\{S_2^t\}$ else$\{S_2^f\}, m_2(\sigma_2)$)
   $\rightarrow$(If(v==l) then$\{S_2^t\}$ else$\{S_2^f\}, m_2(\sigma_2)$) where $v \neq l$
   by the rule Var
   $\rightarrow$(If(0) then$\{S_2^t\}$ else$\{S_2^f\}, m_1(\sigma_2)$)
   by the Eq-F rule
   $\rightarrow(S_2^f, m_2(\sigma_2))$ by the If-F rule.

   By the hypothesis IH, we show the lemma holds. We need to show the conditions are satisfied for the application of the hypothesis IH.
   - $S_2^f \approx_E^S S_1$
   - There are no inputs translated to enum labels in E in S_2^f's execution.
   - Initial values of used variables in S_2 are not enum labels in E.
     The above three conditions are by assumption.
   - The sum of the program size of S_1 and S_2^f is less than k, size(S_1) + size$(S_2^f) < k$.
     By definition, size$(S_1) = 1 + \text{size}(S_1^t) + \text{size}(S_1^f)$. Then, size$(S_1)$ + size$(S_2^f) < k + 1 - 1 - \text{size}(S_2^t) < k$.
   - Value stores σ_1 and σ_2 agree on values of used variables in both S_1 and S_2^f as well as the input, I/O sequence variable.
     By definition, OVar$(S_2^f) \subseteq$ OVar(S_2). In addition, value stores σ_1 and σ_2 are not changed in the evaluation of the predicate expression e. The condition holds.

   By the hypothesis IH, this lemma holds.
5. $S_1 = S_1'; s_1$ and $S_2 = S_2'; s_2$ such that both of the following hold:
   - $S_2' \approx_E^S S_1'$;
   - $s_2 \approx_E^S s_1$;

   By the hypothesis IH, we show S_2' and S_1' terminate in the same way, produce the same output sequence, and have equivalent computation of defined variables in S_1 and S_2. We need to show that all the conditions are satisfied for the application of the hypothesis IH.
   - $S_2' \approx_E^S S_1'$;
   - There are no inputs translated to enum labels in E in S_2''s execution.
   - Initial values of used variables in S_2' are not enum labels in E.
     The above three conditions are by assumption.
   - The sum of the program size of S_1' and S_2' is less than k, size(S_1') + size$(S_2') < k$.
     By definition, size(S_1) = size(S_1') + size(s_1). Then, size(S_1') + size$(S_2') < k + 1 - \text{size}(s_2) - \text{size}(s_1) < k$.
   - Value stores σ_1 and σ_2 agree on values of used variables in both S_1' and S_2' as well as the input, output, I/O sequence variable.
     By definition, OVar$(S_2') \subseteq$ OVar(S_2), OVar$(S_1') \subseteq$ OVar(S_1). In addition, value stores σ_1 and σ_2 are not changed in the evaluation of the predicate expression e. The condition holds.

   By the hypothesis IH, one of the following holds:
   (a) S_1' and S_2' both do not terminate.
   By Lemma E.2, executions of $S_1 = S_1'; s_1$ and $S_2 = S_2'; s_2$ both do not terminate and produce the same output sequence.
   (b) S_1' and S_2' both terminate.
   By assumption, $(S_2', m_2(\sigma_2)) \xrightarrow{*} (\text{skip}, m_2'(\sigma_2'))$, $(S_1', m_1(\sigma_1)) \xrightarrow{*} (\text{skip}, m_1'(\sigma_1'))$.
   By Corollary E.1, $(S_2'; s_2, m_2(\sigma_2)) \xrightarrow{*} (s_2, m_2'(\sigma_2'))$, $(S_1'; s_1, m_1(\sigma_1)) \xrightarrow{*} (s_1, m_1'(\sigma_1'))$.
   By the hypothesis IH, we show that s_2 and s_1 terminate in the same way, produce the same output sequence and when s_2 and s_1 both terminate, s_2 and s_1 have equivalent computation of variables used or defined in s_1 and s_2 and the input, output, and I/O sequence variables.
   We need to show that all conditions are satisfied for the application of the hypothesis IH.
   - There are updates of "enumeration type extension" between s_2 and s_1;
   - There are no input values translated into enum labels in E in the execution of s_2;
     The above two conditions are by assumption.
   - The sum of the program size s_2 and s_1 is less than or equals to k;
     By definition, size$(S_2') \geq 1$, size$(S_1') \geq 1$. Therefore, size(s_2) + size$(s_1) < k + 1 - \text{size}(S_2') - \text{size}(S_1') \leq k$.
   - Value stores σ_1' and σ_2' agree on values of used variables in s_2 and s_1 as well as the input, output, I/O sequence variable.
     By Lemma 6.5, OVar$(s_1) \subseteq$ OVar(s_2), then OVar(s_2) $\cap$ OVar(s_1) = OVar(s_1). Similarly, by Lemma 6.5, OVar$(S_1') \subseteq$ OVar(S_2'). For any variable id in OVar(s_1), if id is not in OVar(S_1'), then the value of id is not changed in the execution of S_1' and S_2', $\sigma_1'(id) = \sigma_1(id) = \sigma_2(id) = \sigma_2'(id)$. Otherwise, the variable id is defined in the execution of S_1' and S_2', by assumption, $\sigma_1'(id) = \sigma_2'(id)$. The condition holds.
   - Values of used variables in s_2 are not of value as enum labels in E, $\forall id \in$ OVar$(s_2) : \sigma_2'(id) \in E$.
     By assumption, initial values of used variables in s_2 are not of values as enum labels in E. S_2' and S_1' have equivalent computation of defined variables in S_2' and S_1'. Because enum labels are not defined in P_1, defined variables in the execution of S_2' and S_1' are not of values as enum labels in E.

   By the hypothesis IH, the lemma holds.

□

We show a auxiliary lemma telling that the two programs with updates of enumeration type extension have same set of used variables and the same set of defined variables.

Lemma 6.5. *If there are updates of enumeration type extension in a statement sequence S_2 against a statement sequence S_1, $S_2 \approx^S_E S_1$, then the output deciding variables in S_1 are a subset of those in S_2, OVar$(S_1) \subseteq$ OVar(S_2).*

Proof. By induction on the sum of the program size of S_1 and S_2. □

Lemma 6.6. *Let $S_1 = while_{\langle n_1 \rangle}(e)\,\{S_1'\}$ and $S_2 = while_{\langle n_2 \rangle}(e)$ $\{S_2'\}$ be two loop statements in programs P_1 and P_2 respectively where all of the following hold:*

- *Enumeration types EN_1 in P_1 are a proper subset of EN_2 in P_2, $EN_1 \subset EN_2$, such that there are a set of enum labels E only defined in P_2;*
- *When started in states agreeing on values of output deciding variables in both S_1' and S_2' as well as the input sequence variable and the I/O sequence variable, initial values of used variables in S_2' are not enum labels in E, and there are no inputs in S_2's execution translated into any label in E, $\forall x \in$ OVar$(S_1') \cup \{id_I, id_{IO}\}$ $\forall m_1(\sigma_1)\ m_2(\sigma_2) : \sigma_1(x) = \sigma_2(x)$, and S_1' and S_2' terminate in the same way, produce the same output sequence, and have equivalent computation of defined variables in S_1' and S_2' as well as the input sequence variable and the I/O sequence variable $((S_1', m_1) \equiv_H (S_2', m_2)) \wedge ((S_1', m_1) \equiv_O (S_2', m_2)) \wedge (\forall x \in$ OVar$(S_1) \cup$ OVar$(S_2) \cup \{id_I, id_{IO}\} :$ $(S_1', m_1) \equiv_x (S_2', m_2))$;*

If S_1 and S_2 start in states $m_1(loop_c^1, \sigma_1), m_2(loop_c^2, \sigma_2)$, with loop counters of S_1 and S_2 not initialized (S_1, S_2 have not executed yet), value stores agree on values of output deciding variables in S_1 and S_2 as well as the input sequence variable, the I/O sequence variable, initial values of used variables in S_2 are not of values as enum labels in E, no inputs are translated into enum labels in E, then, for any positive integer i, one of the following holds:

1. *Loop counters for S_1 and S_2 are always less than i if any is present, $\forall m_1'(loop_c^{1'})\, m_2'(loop_c^{2'}) : (S_1, m_1(loop_c^1, \sigma_1)) \xrightarrow{*} (S_1'', m_1'(loop_c^{1'})), loop_c^{1'}(n_1) < i, (S_2, m_2(loop_c^2, \sigma_2)) \xrightarrow{*} (S_2'', m_2'(loop_c^{2'})), loop_c^{2'}(n_2) < i$, S_1 and S_2 terminate in the same way, produce the same output sequence, and have equivalent computation of output deciding variables in both S_1 and S_2 and the input sequence variable, the I/O sequence variable, $(S_1, m_1) \equiv_H (S_2, m_2)$ and $(S_1, m_1) \equiv_O (S_2, m_2)$ and $\forall x \in ($OVar$(S_1) \cup$ OVar$(S_2)) \cup \{id_I, id_{IO}\} : (S_1, m_1) \equiv_x (S_2, m_2)$;*
2. *The loop counter of S_1 and S_2 are of value less than or equal to i, and there are no reachable configurations $(S_1, m_1(loop_c^{1_i}, \sigma_{1_i}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(loop_c^{2_i}, \sigma_{2_i}))$ from $(S_2,\ m_2(\sigma_2))$ where all of the following hold:*
   - *The loop counters of S_1 and S_2 are of value i, $loop_c^{1_i}(n_1) = loop_c^{2_i}(n_2) = i$.*
   - *Value stores σ_{1_i} and σ_{2_i} agree on values of output deciding variables in both S_1 and S_2 as well as the input sequence variable and the I/O sequence variable, $\forall x \in ($OVar$(S_1) \cap$ OVar$(S_2)) \cup \{id_I, id_{IO}\} : \sigma_{1_i}(x) = \sigma_{2_i}(x)$.*
3. *There are reachable configurations $(S_1, m_1(loop_c^{1_i}, \sigma_{1_i}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(loop_c^{2_i}, \sigma_{2_i}))$ from $(S_2,\ m_2(\sigma_2))$ where all of the following hold:*
   - *The loop counter of S_1 and S_2 are of value i, $loop_c^{1_i}(n_1) = loop_c^{2_i}(n_2) = i$.*
   - *Value stores σ_{1_i} and σ_{2_i} agree on values of output deciding variables in both S_1 and S_2 as well as the input sequence variable and the I/O sequence variable, $\forall x \in ($OVar$(S_1) \cap$ OVar$(S_2)) \cup \{id_I, id_{IO}\} : \sigma_{1_i}(x) = \sigma_{2_i}(x)$.*

Proof. By induction on i.
Base case.
We show that, when $i = 1$, one of the following holds:

1. Loop counters for S_1 and S_2 are always less than 1 if any is present, $\forall m_1'(\text{loop}_c^{1'})\, m_2'(\text{loop}_c^{2'}) : (S_1, m_1(\text{loop}_c^1, \sigma_1)) \xrightarrow{*} (S_1'', m_1'(\text{loop}_c^{1'})), \text{loop}_c^{1'}(n_1) < i, (S_2, m_2(\text{loop}_c^2, \sigma_2)) \xrightarrow{*} (S_2'', m_2'(\text{loop}_c^{2'})), \text{loop}_c^{2'}(n_2) < i$, S_1 and S_2 terminate in the same way, produce the same output sequence, and have equivalent computation of defined variables in both S_1 and S_2 and the input sequence variable, the I/O sequence variable, $(S_1, m_1) \equiv_H (S_2, m_2)$ and $(S_1, m_1) \equiv_O (S_2, m_2)$ and $\forall x \in (\text{Def}(S_1) \cap \text{Def}(S_2)) \cup \{id_I, id_{IO}\} : (S_1, m_1) \equiv_x (S_2, m_2)$;
2. Loop counters of S_1 and S_2 are of value less than or equal to 1 but there are no reachable configurations $(S_1, m_1(\text{loop}_c^{1_1}, \sigma_{1_i}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(\text{loop}_c^{2_1}, \sigma_{2_i}))$ from $(S_2,\ m_2(\sigma_2))$ where all of the following hold:
   - The loop counter of S_1 and S_2 are of value 1, $\text{loop}_c^{1_1}(n_1) = \text{loop}_c^{2_1}(n_2) = 1$.
   - Value stores σ_{1_1} and σ_{2_1} agree on values of used variables in both S_1 and S_2 as well as the input sequence variable, and the I/O sequence variable, $\forall x \in (\text{Use}(S_1) \cap \text{Use}(S_2)) \cup \{id_I, id_{IO}\} : \sigma_{1_1}(x) = \sigma_{2_1}(x)$.
3. There are reachable configuration $(S_1, m_1(\text{loop}_c^{1_1}, \sigma_{1_i}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(\text{loop}_c^{2_1}, \sigma_{2_i}))$ from $(S_2,\ m_2(\sigma_2))$ where all of the following hold:
   - The loop counter of S_1 and S_2 are of value 1, $\text{loop}_c^{1_1}(n_1) = \text{loop}_c^{2_1}(n_2) = 1$.
   - Value stores σ_{1_1} and σ_{2_1} agree on values of used variables in both S_1 and S_2 as well as the input sequence variable, and the I/O sequence variable, $\forall x \in (\text{Use}(S_1) \cap \text{Use}(S_2)) \cup \{id_I, id_{IO}\} : \sigma_{1_1}(x) = \sigma_{2_1}(x)$.

By definition, variables used in the predicate expression e of S_1 and S_2 are used in S_1 and S_2, Use$(e) \subseteq$ Use$(S_1) \cap$ Use(S_2). By assumption, value stores σ_1 and σ_2 agree on values of variables in Use(e), the predicate expression e evaluates to the same value w.r.t value stores σ_1 and σ_2 by Lemma D.2. There are three possibilities.

1. The evaluation of e crashes,
   $\mathcal{E}'[\![e]\!]\sigma_1 = \mathcal{E}'[\![e]\!]\sigma_2 = (\text{error}, v_{\mathfrak{of}})$.
   The execution of S_1 continues as follows:

   $$\begin{aligned}&(\text{while}_{\langle n_1 \rangle}(e)\,\{S_1'\}, m_1(\sigma_1))\\ \to&(\text{while}_{\langle n_1 \rangle}((\text{error}, v_{\mathfrak{of}}))\,\{S_1'\}, m_1(\sigma_1))\\ &\text{by the rule EEval'}\\ \to&(\text{while}_{\langle n_1 \rangle}(0)\,\{S_1'\}, m_1(1/\mathfrak{f}))\\ &\text{by the ECrash rule}\\ \xrightarrow{i}&(\text{while}_{\langle n_1 \rangle}(0)\,\{S_1'\}, m_1(1/\mathfrak{f}))\text{ for any } i > 0\\ &\text{by the Crash rule.}\end{aligned}$$

   Similarly, the execution of S_2 started from the state $m_2(\sigma_2)$ crashes. Therefore S_1 and S_2 terminate in the same way when started from m_1 and m_2 respectively. Because $\sigma_1(id_{IO}) = \sigma_2(id_{IO})$, the lemma holds.
2. The evaluation of e reduces to zero, $\mathcal{E}'[\![e]\!]\sigma_1 = \mathcal{E}'[\![e]\!]\sigma_2 = (0, v_{\mathfrak{of}})$.
   The execution of S_1 continues as follows.

$(\text{while}_{\langle n_1 \rangle}(e)\,\{S_1'\}, m_1(\sigma_1))$
$= (\text{while}_{\langle n_1 \rangle}((0, v_{\mathtt{of}}))\,\{S_1'\}, m_1(\sigma_1))$
by the rule EEval'
$\rightarrow (\text{while}_{\langle n_1 \rangle}(0)\,\{S_1'\}, m_1(\sigma_1))$
by the E-Oflow1 or E-Oflow2 rule
$\rightarrow (\text{skip}, m_1(\sigma_1))$ by the Wh-F rule.

Similarly, the execution of S_2 gets to the configuration (skip, $m_2(\sigma_2)$). Loop counters of S_1 and S_2 are less than 1 and value stores agree on values of used/defined variables in both S_1 and S_2 as well as the input sequence variable, and the I/O sequence variable.

3. The evaluation of e reduces to the same nonzero integer value, $\mathcal{E}'[\![e]\!]\sigma_1 = \mathcal{E}'[\![e]\!]\sigma_2 = (0, v_{\mathtt{of}})$.
Then the execution of S_1 proceeds as follows:

$(\text{while}_{\langle n_1 \rangle}(e)\,\{S_1'\}, m_1(\sigma_1))$
$= (\text{while}_{\langle n_1 \rangle}((v, v_{\mathtt{of}}))\,\{S_1'\}, m_1(\sigma_1))$
by the rule EEval'
$\rightarrow (\text{while}_{\langle n_1 \rangle}(v)\,\{S_1'\}, m_1(\sigma_1))$
by the E-Oflow1 or E-Oflow2 rule
$\rightarrow (S_1'; \text{while}_{\langle n_1 \rangle}(e)\,\{S_1'\}, m_1($
$\text{loop}_c^1[1/n_1], \sigma_1))$ by the Wh-T rule.

Similarly, the execution of S_2 proceeds to the configuration $(S_2'; \text{while}_{\langle n_2 \rangle}(e)\,\{S_2'\}, m_2(\text{loop}_c^2[1/n_1], \sigma_2))$.
By the assumption, we show that S_1' and S_2' terminate in the same way and produce the same output sequence when started in the state $m_1(\text{loop}_c^{1_1}, \sigma_1)$ and $m_2(\text{loop}_c^{2_1}, \sigma_2)$ respectively, and S_1' and S_2' have equivalent computation of variables defined in both statement sequences if both terminate. We need to show that all conditions are satisfied for the application of the assumption.

- There are no inputs translated into enum labels in E in the execution of S_2'.
The above condition is by assumption.
- Initial values of used variables in S_2' are not enum labels in E.
By the definition of used variables, Use$(S_2') \subseteq$ Use(S_2). By assumption, initial values of used variables in S_2 are not enum labels in E. The condition holds.
- Value stores σ_1 and σ_2 agree on values of used variables in S_1' and S_2' as well as the input, output, I/O sequence variable.
By definition, Use$(S_1') \subseteq$ Use(S_1). So are the cases to S_2' and S_2. In addition, value stores σ_1 and σ_2 are not changed in the evaluation of the predicate expression e. The condition holds.

By assumption, S_1' and S_2' terminate in the same way and produce the same output sequence when started in states $m_1(\text{loop}_c', \sigma_1)$ and $m_2(\text{loop}_c', \sigma_2)$. In addition, S_1' and S_2' have equivalent computation of variables used or defined in S_1' and S_2' when started in states $m_1(\text{loop}_c', \sigma_1)$ and $m_2(\text{loop}_c', \sigma_2)$.
Then there are two cases.

(a) S_1' and S_2' both do not terminate and produce the same output sequence.
By Lemma E.2, $S_1'; S_1$ and $S_2'; S_2$ both do not terminate and produce the same output sequence.

(b) S_1' and S_2' both terminate and have equivalent computation of variables defined in S_1' and S_2'.
By assumption, $(S_1', m_1(\text{loop}_c', \sigma_1)) \xrightarrow{*} (\text{skip}, m_1'(\text{loop}_c'', \sigma_1'))$; $(S_2', m_2(\text{loop}_c', \sigma_2)) \xrightarrow{*} (\text{skip}, m_2'(\text{loop}_c'', \sigma_2'))$ where $\forall x \in (\text{Def}(S_1') \cap \text{Def}(S_2')) \cup \{id_I, id_{IO}\}, \sigma_1'(x) = \sigma_2'(x)$.
By assumption,Use$(S_1') \subseteq$ Use(S_2') and Def(S_1') $=$ Def(S_2'). Then variables used in the predicate expression of S_1 and S_2 are either in variables used or defined in both S_1' and S_2' or not. Therefore value stores σ_2' and σ_1' agree on values of variables used in the expression e and even variables used or defined in S_1 and S_2.

Induction step on iterations

The induction hypothesis (IH) is that, when $i \geq 1$, one of the following holds:

1. Loop counters for S_1 and S_2 are always less than i if any is present, $\forall m_1'(\text{loop}_c^{1'})\, m_2'(\text{loop}_c^{2'}) : (S_1, m_1(\text{loop}_c^1, \sigma_1)) \xrightarrow{*} (S_1'', m_1'(\text{loop}_c^{1'})), \text{loop}_c^{1'}(n_1) < i, (S_2, m_2(\text{loop}_c^2, \sigma_2)) \xrightarrow{*} (S_2'', m_2'(\text{loop}_c^{2'})), \text{loop}_c^{2'}(n_2) < i$, S_1 and S_2 terminate in the same way, produce the same output sequence, and have equivalent computation of used/defined variables in both S_1 and S_2 and the input sequence variable, the I/O sequence variable, $(S_1, m_1) \equiv_H (S_2, m_2)$ and $(S_1, m_1) \equiv_O (S_2, m_2)$ and $\forall x \in (\text{Def}(S_1) \cap \text{Def}(S_2)) \cup \{id_I, id_{IO}\} : (S_1, m_1) \equiv_x (S_2, m_2)$;
2. The loop counter of S_1 and S_2 are of value less than or equal to i, and there are no reachable configurations $(S_1, m_1(\text{loop}_c^{1_i}, \sigma_{1_i}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(\text{loop}_c^{2_i}, \sigma_{2_i}))$ from $(S_2,\ m_2(\sigma_2))$ where all of the following hold:
   - The loop counters of S_1 and S_2 are of value i, $\text{loop}_c^{1_i}(n_1) = \text{loop}_c^{2_i}(n_2) = i$.
   - Value stores σ_{1_i} and σ_{2_i} agree on values of used variables in both S_1 and S_2 as well as the input sequence variable, and the I/O sequence variable, $\forall x \in (\text{Use}(S_1) \cap \text{Use}(S_2)) \cup \{id_I, id_{IO}\} : \sigma_{1_i}(x) = \sigma_{2_i}(x)$.
3. There are reachable configurations $(S_1, m_1(\text{loop}_c^{1_i}, \sigma_{1_i}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(\text{loop}_c^{2_i}, \sigma_{2_i}))$ from $(S_2,\ m_2(\sigma_2))$ where all of the following hold:
   - The loop counter of S_1 and S_2 are of value i, $\text{loop}_c^{1_i}(n_1) = \text{loop}_c^{2_i}(n_2) = i$.
   - Value stores σ_{1_i} and σ_{2_i} agree on values of used variables in both S_1 and S_2 as well as the input sequence variable, and the I/O sequence variable, $\forall x \in (\text{Use}(S_1) \cap \text{Use}(S_2)) \cup \{id_I, id_{IO}\} : \sigma_{1_i}(x) = \sigma_{2_i}(x)$.

Then we show that, when $i + 1$, one of the following holds: The induction hypothesis (IH) is that, when $i \geq 1$, one of the following holds:

1. Loop counters for S_1 and S_2 are always less than $i + 1$ if any is present, $\forall m_1'(\text{loop}_c^{1'})\, m_2'(\text{loop}_c^{2'}) : (S_1, m_1(\text{loop}_c^1, \sigma_1)) \xrightarrow{*} (S_1'', m_1'(\text{loop}_c^{1'})), \text{loop}_c^{1'}(n_1) < i{+}1, (S_2, m_2(\text{loop}_c^2, \sigma_2)) \xrightarrow{*} (S_2'', m_2'(\text{loop}_c^{2'})), \text{loop}_c^{2'}(n_2) < i + 1$, S_1 and S_2 terminate in the same way, produce the same output sequence, and have equivalent computation of used/defined variables in both S_1 and S_2 and the input sequence variable, the I/O sequence variable, $(S_1, m_1) \equiv_H (S_2, m_2)$ and $(S_1, m_1) \equiv_O (S_2, m_2)$ and $\forall x \in (\text{Def}(S_1) \cap \text{Def}(S_2)) \cup \{id_I, id_{IO}\} : (S_1, m_1) \equiv_x (S_2, m_2)$;
2. The loop counter of S_1 and S_2 are of value less than or equal to $i + 1$, and there are no reachable configurations $(S_1, m_1(\text{loop}_c^{1_{i+1}}, \sigma_{1_{i+1}}))$ from $(S_1, m_1(\sigma_1))$,
$(S_2, m_2(\text{loop}_c^{2_{i+1}}, \sigma_{2_{i+1}}))$ from $(S_2,\ m_2(\sigma_2))$ where all of the following hold:
   - The loop counters of S_1 and S_2 are of value $i + 1$, $\text{loop}_c^{1_{i+1}}(n_1) = \text{loop}_c^{2_{i+1}}(n_2) = i + 1$.
   - Value stores $\sigma_{1_{i+1}}$ and $\sigma_{2_{i+1}}$ agree on values of used variables in both S_1 and S_2 as well as the input sequence variable, and the I/O sequence variable, $\forall x \in (\text{Use}(S_1) \cap \text{Use}(S_2)) \cup \{id_I, id_{IO}\} : \sigma_{1_{i+1}}(x) = \sigma_{2_{i+1}}(x)$.

3. There are reachable configurations $(S_1, m_1(\text{loop}_c^{1i+1}, \sigma_{1_i}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(\text{loop}_c^{2i+1}, \sigma_{2_i}))$ from $(S_2,\ m_2(\sigma_2))$ where all of the following hold:
   - The loop counter of S_1 and S_2 are of value i, $\text{loop}_c^{1i+1}(n_1) = \text{loop}_c^{2i+1}(n_2) = i + 1$.
   - Value stores $\sigma_{1_{i+1}}$ and $\sigma_{2_{i+1}}$ agree on values of used variables in both S_1 and S_2 as well as the input sequence variable, and the I/O sequence variable, $\forall x \in$ (Use(S_1) $\cap$ Use(S_2)) $\cup \{id_I, id_{IO}\}$: $\sigma_{1_{i+1}}(x) = \sigma_{2_{i+1}}(x)$.

By hypothesis IH, there is no configuration where loop counters of S_1 and S_2 are of value $i + 1$ when any of the following holds:

1. Loop counters for S_1 and S_2 are always less than i if any is present, $\forall m_1'(\text{loop}_c^{1'})\, m_2'(\text{loop}_c^{2'})$: $(S_1, m_1(\text{loop}_c^1, \sigma_1)) \xrightarrow{*} (S_1'', m_1'(\text{loop}_c^{1'}))$, $\text{loop}_c^{1'}(n_1) < i$, $(S_2, m_2(\text{loop}_c^2, \sigma_2)) \xrightarrow{*} (S_2'', m_2'(\text{loop}_c^{2'}))$, $\text{loop}_c^{2'}(n_2) < i$, S_1 and S_2 terminate in the same way, produce the same output sequence, and have equivalent computation of used/defined variables in both S_1 and S_2 and the input sequence variable, the I/O sequence variable, $(S_1, m_1) \equiv_H (S_2, m_2)$ and $(S_1, m_1) \equiv_O (S_2, m_2)$ and $\forall x \in$ (Def(S_1) $\cap$ Def(S_2)) $\cup \{id_I, id_{IO}\}$: $(S_1, m_1) \equiv_x (S_2, m_2)$;
2. The loop counter of S_1 and S_2 are of value less than or equal to i, and there are no reachable configurations $(S_1, m_1(\text{loop}_c^{1i}, \sigma_{1_i}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(\text{loop}_c^{2i}, \sigma_{2_i}))$ from $(S_2,\ m_2(\sigma_2))$ where all of the following hold:
   - The loop counters of S_1 and S_2 are of value i, $\text{loop}_c^{1i}(n_1) = \text{loop}_c^{2i}(n_2) = i$.
   - Value stores σ_{1_i} and σ_{2_i} agree on values of used variables in both S_1 and S_2 as well as the input sequence variable, and the I/O sequence variable, $\forall x \in$ (Use(S_1) $\cap$ Use(S_2)) $\cup$ $\{id_I, id_{IO}\}$: $\sigma_{1_i}(x) = \sigma_{2_i}(x)$.

When there are reachable configurations $(S_1, m_1(\text{loop}_c^{1i}, \sigma_{1_i}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(\text{loop}_c^{2i}, \sigma_{2_i}))$ from $(S_2,\ m_2(\sigma_2))$ where all of the following hold:

- The loop counter of S_1 and S_2 are of value i, $\text{loop}_c^{1i}(n_1) = \text{loop}_c^{2i}(n_2) = i$.
- Value stores σ_{1_i} and σ_{2_i} agree on values of used variables in both S_1 and S_2 as well as the input sequence variable, and the I/O sequence variable, $\forall x \in$ (Use(S_1) $\cap$ Use(S_2)) $\cup$ $\{id_I, id_{IO}\}$: $\sigma_{1_i}(x) = \sigma_{2_i}(x)$.

By similar argument in base case, we have one of the following holds:

1. Loop counters for S_1 and S_2 are always less than $i + 1$ if any is present, $\forall m_1'(\text{loop}_c^{1'})\, m_2'(\text{loop}_c^{2'})$: $(S_1, m_1(\text{loop}_c^1, \sigma_1)) \xrightarrow{*} (S_1'', m_1'(\text{loop}_c^{1'}))$, $\text{loop}_c^{1'}(n_1) < i+1$, $(S_2, m_2(\text{loop}_c^2, \sigma_2)) \xrightarrow{*} (S_2'', m_2'(\text{loop}_c^{2'}))$, $\text{loop}_c^{2'}(n_2) < i$, S_1 and S_2 terminate in the same way, produce the same output sequence, and have equivalent computation of used/defined variables in both S_1 and S_2 and the input sequence variable, the I/O sequence variable, $(S_1, m_1) \equiv_H (S_2, m_2)$ and $(S_1, m_1) \equiv_O (S_2, m_2)$ and $\forall x \in$ (Def(S_1) $\cap$ Def(S_2)) $\cup \{id_I, id_{IO}\}$: $(S_1, m_1) \equiv_x (S_2, m_2)$;
2. The loop counter of S_1 and S_2 are of value less than or equal to $i + 1$, and there are no reachable configurations $(S_1, m_1(\text{loop}_c^{1i}, \sigma_{1_i}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(\text{loop}_c^{2i}, \sigma_{2_i}))$ from $(S_2,\ m_2(\sigma_2))$ where all of the following hold:
   - The loop counters of S_1 and S_2 are of value i, $\text{loop}_c^{1i+1}(n_1) = \text{loop}_c^{2i+1}(n_2) = i + 1$.
   - Value stores $\sigma_{1_{i+1}}$ and $\sigma_{2_{i+1}}$ agree on values of used variables in both S_1 and S_2 as well as the input sequence variable, and the I/O sequence variable, $\forall x \in$ (Use(S_1) $\cap$ Use(S_2)) $\cup \{id_I, id_{IO}\}$: $\sigma_{1_{i+1}}(x) = \sigma_{2_{i+1}}(x)$.
3. There are reachable configurations $(S_1, m_1(\text{loop}_c^{1i+1}, \sigma_{1_{i+1}}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(\text{loop}_c^{2i+1}, \sigma_{2_{i+1}}))$ from $(S_2,$ $m_2(\sigma_2))$ where all of the following hold:
   - The loop counter of S_1 and S_2 are of value i, $\text{loop}_c^{1i+1}(n_1) = \text{loop}_c^{2i+1}(n_2) = i$.
   - The loop counter of S_1 and S_2 are of value i, $\text{loop}_c^{1i+1}(n_1) = \text{loop}_c^{2i+1}(n_2) = i$.
   - Value stores $\sigma_{1_{i+1}}$ and $\sigma_{2_{i+1}}$ agree on values of used variables in both S_1 and S_2 as well as the input sequence variable, and the I/O sequence variable, $\forall x \in$ (Use(S_1) $\cap$ Use(S_2)) $\cup \{id_{I+1}, id_{IO}\}$: $\sigma_{1_{i+1}}(x) = \sigma_{2_{i+1}}(x)$.

□

6.3 Proof rule for variable type weakening

In programs, variable types are changed either to allow for larger ranges (weakening). For example, an integer variable might be changed to become a long variable to avoid integer overflow. Adding a new enumeration value can is also type weakening. Increasing array size is another example of weakening. Allowing for type weakening is essentially an assumption about the intent behind the update. The kinds of weakening that should be allowed are application dependent and would need to be defined by the user in general. The type weakening considered are either changes of type Int to Long or increase of array size. These updates fix integer overflow or array index out of bound. In order to prove the update of variable type weakening to be backward compatible, we assume that there are no integer overflow and array index out of bound in execution of the old program and the updated program. In conclusion, the old program and the new program produce the same output sequence because the integer overflow and index out of bound errors fixed by the new program do not occur.

We formalize the update of variable type weakening, then we show that the updated program produce the same output sequence as the old program in executions if there are no integer overflow or index out of bound exceptions related to variables with type changes. First, we define a relation between variable definitions showing the type weakening.

Definition 28. (Cases of type weakening) *We say there is type weakening from a sequence of variable definitions V_1 to V_2, written $V_1 \nearrow_\tau V_2$, iff one of the following holds:*

1. $V_1 =$ "*Int id*", $V_2 =$ "*Long id*";
2. $V_1 =$ "$\tau\ id[n_2]$", $V_2 =$ "$\tau\ id[n_1]$" *where* $n_2 > n_1$;
3. $V_1 = V_1'$, "$\tau_1\ id_1$", $V_2 = V_2'$, "$\tau_2\ id_2$" *where* $(V_1' \nearrow_\tau V_2') \wedge$ ("$\tau_1\ id_1$" $\nearrow_\tau$ "$\tau_2\ id_2$");
4. $V_1 = V_1'$, "$\tau_1\ id_1[n_1]$", $V_2 = V_2'$, "$\tau_2\ id_2[n_2]$" *where* $(V_1' \nearrow_\tau V_2') \wedge$ ("$\tau_1\ id_1[n_1]$" $\nearrow_\tau$ "$\tau_2\ id_2[n_2]$");

The following is the generalized definition of variable type weakening.

Definition 29. (Variable type weakening) *We say that there are updates of variable type weakening in the program* $P_2 = Pmpt; EN; V_2; S_{entry}$ *compared with the program* $P_1 = Pmpt; EN; V_1; S_{entry}$, *written* $P_2 \approx_\tau^S P_1$, *iff* $V_1 \nearrow_\tau V_2$.

We show that two programs terminate in the same way, produce the same output sequence, and have equivalent computation of defined variables in both programs in valid executions if there are updates of variable type weakening between them.

Lemma 6.7. *Let $P_1 = EN; V_1; S_{entry}$ and $P_2 = EN; V_2; S_{entry}$ be two programs where there are updates of variable type weakening, $P_2 \approx^S_\tau P_1$. If the programs P_1 and P_2 start in states $m_1(\sigma_1)$ and $m_2(\sigma_2)$ such that both of the following hold:*

- *Value stores σ_1 and σ_2 agree on values of variables used in S_{entry} as well as the input sequence variable, the I/O sequence variable, $\forall x \in Use(S_{entry}) \cup \{id_I, id_{IO}\} \ : \ \sigma_1(x) = \sigma_2(x)$;*
- *There is no integer overflow or index out of bound exceptions related to variables of type change;*

then S_{entry} in the program P_1 and P_2 terminate in the same way, produce the same output sequence, and when S_{entry} both terminate, they have equivalent computation of defined variables in S_{entry} in both programs as well as the input sequence variable, the I/O sequence variable,

- *$(S_{entry}, m_1) \equiv_H (S_{entry}, m_2)$;*
- *$(S_{entry}, m_1) \equiv_O (S_{entry}, m_2)$;*
- *$\forall x \in Def(S) \cup \{id_I, id_{IO}\}$:
$(S_{entry}, m_1) \equiv_x (S_{entry}, m_2)$;*

Because S_{entry} are the exactly same in both programs P_1 and P_2, we omit the straightforward proof. Instead, we show that, if there is no array index out of bound and integer overflow in executions of the old program, then there is no array index out of bound or integer overflow in executions of updated program due to the increase of array index and change of type Int to Long.

Proof. The proof is straightforward because the statement sequence S is same in programs P_1 and P_2. The only point is that if there is array index out of bound or integer overflow in execution of S in P_1, then there is no array index out of bound or integer overflow in execution of S in P_2. To show the point, we present the argument for the array index out of bound and integer overflow separately.

1. We show that, as to one expression $id_1[id_2]$, there is no array index out of bound in P_2 if there is no array index out of bound in P_1 when P_1 and P_2 are in states agreeing on values of used variables in P_1 and P_2;

   $(id_1[id_2], m_1(\sigma_1))$
   $\rightarrow (id_1[v], m_1(\sigma_1))$ by the rule Var

   Similarly, $(id_1[id_2], m_2(\sigma_2)) \rightarrow (id_1[v], m_2(\sigma_2))$. By Definition 28, the array bound of id_1 in P_2 is no less than that in P_1, then there is no array out of bound exception in evaluation of $id_1[id_2]$ in P_2 if there is no array out of bound exception in evaluation of $id_1[id_2]$ in P_1.
2. We show that, as to one expression e, there is no integer overflow in evaluation of e in P_2 if there is no integer overflow in evaluation of e in P_1;

   $(e, m_1(\sigma_1))$
   $\rightarrow ((v_e, v_{\mathtt{of}}), m_1(\sigma_1))$ by the rule EEval'

   When every used variable in the expression e is of same type in P_1 and P_2, then the evaluation of the expression e in P_2 is of the same result $(v_e, v_{\mathtt{of}})$ in P_1. When every used variable in the expression e is of type Int in P_1 and of type Long P_2, then there is no integer overflow in the evaluation of the expression e in P_2 if there is no integer overflow in the evaluation of the expression e in P_1. This is because the values of type Long are a superset of those of type Int.

□

```
1:                          1':  If (1/(a − 5)) then
2:                          2':      skip
3:  output a                3':  output a

    old                          new
```

Figure 17: Exit-on-error

6.4 Proof rule for exit on errors

Another bugfix is called "exit-on-error", which causes the program to exit in observation of application-semantic-dependent errors. Figure 17 shows an example of exit-on-error update. In the example, the fixed bugs refer to the program semantic error that $a = 5$. Instead of using an "exit" statement, we rely on the crash from expression evaluations to formalize the update class. In order to prove the update of exit-on-error to be backward compatible, we assume that there are no application related error in executions of the old program. Therefore, the two programs produce the same output sequence because the extra check does not cause the new program's execution to crash.

The following is the generalized definition of the update class "exit-on-error".

Definition 30. **(Exit on error)** *We say a statement sequence S_2 includes updates of exit-on-err from a statement sequence S_1, written $S_2 \approx^S_{Exit} S_1$, iff one of the following holds:*

1. *S_2 = "If(e) then$\{$skip$\}$ else$\{$skip$\}$"; S_1;*
2. *S_1 = "If(e) then$\{S_1^t\}$ else$\{S_1^f\}$", S_2 = "If(e) then$\{S_2^t\}$ else$\{S_2^f\}$" where both of the following hold*
   - *$S_2^t \approx^S_{Exit} S_1^t$;*
   - *$S_2^f \approx^S_{Exit} S_1^f$;*
3. *S_1 = "while$_{\langle n_1 \rangle}(e)\ \{S_1'\}$", S_2 = "while$_{\langle n_2 \rangle}(e)\ \{S_2'\}$" where $S_2' \approx^S_{Exit} S_1'$;*
4. *$S_1 \approx^S_O S_2$;*
5. *$S_1 = S_1'; s_1$ and $S_2 = S_2'; s_2$ such that both of the following hold:*
   - *$S_2' \approx^S_{Exit} S_1'$;*
   - *$S_2' \approx^S_H S_1'$;*
   - *$\forall x \in Imp(s_1, id_{IO}) \cup Imp(s_1, id_{IO})\ :\ S_2' \approx^S_x S_1'$;*
   - *$s_2 \approx^S_{Exit} s_1$;*

Though the bugfix in Definition 30 is not in rare execution in the first case, the definition shows the basic form of bugfix clearly.

We show that two programs terminate in the same way, produce the same output sequence, and have equivalent computation of defined variables in both programs in valid executions if there are updates of exit-on-error between them.

Lemma 6.8. *Let S_1 and S_2 be two statement sequences respectively where there are updates of exit-on-error in S_2 against S_1, $S_2 \approx^S_{Exit} S_1$. If S_1 and S_2 start in states $m_1(\sigma_1)$ and $m_2(\sigma_2)$ such that both of the following hold:*

- *Value stores σ_1 and σ_2 agree on values of variables used in both S_1 and S_2 as well as the input sequence variable and the I/O sequence variable, $\forall x \in (Use(S_1) \cap Use(S_2)) \cup \{id_I, id_{IO}\}$: $\sigma_1(x) = \sigma_2(x)$;*
- *There are no program semantic errors related to the extra check in the update of exit-on-error in the execution of S_1;*

then S_1 and S_2 terminate in the same way, produce the same output sequence, and when S_1 and S_2 both terminate, they have equivalent computation of defined variables in both S_1 and S_2 as well as the input sequence variable and the I/O sequence variable,

- *$(S_1, m_1) \equiv_H (S_2, m_2)$;*
- *$(S_1, m_1) \equiv_O (S_2, m_2)$;*

- $\forall x \in (\mathit{Def}(S_1) \cap \mathit{Def}(S_2)) \cup \{id_I, id_{IO}\}$: $(S_1, m_1) \equiv_x (S_2, m_2)$;

Proof. By induction on the sum of the program size of S_1 and S_2, size(S_1) + size(S_2).
Base case. $S_1 = s$ and $S_2 =$ "If(e) then{skip} else{skip}"; s;

By assumption, there is no program semantic error related to the update of exit-on-error. Then the evaluation of the predicate expression e in the first statement of S_2 does not crash. W.l.o.g., the expression e evaluates to zero. Then the execution of S_2 proceeds as follows.

(If(e) then{skip} else{skip}; $s, m_2(\sigma_2)$)
$\rightarrow$(If$((0, v_{\mathfrak{of}})$) then{skip} else{skip}; $s, m_2(\sigma_2)$)
by the rule EEval'
$\rightarrow$(If(0) then{skip} else{skip}; $s, m_2(\sigma_2)$)
by the rule E-Oflow1 or E-Oflow2.
$\rightarrow$(skip; $s, m_2(\sigma_2)$) by the rule If-F
$\rightarrow(s, m_2(\sigma_2))$ by the rule Seq.

Value stores σ_2 are not changed in the execution of $(S_2, m_2(\sigma_2)) \xrightarrow{*} (s, m_2(\sigma_2))$. By assumption, σ_1 and σ_2 agree on values of used variables as well as the input sequence variable, and the I/O sequence variable, $\forall x \in$ Use$(S_2) \cap$ Use$(S_1) \cup \{id_I, id_{IO}\}$: $\sigma_1(id) = \sigma_2(id)$. By semantics, S_1 and S_2 terminate in the same way, produce the same output sequence, and have equivalent computation of defined variables in both S_1 and S_2 as well as the input sequence variable, and the I/O sequence variable. Then this lemma holds.
Induction step.

The hypothesis is that this lemma holds when the sum k of the program size of S_1 and S_2 are great than or equal to 4, $k \geq 4$.

We then show that this lemma holds when the sum of the program size of S_1 and S_2 is $k + 1$. There are cases to consider.

1. S_1 and S_2 are both "If" statement:
   $S_1 =$ "If(e) then$\{S_1^t\}$ else$\{S_1^f\}$", $S_2 =$ "If(e) then$\{S_2^t\}$ else$\{S_2^f\}$" where both of the following hold
   - $(S_2^t \approx^S_{\text{Exit}} S_1^t)$;
   - $(S_2^f \approx^S_{\text{Exit}} S_1^f)$;

   By the definition of Use(S_1), variables used in the predicate expression e are a subset of used variables in S_1 and S_2, Use$(e) \subseteq$ Use$(S_1) \cap$ Use(S_2). By assumption, corresponding variables used in e are of same value in value stores σ_1 and σ_2. By Lemma D.1, the expression evaluates to the same value w.r.t value stores σ_1 and σ_2. There are three possibilities.
   (a) The evaluation of e crashes, $\mathcal{E}'[\![e]\!]\sigma_1 = \mathcal{E}'[\![e]\!]\sigma_2 =$ (error, $v_{\mathfrak{of}}$).
   The execution of S_1 continues as follows:
   (If(e) then$\{S_1^t\}$ else$\{S_1^f\}, m_1(\sigma_1)$)
   $\rightarrow$(If((error, $v_{\mathfrak{of}}$)) then$\{S_1^t\}$ else$\{S_1^f\}, m_1(\sigma_1)$)
   by the rule EEval'
   $\rightarrow$(If(0) then$\{S_1^t\}$ else$\{S_1^f\}, m_1(1/\mathfrak{f})$)
   by the ECrash rule
   $\xrightarrow{i}$(If(0) then$\{S_1^t\}$ else$\{S_1^f\}, m_1(1/\mathfrak{f})$) for any $i > 0$
   by the Crash rule.
   Similarly, the execution of S_2 started from the state $m_2(\sigma_2)$ crashes. The lemma holds.
   (b) The evaluation of e reduces to zero, $\mathcal{E}'[\![e]\!]\sigma_1 = \mathcal{E}'[\![e]\!]\sigma_2 = (0, v_{\mathfrak{of}})$.
   The execution of S_1 continues as follows.
   (If(e) then$\{S_1^t\}$ else$\{S_1^f\}, m_1(\sigma_1)$)
   = (If$((0, v_{\mathfrak{of}}))$ then$\{S_1^t\}$ else$\{S_1^f\}, m_1(\sigma_1)$)
   by the rule EEval'
   $\rightarrow$(If(0) then$\{S_1^t\}$ else$\{S_1^f\}, m_1(\sigma_1)$)
   by the E-Oflow1 or E-Oflow2 rule
   $\rightarrow(S_1^f, m_1(\sigma_1))$ by the If-F rule.
   Similarly, the execution of S_2 gets to the configuration $(S_2^f, m_2(\sigma_2))$.
   By the hypothesis IH, we show the lemma holds. We need to show that all conditions are satisfied for the application of the hypothesis IH.
   - $(S_2^f \approx^S_{\text{Exit}} S_1^f)$
     By assumption.
   - The sum of the program size of S_1^f and S_2^f is less than k, size(S_1^f) + size$(S_2^f) < k$.
     By definition, size$(S_1) = 1+$size$(S_1^t)+$size(S_1^f). Then, size(S_1^f) + size$(S_2^f) < k + 1 - 2 = k - 1$.
   - Value stores σ_1 and σ_2 agree on values of used variables in S_1^f and S_2^f as well as the input, I/O sequence variable. By definition, Use$(S_1^f) \subseteq$ Use(S_1). So are the cases to S_2^f and S_2. In addition, value stores σ_1 and σ_2 are not changed in the evaluation of the predicate expression e. The condition holds.
   - There are no program semantic error related to the extra check in the update of exit-on-error in the execution of S_2.
     By assumption.

   By the hypothesis IH, the lemma holds.
   (c) The evaluation of e reduces to the same nonzero integer value, $\mathcal{E}'[\![e]\!]\sigma_1 = \mathcal{E}'[\![e]\!]\sigma_2 = (v, v_{\mathfrak{of}})$ where $v \neq 0$.
   By similar to the second subcase above.
2. S_1 and S_2 are both "while" statements:
   $S_1 =$ "while$_{\langle n \rangle}(e)\,\{S_1'\}$", $S_2 =$ "while$_{\langle n \rangle}(e)\,\{S_2'\}$" where $(S_2' \approx^S_{\text{exit}} S_1')$;
   By Lemma 6.10, we show this lemma holds. We need to show that all required conditions are satisfied for the application of Lemma 6.10.
   - The output deciding variables in S_1' are a subset of those in S_2', OVar(S_1') = OVar(S_2');
     By Lemma 6.9.
   - When started in states $m_1'(\sigma_1'), m_2'(\sigma_2')$ where value stores σ_1' and σ_2' agree on values of used variables in both S_1' and S_2' as well as the input sequence variable, and the I/O sequence variable, then S_1' and S_2' terminate in the same way, produce the same output sequence, and have equivalent computation of defined variables in both S_1 and S_2 as well as the input sequence variable, and the I/O sequence variable.
     By the induction hypothesis IH. This is because the sum of the program size of S_1' and S_2' is less than k. By definition, size$(S_1) = 1 +$ size(S_1').

   By Lemma 6.10, this lemma holds.
3. $S_1 = S_1'; s_1$ and $S_2 = S_2'; s_2$ where both of the following hold:
   - $(S_2' \approx^S_{\text{Exit}} S_1')$;
   - $(s_2 \approx^S_{\text{Exit}} s_1)$;

   By the hypothesis IH, we show S_2' and S_1' terminate in the same way and produce the same output sequence and when S_2' and S_1' both terminate, S_2' and S_1' have equivalent terminating computation of variables used or defined in S_2' and S_1' as well as the input sequence variable, and the I/O sequence variable.
   We show all the required conditions are satisfied for the application of the hypothesis IH.
   - $(S_2' \approx^S_{\text{Exit}} S_1')$.
     By assumption.

- The sum of the program size of S_1' and S_2' is less than k, $\text{size}(S_1') + \text{size}(S_2') < k$.
  By definition, $\text{size}(S_2) = \text{size}(s_2) + \text{size}(S_2')$ where $\text{size}(s_2) < 1$. Then, $\text{size}(S_2') + \text{size}(S_1') < k + 1 - \text{size}(s_2) - \text{size}(s_1) < k$.
- Value stores σ_1 and σ_2 agree on values of used variables in both S_2' and S_1' as well as the input, I/O sequence variable.
  By definition, $\text{Use}(S_2') \subseteq \text{Use}(S_2)$, $\text{Use}(S_1') \subseteq \text{Use}(S_1)$. The condition holds.

By the hypothesis IH, one of the following holds:

(a) S_1' and S_2' both do not terminate.
By Lemma E.2, executions of $S_1 = S_1'; s_1$ and $S_2 = S_2'; s_2$ both do not terminate and produce the same output sequence.

(b) S_1' and S_2' both terminate.
By assumption, $(S_2', m_2(\sigma_2)) \xrightarrow{*} (\text{skip}, m_2'(\sigma_2'))$, $(S_1', m_1(\sigma_1)) \xrightarrow{*} (\text{skip}, m_1'(\sigma_1'))$.
By Corollary E.1, $(S_2'; s_2, m_2(\sigma_2)) \xrightarrow{*} (s_2, m_2'(\sigma_2'))$, $(S_1'; s_1, m_1(\sigma_1)) \xrightarrow{*} (s_1, m_1'(\sigma_1'))$.
By the hypothesis IH, we show that s_2 and s_1 terminate in the same way, produce the same output sequence and when s_2 and s_1 both terminate, s_2 and s_1 have equivalent computation of variables defined in both s_1 and s_2 and the input, and I/O sequence variables.
We need to show that all conditions are satisfied for the application of the hypothesis IH.

- There are updates of "exit-on-error" between s_2 and s_1, $s_2 \approx^{S}_{\text{Exit}} s_1$;
  By assumption, $s_2 \approx^{S}_{\text{Exit}} s_1$.
- The sum of the program size s_2 and s_1 is less than or equals to k;
  By definition, $\text{size}(S_2') \geq 1, \text{size}(S_1') \geq 1$. Therefore, $\text{size}(s_2) + \text{size}(s_1) < k + 1 - \text{size}(S_2') - \text{size}(S_1') \leq k$.
- Value stores σ_1' and σ_2' agree on values of output deciding variables in s_2 and s_1 as well as the input, I/O sequence variable.
  By Lemma 6.9, $\text{OVar}(s_1) \subseteq \text{OVar}(s_2)$, then $\text{OVar}(s_2) \cap \text{OVar}(s_1) = \text{OVar}(s_1)$. For any variable id in $\text{Use}(s_1)$, if id is not in $\text{OVar}(S_1')$, then the value of id is not changed in the execution of S_1' and S_2', $\sigma_1'(id) = \sigma_1(id) = \sigma_2(id) = \sigma_2'(id)$. Otherwise, the variable id is defined in the execution of S_1' and S_2', by assumption, $\sigma_1'(id) = \sigma_2'(id)$. The condition holds.
- There are no program semantic errors related to the extra check in the update of exit-on-error in the execution of S_2.
  By assumption.

By the hypothesis IH, the lemma holds.

□

We list the auxiliary lemmas below. One lemma shows that, if there are updates of exit-on-error between two statement sequences, then there are same set of defined variables in the two statement sequences, and the used variables in the update program are the superset of those in the old program.

Lemma 6.9. *Let S_2 be a statement sequence and S_1 where there are updates of exit-on-error, $S_2 \approx^{S}_{Exit} S_1$. Then output deciding variables in S_1 are a subset of those in S_2, $OVar(S_1) \subseteq OVar(S_2)$.*

Proof. By induction on the sum of the program size of S_1 and S_2. □

Lemma 6.10. *Let $S_1 = while_{\langle n_1 \rangle}(e)\ \{S_1'\}$ and $S_2 = while_{\langle n_2 \rangle}(e)\ \{S_2'\}$ be two loop statements where all of the following hold:*

- *the output deciding variables in S_1' are a subset of those in S_2', $OVar(S_1') \subseteq OVar(S_2') = OVar(S)$;*
- *When started in states $m_1'(\sigma_1'), m_2'(\sigma_2')$ where*
  - *Value stores agree on values of output deciding variables in both S_1' and S_2' as well as the input sequence variable, and the I/O sequence variable, $\forall x \in OVar(S_2') \cup \{id_I, id_{IO}\} \forall m_1'(\sigma_1')\, m_2'(\sigma_2') : \sigma_1'(x) = \sigma_2'(x)$;*
  - *There are no program semantic errors related to the extra check in the update of exit-on-error in executions of S_1' and S_2';*

  then S_1' and S_2' terminate in the same way, produce the same output sequence, and have equivalent computation of defined variables in S_1' and S_2' as well as the input sequence variable, and the I/O sequence variable $((S_1', m_1) \equiv_H (S_2', m_2)) \wedge ((S_1', m_1) \equiv_O (S_2', m_2)) \wedge (\forall x \in OVar(S) \cup \{id_I, id_{IO}\} : (S_1', m_1) \equiv_x (S_2', m_2))$;

If S_1 and S_2 start in states $m_1(loop_c^1, \sigma_1), m_2(loop_c^2, \sigma_2)$ respectively, with loop counters of S_1 and S_2 not initialized (S_1, S_2 have not executed yet), value stores agree on values of used variables in S_1 and S_2, and there are no program semantic errors related to the extra check in the update of exit-on-error, then, for any positive integer i, one of the following holds:

1. *Loop counters for S_1 and S_2 are always less than i if any is present, $\forall m_1'(loop_c^{1'})\, m_2'(loop_c^{2'}) : (S_1, m_1(loop_c^1, \sigma_1)) \xrightarrow{*} (S_1'', m_1'(loop_c^{1'})), loop_c^{1'}(n_1) < i, (S_2, m_2(loop_c^2, \sigma_2)) \xrightarrow{*} (S_2'', m_2'(loop_c^{2'})), loop_c^{2'}(n_2) < i$, S_1 and S_2 terminate in the same way, produce the same output sequence, and have equivalent computation of output deciding variables in S_1 and S_2 and the input sequence variable, the I/O sequence variable, $(S_1, m_1) \equiv_H (S_2, m_2)$ and $(S_1, m_1) \equiv_O (S_2, m_2)$ and $\forall x \in (OVar(S_1) \cup OVar(S_2)) \cup \{id_I, id_{IO}\} : (S_1, m_1) \equiv_x (S_2, m_2)$;*
2. *The loop counter of S_1 and S_2 are of value less than or equal to i, and there are no reachable configurations $(S_1, m_1(loop_c^{1_i}, \sigma_{1_i}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(loop_c^{2_i}, \sigma_{2_i}))$ from $(S_2, m_2(\sigma_2))$ where all of the following hold:*
   - *The loop counters of S_1 and S_2 are of value i, $loop_c^{1_i}(n_1) = loop_c^{2_i}(n_2) = i$.*
   - *Value stores σ_{1_i} and σ_{2_i} agree on values of output deciding variables in S_1 and S_2 as well as the input sequence variable, and the I/O sequence variable, $\forall x \in (OVar(S_1) \cup OVar(S_2)) \cup \{id_I, id_{IO}\} : \sigma_{1_i}(x) = \sigma_{2_i}(x)$.*
3. *There are reachable configurations $(S_1, m_1(loop_c^{1_i}, \sigma_{1_i}))$ from $(S_1, m_1(\sigma_1)), (S_2, m_2(loop_c^{2_i}, \sigma_{2_i}))$ from $(S_2, m_2(\sigma_2))$ where all of the following hold:*
   - *The loop counter of S_1 and S_2 are of value i, $loop_c^{1_i}(n_1) = loop_c^{2_i}(n_2) = i$.*
   - *Value stores σ_{1_i} and σ_{2_i} agree on values of output deciding variables in S_1 and S_2 including the input sequence variable and the I/O sequence variable, $\forall x \in (OVar(S_1) \cup OVar(S_2)) \cup \{id_I, id_{IO}\} : \sigma_{1_i}(x) = \sigma_{2_i}(x)$.*

Proof. By induction on i.
Base case.
We show that, when $i = 1$, one of the following holds:

1. Loop counters for S_1 and S_2 are always less than 1 if any is present, $\forall m_1'(\text{loop}_c^{1'})\, m_2'(\text{loop}_c^{2'}) : (S_1, m_1(\text{loop}_c^1, \sigma_1)) \xrightarrow{*} (S_1'', m_1'(\text{loop}_c^{1'})), \text{loop}_c^{1'}(n_1) < i, (S_2, m_2(\text{loop}_c^2, \sigma_2)) \xrightarrow{*} (S_2'', m_2'(\text{loop}_c^{2'})), \text{loop}_c^{2'}(n_2) < i$, S_1 and S_2 terminate in the

same way, produce the same output sequence, and have equivalent computation of output deciding variables in S_1 and S_2 including the input sequence variable, the I/O sequence variable, $(S_1, m_1) \equiv_H (S_2, m_2)$ and $(S_1, m_1) \equiv_O (S_2, m_2)$ and $\forall x \in (\text{OVar}(S_1) \cup \text{OVar}(S_2)) \cup \{id_I, id_{IO}\} : (S_1, m_1) \equiv_x (S_2, m_2)$;

2. Loop counters of S_1 and S_2 are of values less than or equal to 1 but there are no reachable configurations $(S_1, m_1(\text{loop}_c^{1_1}, \sigma_{1_i}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(\text{loop}_c^{2_1}, \sigma_{2_i}))$ from $(S_2,\ m_2(\sigma_2))$ where all of the following hold:
   - The loop counter of S_1 and S_2 are of value 1, $\text{loop}_c^{1_1}(n_1) = \text{loop}_c^{2_1}(n_2) = 1$.
   - Value stores σ_{1_1} and σ_{2_1} agree on values of output deciding variables in S_1 and S_2 including the input sequence variable and the I/O sequence variable, $\forall x \in (\text{OVar}(S_1) \cup \text{OVar}(S_2)) \cup \{id_I, id_{IO}\} : \sigma_{1_1}(x) = \sigma_{2_1}(x)$.
3. There are reachable configuration $(S_1, m_1(\text{loop}_c^{1_1}, \sigma_{1_i}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(\text{loop}_c^{2_1}, \sigma_{2_i}))$ from $(S_2,\ m_2(\sigma_2))$ where all of the following hold:
   - The loop counter of S_1 and S_2 are of value 1, $\text{loop}_c^{1_1}(n_1) = \text{loop}_c^{2_1}(n_2) = 1$.
   - Value stores σ_{1_1} and σ_{2_1} agree on values of output deciding variables in S_1 and S_2 including the input sequence variable and the I/O sequence variable, $\forall x \in (\text{OVar}(S_1) \cup \text{OVar}(S_2)) \cup \{id_I, id_{IO}\} : \sigma_{1_1}(x) = \sigma_{2_1}(x)$.

By definition, variables used in the predicate expression e of S_1 and S_2 are in output deciding variables in S_1 and S_2, $\text{Use}(e) \subseteq \text{OVar}(S_1) \cup \text{OVar}(S_2)$. By assumption, value stores σ_1 and σ_2 agree on values of variables in $\text{Use}(e)$, the predicate expression e evaluates to the same value w.r.t value stores σ_1 and σ_2 by Lemma D.2. There are three possibilities.

1. The evaluation of e crashes,
   $\mathcal{E}'[\![e]\!]\sigma_1 = \mathcal{E}'[\![e]\!]\sigma_2 = (\text{error}, v_{\mathfrak{of}})$.
   The execution of S_1 continues as follows:

   $(\text{while}_{\langle n_1 \rangle}(e)\,\{S_1'\}, m_1(\sigma_1))$
   $\rightarrow(\text{while}_{\langle n_1 \rangle}((\text{error}, v_{\mathfrak{of}}))\,\{S_1'\}, m_1(\sigma_1))$
   by the rule EEval'
   $\rightarrow(\text{while}_{\langle n_1 \rangle}(0)\,\{S_1'\}, m_1(1/\mathfrak{f}))$
   by the ECrash rule
   $\xrightarrow{i}(\text{while}_{\langle n_1 \rangle}(0)\,\{S_1'\}, m_1(1/\mathfrak{f}))$ for any $i > 0$
   by the Crash rule.

   Similarly, the execution of S_2 started from the state $m_2(\sigma_2)$ crashes. Therefore S_1 and S_2 terminate in the same way when started from m_1 and m_2 respectively. Because $\sigma_1(id_{IO}) = \sigma_2(id_{IO})$, the lemma holds.
2. The evaluation of e reduces to zero, $\mathcal{E}'[\![e]\!]\sigma_1 = \mathcal{E}'[\![e]\!]\sigma_2 = (0, v_{\mathfrak{of}})$.
   The execution of S_1 continues as follows.

   $(\text{while}_{\langle n_1 \rangle}(e)\,\{S_1'\}, m_1(\sigma_1))$
   $= (\text{while}_{\langle n_1 \rangle}((0, v_{\mathfrak{of}}))\,\{S_1'\}, m_1(\sigma_1))$
   by the rule EEval'
   $\rightarrow(\text{while}_{\langle n_1 \rangle}(0)\,\{S_1'\}, m_1(\sigma_1))$
   by the E-Oflow1 or E-Oflow2 rule
   $\rightarrow(\text{skip}, m_1(\sigma_1))$ by the Wh-F rule.

   Similarly, the execution of S_2 gets to the configuration $(\text{skip}, m_2(\sigma_2))$. Loop counters of S_1 and S_2 are less than 1 and value stores agree on values of output deciding variables in S_1 and S_2 including the input sequence variable and the I/O sequence variable.
3. The evaluation of e reduces to the same nonzero integer value, $\mathcal{E}'[\![e]\!]\sigma_1 = \mathcal{E}'[\![e]\!]\sigma_2 = (0, v_{\mathfrak{of}})$.
   Then the execution of S_1 proceeds as follows:

   $(\text{while}_{\langle n_1 \rangle}(e)\,\{S_1'\}, m_1(\sigma_1))$
   $= (\text{while}_{\langle n_1 \rangle}((v, v_{\mathfrak{of}}))\,\{S_1'\}, m_1(\sigma_1))$
   by the rule EEval'
   $\rightarrow(\text{while}_{\langle n_1 \rangle}(v)\,\{S_1'\}, m_1(\sigma_1))$
   by the E-Oflow1 or E-Oflow2 rule
   $\rightarrow(S_1'; \text{while}_{\langle n_1 \rangle}(e)\,\{S_1'\}, m_1($
   $\text{loop}_c^1 \cup \{(n_1) \mapsto 1\}, \sigma_1))$ by the Wh-T rule.

   Similarly, the execution of S_2 proceeds to the configuration $(S_2'; \text{while}_{\langle n_2 \rangle}(e)\,\{S_2'\}, m_2(\text{loop}_c^2 \cup \{(n_2) \mapsto 1\}, \sigma_2))$.
   By the assumption, we show that S_1' and S_2' terminate in the same way and produce the same output sequence when started in the state $m_1(\text{loop}_c^{1_1}, \sigma_1)$ and $m_2(\text{loop}_c^{2_1}, \sigma_2)$ respectively, and S_1' and S_2' have equivalent computation of variables defined in both statement sequences if both terminate. We need to show that all conditions are satisfied for the application of the assumption.
   - There are no program semantic errors related to the extra check in the update of exit-on-error in executions of S_2' and S_1'.
     The above two conditions are by assumption.
   - Value stores σ_1 and σ_2 agree on values of output deciding variables in S_1' and S_2' including the input, I/O sequence variable.
     By definition, $\text{OVar}(S_1') \subseteq \text{OVar}(S_1)$. So are the cases to S_2' and S_2. In addition, value stores σ_1 and σ_2 are not changed in the evaluation of the predicate expression e. The condition holds.

   By assumption, S_1' and S_2' terminate in the same way and produce the same output sequence when started in states $m_1(\text{loop}_c', \sigma_1)$ and $m_2(\text{loop}_c', \sigma_2)$. In addition, S_1' and S_2' have equivalent computation of output deciding variables in S_1' and S_2' when started in states $m_1(\text{loop}_c', \sigma_1)$ and $m_2(\text{loop}_c', \sigma_2)$.
   Then there are two cases.
   (a) S_1' and S_2' both do not terminate and produce the same output sequence.
   By Lemma E.2, $S_1'; S_1$ and $S_2'; S_2$ both do not terminate and produce the same output sequence.
   (b) S_1' and S_2' both terminate and have equivalent computation of output deciding variables in S_1' and S_2'.
   By assumption, $(S_1', m_1(\text{loop}_c', \sigma_1)) \xrightarrow{*} (\text{skip},\ m_1'(\text{loop}_c'', \sigma_1'))$; $(S_2', m_2(\text{loop}_c', \sigma_2)) \xrightarrow{*} (\text{skip}, m_2'(\text{loop}_c'', \sigma_2'))$ where $\forall x \in (\text{OVar}(S_1') \cup \text{OVar}(S_2')) \cup \{id_I, id_{IO}\}, \sigma_1'(x) = \sigma_2'(x)$.
   By Lemma 6.9, $\text{OVar}(S_1') \subseteq \text{OVar}(S_2')$. Then variables used in the predicate expression of S_1 and S_2 are either in output deciding variables in both S_1' and S_2' or not. Therefore value stores σ_2' and σ_1' agree on values of variables used in the expression e and even output deciding variables in S_1 and S_2.

Induction step on iterations

The induction hypothesis (IH) is that, when $i \geq 1$, one of the following holds:

1. Loop counters for S_1 and S_2 are always less than i if any is present, $\forall m_1'(\text{loop}_c^{1'})\, m_2'(\text{loop}_c^{2'}) : (S_1, m_1(\text{loop}_c^1, \sigma_1)) \xrightarrow{*} (S_1'', m_1'(\text{loop}_c^{1'})), \text{loop}_c^{1'}(n_1) < i, (S_2, m_2(\text{loop}_c^2, \sigma_2)) \xrightarrow{*} (S_2'', m_2'(\text{loop}_c^{2'})), \text{loop}_c^{2'}(n_2) < i$, S_1 and S_2 terminate in the same way, produce the same output sequence, and have equivalent computation of output deciding variables in both S_1 and S_2

as well as the input sequence variable, the I/O sequence variable, $(S_1, m_1) \equiv_H (S_2, m_2)$ and $(S_1, m_1) \equiv_O (S_2, m_2)$ and $\forall x \in (\text{OVar}(S_1) \cup \text{OVar}(S_2)) \cup \{id_I, id_{IO}\} : (S_1, m_1) \equiv_x (S_2, m_2)$;

2. The loop counter of S_1 and S_2 are of value less than or equal to i, and there are no reachable configurations $(S_1, m_1(\text{loop}_c^{1i}, \sigma_{1_i}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(\text{loop}_c^{2i}, \sigma_{2_i}))$ from $(S_2,\ m_2(\sigma_2))$ where all of the followings hold:
   - The loop counters of S_1 and S_2 are of value i, $\text{loop}_c^{1i}(n_1) = \text{loop}_c^{2i}(n_2) = i$.
   - Value stores σ_{1_i} and σ_{2_i} agree on values of output deciding variables in S_1 and S_2 including the input sequence variable, and the I/O sequence variable, $\forall x \in (\text{OVar}(S_1) \cup \text{OVar}(S_2)) \cup \{id_I, id_{IO}\} : \sigma_{1_i}(x) = \sigma_{2_i}(x)$.
3. There are reachable configurations $(S_1, m_1(\text{loop}_c^{1i}, \sigma_{1_i}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(\text{loop}_c^{2i}, \sigma_{2_i}))$ from $(S_2,\ m_2(\sigma_2))$ where all of the following hold:
   - The loop counter of S_1 and S_2 are of value i, $\text{loop}_c^{1i}(n_1) = \text{loop}_c^{2i}(n_2) = i$.
   - Value stores σ_{1_i} and σ_{2_i} agree on values of output deciding variables in S_1 and S_2 including the input sequence variable, and the I/O sequence variable, $\forall x \in (\text{OVar}(S_1) \cup \text{OVar}(S_2)) \cup \{id_I, id_{IO}\} : \sigma_{1_i}(x) = \sigma_{2_i}(x))$.

Then we show that, when $i + 1$, one of the following holds:

1. Loop counters for S_1 and S_2 are always less than $i + 1$ if any is present, $\forall m_1'(\text{loop}_c^{1'})\, m_2'(\text{loop}_c^{2'}) : (S_1, m_1(\text{loop}_c^1, \sigma_1)) \xrightarrow{*} (S_1'', m_1'(\text{loop}_c^{1'}))$, $\text{loop}_c^{1'}(n_1) < i{+}1$, $(S_2, m_2(\text{loop}_c^2, \sigma_2)) \xrightarrow{*} (S_2'', m_2'(\text{loop}_c^{2'}))$, $\text{loop}_c^{2'}(n_2) < i + 1$, S_1 and S_2 terminate in the same way, produce the same output sequence, and have equivalent computation of output deciding variables in S_1 and S_2 including the input sequence variable and the I/O sequence variable, $(S_1, m_1) \equiv_H (S_2, m_2)$ and $(S_1, m_1) \equiv_O (S_2, m_2)$ and $\forall x \in (\text{OVar}(S_1) \cup \text{OVar}(S_2)) \cup \{id_I, id_{IO}\} : (S_1, m_1) \equiv_x (S_2, m_2)$;
2. The loop counter of S_1 and S_2 are of value less than or equal to $i + 1$, and there are no reachable configurations $(S_1, m_1(\text{loop}_c^{1i+1}, \sigma_{1_{i+1}}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(\text{loop}_c^{2i+1}, \sigma_{2_{i+1}}))$ from $(S_2,\ m_2(\sigma_2))$ where all of the following hold:
   - The loop counters of S_1 and S_2 are of value $i + 1$, $\text{loop}_c^{1i+1}(n_1) = \text{loop}_c^{2i+1}(n_2) = i + 1$.
   - Value stores $\sigma_{1_{i+1}}$ and $\sigma_{2_{i+1}}$ agree on values of output deciding variables in S_1 and S_2 including the input sequence variable, and the I/O sequence variable, $\forall x \in (\text{OVar}(S_1) \cup \text{OVar}(S_2)) \cup \{id_I, id_{IO}\} : \sigma_{1_{i+1}}(x) = \sigma_{2_{i+1}}(x)$.
3. There are reachable configurations $(S_1, m_1(\text{loop}_c^{1i+1}, \sigma_{1_i}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(\text{loop}_c^{2i+1}, \sigma_{2_i}))$ from $(S_2,\ m_2(\sigma_2))$ where all of the following hold:
   - The loop counter of S_1 and S_2 are of value i, $\text{loop}_c^{1i+1}(n_1) = \text{loop}_c^{2i+1}(n_2) = i + 1$.
   - Value stores $\sigma_{1_{i+1}}$ and $\sigma_{2_{i+1}}$ agree on values of output deciding variables in S_1 and S_2 including the input sequence variable, and the I/O sequence variable, $\forall x \in (\text{OVar}(S_1) \cup \text{OVar}(S_2)) \cup \{id_I, id_{IO}\} : \sigma_{1_{i+1}}(x) = \sigma_{2_{i+1}}(x)$.

By hypothesis IH and theorem 4 and 5, there is no configuration where loop counters of S_1 and S_2 are of value $i + 1$ when any of the following holds:

1. Loop counters for S_1 and S_2 are always less than i if any is present, $\forall m_1'(\text{loop}_c^{1'})\, m_2'(\text{loop}_c^{2'}) : (S_1, m_1(\text{loop}_c^1, \sigma_1)) \xrightarrow{*} (S_1'', m_1'(\text{loop}_c^{1'}))$, $\text{loop}_c^{1'}(n_1) < i$, $(S_2, m_2(\text{loop}_c^2, \sigma_2)) \xrightarrow{*} (S_2'', m_2'(\text{loop}_c^{2'}))$, $\text{loop}_c^{2'}(n_2) < i$, S_1 and S_2 terminate in the same way, produce the same output sequence, and have equivalent computation of output deciding variables in S_1 and S_2 including the input sequence variable and the I/O sequence variable, $(S_1, m_1) \equiv_H (S_2, m_2)$ and $(S_1, m_1) \equiv_O (S_2, m_2)$ and $\forall x \in (\text{OVar}(S_1) \cup \text{OVar}(S_2)) \cup \{id_I, id_{IO}\} : (S_1, m_1) \equiv_x (S_2, m_2)$;
2. The loop counter of S_1 and S_2 are of value less than or equal to i, and there are no reachable configurations $(S_1, m_1(\text{loop}_c^{1i}, \sigma_{1_i}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(\text{loop}_c^{2i}, \sigma_{2_i}))$ from $(S_2,\ m_2(\sigma_2))$ where all of the following hold:
   - The loop counters of S_1 and S_2 are of value i, $\text{loop}_c^{1i}(n_1) = \text{loop}_c^{2i}(n_2) = i$.
   - Value stores σ_{1_i} and σ_{2_i} agree on values of output deciding variables in both S_1 and S_2 as well as the input sequence variable, and the I/O sequence variable, $\forall x \in (\text{OVar}(S_1) \cup \text{OVar}(S_2)) \cup \{id_I, id_{IO}\} : \sigma_{1_i}(x) = \sigma_{2_i}(x)$.

When there are reachable configurations $(S_1, m_1(\text{loop}_c^{1i}, \sigma_{1_i}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(\text{loop}_c^{2i}, \sigma_{2_i}))$ from $(S_2,\ m_2(\sigma_2))$ where all of the following hold:

- The loop counter of S_1 and S_2 are of value i, $\text{loop}_c^{1i}(n_1) = \text{loop}_c^{2i}(n_2) = i$.
- Value stores σ_{1_i} and σ_{2_i} agree on values of output deciding variables in S_1 and S_2 including the input sequence variable and the I/O sequence variable, $\forall x \in (\text{OVar}(S_1) \cup \text{OVar}(S_2)) \cup \{id_I, id_{IO}\} : \sigma_{1_i}(x) = \sigma_{2_i}(x)$.

By similar argument in base case, we have one of the following holds:

1. Loop counters for S_1 and S_2 are always less than $i + 1$ if any is present, $\forall m_1'(\text{loop}_c^{1'})\, m_2'(\text{loop}_c^{2'}) : (S_1, m_1(\text{loop}_c^1, \sigma_1)) \xrightarrow{*} (S_1'', m_1'(\text{loop}_c^{1'}))$, $\text{loop}_c^{1'}(n_1) < i{+}1$, $(S_2, m_2(\text{loop}_c^2, \sigma_2)) \xrightarrow{*} (S_2'', m_2'(\text{loop}_c^{2'}))$, $\text{loop}_c^{2'}(n_2) < i$, S_1 and S_2 terminate in the same way, produce the same output sequence, and have equivalent computation of output deciding variables in S_1 and S_2 including the input sequence variable, the I/O sequence variable, $(S_1, m_1) \equiv_H (S_2, m_2)$ and $(S_1, m_1) \equiv_O (S_2, m_2)$ and $\forall x \in (\text{OVar}(S_1) \cup \text{OVar}(S_2)) \cup \{id_I, id_{IO}\} : (S_1, m_1) \equiv_x (S_2, m_2)$;
2. The loop counter of S_1 and S_2 are of value less than or equal to $i + 1$, and there are no reachable configurations $(S_1, m_1(\text{loop}_c^{1i}, \sigma_{1_i}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(\text{loop}_c^{2i}, \sigma_{2_i}))$ from $(S_2,\ m_2(\sigma_2))$ where all of the following hold:
   - The loop counters of S_1 and S_2 are of value i, $\text{loop}_c^{1i+1}(n_1) = \text{loop}_c^{2i+1}(n_2) = i + 1$.
   - Value stores $\sigma_{1_{i+1}}$ and $\sigma_{2_{i+1}}$ agree on values of output deciding variables in S_1 and S_2 including the input sequence variable and the I/O sequence variable, $\forall x \in (\text{OVar}(S_1) \cup \text{OVar}(S_2)) \cup \{id_I, id_{IO}\} : \sigma_{1_{i+1}}(x) = \sigma_{2_{i+1}}(x)$.
3. There are reachable configurations $(S_1, m_1(\text{loop}_c^{1i+1}, \sigma_{1_{i+1}}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(\text{loop}_c^{2i+1}, \sigma_{2_{i+1}}))$ from $(S_2,\ m_2(\sigma_2))$ where all of the following hold:
   - The loop counter of S_1 and S_2 are of value i, $\text{loop}_c^{1i+1}(n_1) = \text{loop}_c^{2i+1}(n_2) = i$.
   - Value stores $\sigma_{1_{i+1}}$ and $\sigma_{2_{i+1}}$ agree on values of output deciding variables in S_1 and S_2 including the input sequence variable, and the I/O sequence variable, $\forall x \in (\text{OVar}(S_1) \cup \text{OVar}(S_2)) \cup \{id_{I+1}, id_{IO}\} : \sigma_{1_{i+1}}(x) = \sigma_{2_{i+1}}(x)$.

$\square$

6.5 Proof rule for improved prompt message

If the only difference between two programs are the constant messages that the user receives, we consider that the two programs to be equivalent. We realize that in general it is possible to introduce new semantics even by changing constant strings. An old version might have incorrectly labeled output: "median value = 5" instead of "average value = 5, for example. We rule out such possibilities because all non-constant values are guaranteed to be exactly same. In practice, outputs could be classified into prompt outputs and actual outputs. Prompt outputs are those asking clients for inputs, which are constants hardcoded in the output statement. Actual outputs are dynamic messages produced by evaluation of non-constant expression in execution. The changes of prompt outputs are equivalent only for interactions with human clients. In order to prove the update of improved prompt messages to be backward compatible, we assume that the different prompt outputs produced in executions of the old program and the updated program, due to the different constants in output statements, are equivalent. Because the old program and the new program are exactly same except some output statements with different constants as expression e, we could show two programs produce the "equivalent" output sequence under the assumption of equivalent prompt outputs.

We formalize the generalized update of improved prompt messages, then we show that the updated program produce the same I/O sequence as the old program in executions without program semantic errors. The following is the definition of the update class of improved prompt messages.

Definition 31. (Improved user messages) *A program $P_2 = Pmpt_2; EN; V; S_{entry}$ includes updates of improved prompt messages compared with a program $P_1 = Pmpt_1; EN; V; S_{entry}$, written $P_2 \approx^S_{Out} P_1$, iff $Pmpt_2 \neq Pmpt_1$.*

We give the lemma that two programs terminate in the same way, produce the equivalent output sequence, and have equivalent computation of defined variables in both programs in valid executions if there are updates of improved prompt messages between them.

Lemma 6.11. *Let $P_1 = Pmpt_1; EN; V; S_{entry}$ and $P_2 = Pmpt_2; EN; V; S_{entry}$ be two programs where there are updates of improved prompt messages in P_2 compared with P_1. If S_1 and S_2 start in states $m_1(\sigma_1)$ and $m_2(\sigma_2)$ such that both of the following hold:*

- *Value stores σ_1 and σ_2 agree on values of variables used in S_{entry} in both programs as well as the input sequence variable, $\forall x \in Use(S_{entry}) \cup \{id_I\} : \sigma_1(x) = \sigma_2(x)$;*
- *Value stores σ_1 and σ_2 have "equivalent" I/O sequence, $\sigma_1(id_{IO}) \equiv \sigma_2(id_{IO})$;*
- *The different prompt outputs in the update of improved prompt messages are equivalent;*

then S_1 and S_2 terminate in the same way, produce the equivalent output sequence, and when S_1 and S_2 both terminate, they have equivalent computation of defined variables in S_{entry} in both programs as well as the input sequence variable, S_{entry} in the two programs produce the equivalent I/O sequence variable,

- *$(S_{entry}, m_1) \equiv_H (S_{entry}, m_2)$;*
- *$\forall x \in (Def(S_1) \cap Def(S_2)) \cup \{id_I\} : (S_{entry}, m_1) \equiv_x (S_{entry}, m_2)$;*
- *The produced output sequences in executions of S_{entry} in both programs are "equivalent", $\sigma_1(id_{IO}) \equiv \sigma_2(id_{IO})$.*

The difference between prompt types in P_1 and P_2 can be either addition/removal of labels as well as the change of the mapping of labels with constants. The proof is straightforward because programs P_1 and P_2 have the same entry statement sequence and we have the assumption that different prompt outputs due to the difference of the prompt type are equivalent.

Proof. By induction on the sum of the program size of S_1 and S_2, $\text{size}(S_1) + \text{size}(S_2)$.
Base case. $S_1 =$ "output v_1" and $S_2 =$ "output v_2";
Then the execution of S_2 proceeds as follows.

$$\begin{aligned}&(\text{output}\, v_2, m_2(\sigma_2))\\ \rightarrow&(\text{skip}, m_2(\sigma_2[\text{``}\sigma_2(id_{IO}) \cdot \bar{v_2}\text{''}/id_{IO}]))\\ &\text{by the rule Out-1 or Out-2}\end{aligned}$$

Similarly, $(\text{output}\, v_1, m_1(\sigma_1)) \xrightarrow{*} (\text{skip}, m_1(\sigma_1[\text{``}\sigma_1(id_{IO}) \cdot \bar{v_1}\text{''}/id_{IO}]))$.
By assumption, $\sigma_2(id_{IO}) \equiv \sigma_1(id_{IO})$. In addition, by assumption, $\bar{v_2} \equiv \bar{v_1}$. Therefore, S_1 and S_2 terminate in the same way, produce the same output sequence and have equivalent computation of defined variables in S_1 and S_2. This lemma holds.
Induction step.

The hypothesis is that this lemma holds when the sum k of the program size of S_1 and S_2 are great than or equal to 2, $k \geq 2$.

We then show that this lemma holds when the sum of the program size of S_1 and S_2 is $k + 1$. There are cases to consider.

1. S_1 and S_2 are both "If" statement:
   $S_1 =$ "If(e) then$\{S_1^t\}$ else$\{S_1^f\}$", $S_2 =$ "If(e) then$\{S_2^t\}$ else$\{S_2^f\}$"
   where both of the following hold
   - $S_2^t \approx^S_{\text{Out}} S_1^t$;
   - $S_2^f \approx^S_{\text{Out}} S_1^f$;

   By the definition of Use(S_1), variables used in the predicate expression e are a subset of used variables in S_1 and S_2, Use$(e) \subseteq$ Use$(S_1) \cap$ Use(S_2). By assumption, corresponding variables used in e are of same value in value stores σ_1 and σ_2. By Lemma D.1, the expression evaluates to the same value w.r.t value stores (σ_1 and σ_2. There are three possibilities.
   (a) The evaluation of e crashes, $\mathcal{E}'[\![e]\!]\sigma_1 = \mathcal{E}'[\![e]\!]\sigma_2 = (\text{error}, v_{\mathfrak{of}})$.
   The execution of S_1 continues as follows:
   $$\begin{aligned}&(\text{If}(e)\,\text{then}\{S_1^t\}\,\text{else}\{S_1^f\}, m_1(\sigma_1))\\ \rightarrow&(\text{If}((\text{error}, v_{\mathfrak{of}}))\,\text{then}\{S_1^t\}\,\text{else}\{S_1^f\}, m_1(\sigma_1))\\ &\text{by the rule EEval'}\\ \rightarrow&(\text{If}(0)\,\text{then}\{S_1^t\}\,\text{else}\{S_1^f\}, m_1(1/\mathfrak{f}))\\ &\text{by the ECrash rule}\\ \xrightarrow{i}&(\text{If}(0)\,\text{then}\{S_1^t\}\,\text{else}\{S_1^f\}, m_1(1/\mathfrak{f}))\text{ for any } i > 0\\ &\text{by the Crash rule.}\end{aligned}$$
   Similarly, the execution of S_2 started from the state $m_2(\sigma_2)$ crashes. The lemma holds.
   (b) The evaluation of e reduces to zero, $\mathcal{E}'[\![e]\!]\sigma_1 = \mathcal{E}'[\![e]\!]\sigma_2 = (0, v_{\mathfrak{of}})$.
   The execution of S_1 continues as follows.
   $$\begin{aligned}&(\text{If}(e)\,\text{then}\{S_1^t\}\,\text{else}\{S_1^f\}, m_1(\sigma_1))\\ =\ &(\text{If}((0, v_{\mathfrak{of}}))\,\text{then}\{S_1^t\}\,\text{else}\{S_1^f\}, m_1(\sigma_1))\\ &\text{by the rule EEval'}\\ \rightarrow&(\text{If}(0)\,\text{then}\{S_1^t\}\,\text{else}\{S_1^f\}, m_1(\sigma_1))\\ &\text{by the E-Oflow1 or E-Oflow2 rule}\\ \rightarrow&(S_1^f, m_1(\sigma_1))\text{ by the If-F rule.}\end{aligned}$$
   Similarly, the execution of S_2 gets to the configuration $(S_2^f, m_2(\sigma_2))$.
   By the hypothesis IH, we show the lemma holds. We need to show that all conditions are satisfied for the application of the hypothesis IH.
   - $S_2^f \approx^S_{\text{Out}} S_1^f$
     By assumption.

- The sum of the program size of S_1^f and S_2^f is less than k, size(S_1^f) + size$(S_2^f) < k$.
  By definition, size$(S_1) = 1+$size$(S_1^t)+$size(S_1^f). Then, size(S_1^f) + size$(S_2^f) < k + 1 - 2 = k - 1$.
- Value stores σ_1 and σ_2 agree on values of used variables in S_1^f and S_2^f as well as the input, I/O sequence variable.
  By definition, Use$(S_1^f) \subseteq$ Use(S_1). So are the cases to S_2^f and S_2. In addition, value stores σ_1 and σ_2 are not changed in the evaluation of the predicate expression e. The condition holds.
- Different constants used in output statements are equivalent as output values.
  By assumption.

By the hypothesis IH, the lemma holds.

(c) The evaluation of e reduces to the same nonzero integer value, $\mathcal{E}'[\![e]\!]\sigma_1 = \mathcal{E}'[\![e]\!]\sigma_2 = (v, v_{\text{of}})$ where $v \neq 0$.
By argument similar to the second subcase above.

2. S_1 and S_2 are both "while" statements:
S_1 = "while$_{\langle n\rangle}(e)\,\{S_1'\}$", S_2 = "while$_{\langle n\rangle}(e)\,\{S_2'\}$" where $S_2' \approx^S_{\text{Out}} S_1'$;
By Lemma 6.13, we show this lemma holds. We need to show that all required conditions are satisfied for the application of Lemma 6.13.
   - S_1' and S_2' have same set of defined variables, Def(S_1') = Def(S_2') = Def(S);
   - The used variables in S_1' are a subset of those in S_2', Use(S_1') = Use(S_2');
     By Lemma 6.12.
   - When started in states $m_1'(\sigma_1'), m_2'(\sigma_1')$ where value stores σ_1' and σ_2' agree on values of used variables in both S_1' and S_2' as well as the input sequence variable, , and the I/O sequence variable, then S_1' and S_2' terminate in the same way, produce the same output sequence, and have equivalent computation of defined variables in both S_1 and S_2 as well as the input sequence variable and the I/O sequence variable.
     By the induction hypothesis IH. This is because the sum of the program size of S_1' and S_2' is less than k. By definition, size$(S_1) = 1 +$ size(S_1').

   By Lemma 6.13, this lemma holds.
3. $S_1 = S_1'; s_1$ and $S_2 = S_2'; s_2$ where both of the following hold:
   - $S_2' \approx^S_{\text{Out}} S_1'$;
   - $s_2 \approx^S_{\text{Out}} s_1$;

   By the hypothesis IH, we show S_2' and S_1' terminate in the same way and produce the equivalent output sequence and when S_2' and S_1' both terminate, S_2' and S_1' have equivalent terminating computation of variables defined in S_2' and S_1' as well as the input sequence variable. By assumption, the different value of the I/O sequence in executions of S_1 and S_2 are equivalent.
   We show all the required conditions are satisfied for the application of the hypothesis IH.
   - $S_2' \approx^S_{\text{Out}} S_1'$;
   - The I/O sequence variable in executions of S_1 and S_2 are equivalent, $\sigma_1(id_{IO}) \equiv \sigma_2(id_{IO})$;
     By assumption.
   - The sum of the program size of S_1' and S_2' is less than k, size(S_1') + size$(S_2') < k$.
     By definition, size(S_2) = size(s_2) + size(S_2') where size$(s_2) < 1$. Then, size(S_2') + size$(S_1') < k + 1 -$ size$(s_2) -$ size$(s_1) < k$.
   - Value stores σ_1 and σ_2 agree on values of used variables in both S_2' and S_1' as well as the input sequence variable.
     By definition, Use$(S_2') \subseteq$ Use(S_2), Use$(S_1') \subseteq$ Use(S_1). The condition holds.

   By the hypothesis IH, one of the following holds:
   (a) S_1' and S_2' both do not terminate.
   By Lemma E.2, executions of $S_1 = S_1'; s_1$ and $S_2 = S_2'; s_2$ both do not terminate and produce the same output sequence.
   (b) S_1' and S_2' both terminate.
   By assumption, $(S_2', m_2(\sigma_2)) \xrightarrow{*} (\text{skip}, m_2'(\sigma_2'))$, $(S_1', m_1(\sigma_1)) \xrightarrow{*} (\text{skip}, m_1'(\sigma_1'))$.
   By Corollary E.1, $(S_2'; s_2, m_2(\sigma_2)) \xrightarrow{*} (s_2, m_2'(\sigma_2'))$, $(S_1'; s_1, m_1(\sigma_1)) \xrightarrow{*} (s_1, m_1'(\sigma_1'))$.
   By the hypothesis IH, we show that s_2 and s_1 terminate in the same way, produce the "equivalent" output sequence and when s_2 and s_1 both terminate, s_2 and s_1 have equivalent computation of variables defined in both s_1 and s_2 and the input sequence variable; s_2 and s_1 produce "equivalent" output sequence.
   We need to show that all conditions are satisfied for the application of the hypothesis IH.
   - There are updates of "improved prompt messages" in s_2 compared with s_1, $s_2 \approx^S_{\text{Out}} s_1$;
     By assumption, $s_2 \approx^S_{\text{Out}} s_1$.
   - The sum of the program size s_2 and s_1 is less than or equals to k;
     By definition, size$(S_2') \geq 1$, size$(S_1') \geq 1$. Therefore, size(s_2) + size$(s_1) < k + 1 -$ size$(S_2') -$ size$(S_1') \leq k$.
   - Value stores σ_1' and σ_2' agree on values of used variables in s_2 and s_1 as well as the input sequence variable;
     By Lemma 6.9, Use(s_1) = Use(s_2), then Use(s_2) = Use(s_1) = Use(s). Similarly, by Lemma 6.9, Def(S_1') = Def(S_2'). For any variable id in Use(s_1), if id is not in Def(S_1'), then the value of id is not changed in the execution of S_1' and S_2', $\sigma_1'(id) = \sigma_1(id) = \sigma_2(id) = \sigma_2'(id)$. Otherwise, the variable id is defined in the execution of S_1' and S_2', by assumption, $\sigma_1'(id) = \sigma_2'(id)$. The condition holds.
   - Values of , the I/O sequence variable in value stores σ_1' and σ_2' are equivalent.
     By assumption.

   By the hypothesis IH, the lemma holds.

□

We list the auxiliary lemmas below. One lemma shows that, if there are updates of improved prompt messages between two statement sequences, then there are same set of defined variables and used variables in the two statement sequences. The second lemma shows that, if there are updates of improved prompt messages between two loop statements, then the two loop statement terminate in the same way, produce the equivalent output sequence, and have equivalent computation of defined variables in both the old and updated programs as well as the input sequence variable.

Lemma 6.12. *Let S_2 be a statement sequence and S_1 where there are updates of "improved prompt messages", $S_2 \approx^S_{Out} S_1$. Then used variables in S_2 are the same of used variables in S_1, Use(S_1) = Use(S_2), defined variables in S_2 are the same as used variables in S_1, Def(S_1) = Def(S_2).*

Proof. By induction on the sum of the program size of S_1 and S_2. □

Lemma 6.13. *Let S_1 = while$_{\langle n_1\rangle}(e)\,\{S_1'\}$ and S_2 = while$_{\langle n_2\rangle}(e)$ $\{S_2'\}$ be two loop statements where all of the following hold:*

- *There are updates of improved prompt messages in S_2' compared with S_1', $S_2' \approx^{S}_{Out} S_1'$;*
- *S_1' and S_2' have same set of defined variables, $Def(S_1') = Def(S_2') = Def(S)$;*
- *S_1' and S_2' have same set of used variables, $Use(S_1') = Use(S_2')$;*
- *When started in states $m_1'(\sigma_1'), m_2'(\sigma_2')$ where*
  - *Value stores agree on values of used variables in both S_1' and S_2' as well as the input sequence variable, $\forall x \in Use(S_1') \cup \{id_I\} \, \forall m_1'(\sigma_1') \, m_2'(\sigma_2') \, : \, \sigma_1'(x) = \sigma_2'(x)$;*
  - *Values of the I/O sequence variable in value stores σ_1', σ_2' are equivalent, $\sigma_1'(id_{IO}) \equiv \sigma_2'(id_{IO})$);*

  then S_1' and S_2' terminate in the same way, produce the "equivalent" output sequence, and have equivalent computation of defined variables in S_1' and S_2' as well as the input sequence variable, $((S_1', m_1) \equiv_H (S_2', m_2)) \wedge (\forall x \in Def(S) \cup \{id_I\} : (S_1', m_1) \equiv_x (S_2', m_2))$;

If S_1 and S_2 start in states $m_1(loop_c^1, \sigma_1), m_2(loop_c^2, \sigma_2)$ respectively, with loop counters of S_1 and S_2 not initialized (S_1, S_2 have not executed yet), value stores agree on values of used variables in S_1 and S_2, and there are no program semantic errors, then, for any positive integer i, one of the following holds:

1. *Loop counters for S_1 and S_2 are always less than i if any is present, $\forall m_1'(loop_c^{1'}) \, m_2'(loop_c^{2'}) \, : \, (S_1, m_1(loop_c^1, \sigma_1)) \xrightarrow{*} (S_1'', m_1'(loop_c^{1'})), loop_c^{1'}(n_1) < i, (S_2, m_2(loop_c^2, \sigma_2)) \xrightarrow{*} (S_2'', m_2'(loop_c^{2'})), loop_c^{2'}(n_2) < i$, S_1 and S_2 terminate in the same way, produce the equivalent output sequence, and have equivalent computation of defined variables in both S_1 and S_2 and the input sequence variable, $(S_1, m_1) \equiv_H (S_2, m_2)$ and $\forall x \in (Def(S_1) \cap Def(S_2)) \cup \{id_I\} \, : \, (S_1, m_1) \equiv_x (S_2, m_2)$;$S_1$ and S_2 produce the "equivalent" I/O sequence;*
2. *The loop counter of S_1 and S_2 are of value less than or equal to i, and there are no reachable configurations $(S_1, m_1(loop_c^{1_i}, \sigma_{1_i}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(loop_c^{2_i}, \sigma_{2_i}))$ from $(S_2, m_2(\sigma_2))$ where all of the following hold:*
   - *The loop counters of S_1 and S_2 are of value i, $loop_c^{1_i}(n_1) = loop_c^{2_i}(n_2) = i$.*
   - *Value stores σ_{1_i} and σ_{2_i} agree on values of used variables in both S_1 and S_2 as well as the input sequence variable, $\forall x \in (Use(S_1) \cap Use(S_2)) \cup \{id_I\} \, : \, \sigma_{1_i}(x) = \sigma_{2_i}(x)$.*
   - *Values of the I/O sequence variable in value stores $\sigma_{1_i}(id_{IO}) \equiv \sigma_{2_i}(id_{IO})$;*
3. *There are reachable configurations $(S_1, m_1(loop_c^{1_i}, \sigma_{1_i}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(loop_c^{2_i}, \sigma_{2_i}))$ from $(S_2, m_2(\sigma_2))$ where all of the following hold:*
   - *The loop counter of S_1 and S_2 are of value i, $loop_c^{1_i}(n_1) = loop_c^{2_i}(n_2) = i$.*
   - *Value stores σ_{1_i} and σ_{2_i} agree on values of used variables in both S_1 and S_2 as well as the input sequence variable, $\forall x \in (Use(S_1) \cap Use(S_2)) \cup \{id_I\} \, : \, \sigma_{1_i}(x) = \sigma_{2_i}(x)$.*
   - *Values of the I/O sequence variable in value stores $\sigma_{1_i}, \sigma_{2_i}$ are equivalent, $\sigma_{1_i}(id_{IO}) \equiv \sigma_{2_i}(id_{IO})$;*

Proof. By induction on i.
Base case.

We show that, when $i = 1$, one of the followings holds:

1. Loop counters for S_1 and S_2 are always less than 1 if any is present, $\forall m_1'(\text{loop}_c^{1'}) \, m_2'(\text{loop}_c^{2'}) \, : \, (S_1, m_1(\text{loop}_c^1, \sigma_1)) \xrightarrow{*} (S_1'', m_1'(\text{loop}_c^{1'})), \text{loop}_c^{1'}(n_1) < i, (S_2, m_2(\text{loop}_c^2, \sigma_2)) \xrightarrow{*} (S_2'', m_2'(\text{loop}_c^{2'})), \text{loop}_c^{2'}(n_2) < i$, S_1 and S_2 terminate in the same way, produce the equivalent I/O sequence, and have equivalent computation of defined variables in both S_1 and S_2 and the input sequence variable, the I/O sequence variable, $(S_1, m_1) \equiv_H (S_2, m_2)$ and $\forall x \in \text{Def}(S) \cup \{id_I\} \, : \, (S_1, m_1) \equiv_x (S_2, m_2)$;
2. S_1 and S_2 produce the equivalent output sequence and the equivalent I/O sequence;
3. Loop counters of S_1 and S_2 are of values less than or equal to 1 but there are no reachable configurations $(S_1, m_1(\text{loop}_c^{1_1}, \sigma_{1_i}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(\text{loop}_c^{2_1}, \sigma_{2_i}))$ from $(S_2, m_2(\sigma_2))$ where all of the following hold:
   - The loop counter of S_1 and S_2 are of value 1, $\text{loop}_c^{1_1}(n_1) = \text{loop}_c^{2_1}(n_2) = 1$.
   - Value stores σ_{1_1} and σ_{2_1} agree on values of used variables in both S_1 and S_2 as well as the input sequence variable, and the I/O sequence variable, $\forall x \in (\text{Use}(S_1) \cap \text{Use}(S_2)) \cup \{id_I\} \, : \, \sigma_{1_1}(x) = \sigma_{2_1}(x)$.
   - Values of the I/O sequence variable in value stores σ_{1_1} and σ_{2_1} are equivalent, $\sigma_{1_1}(id_{IO}) \equiv \sigma_{2_1}(id_{IO})$;
4. There are reachable configuration $(S_1, m_1(\text{loop}_c^{1_1}, \sigma_{1_i}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(\text{loop}_c^{2_1}, \sigma_{2_i}))$ from $(S_2, m_2(\sigma_2))$ where all of the following hold:
   - The loop counter of S_1 and S_2 are of value 1, $\text{loop}_c^{1_1}(n_1) = \text{loop}_c^{2_1}(n_2) = 1$.
   - Value stores σ_{1_1} and σ_{2_1} agree on values of used variables in both S_1 and S_2 as well as the input sequence variable, and the I/O sequence variable, $\forall x \in (\text{Use}(S_1) \cap \text{Use}(S_2)) \cup \{id_I\} \, : \, \sigma_{1_1}(x) = \sigma_{2_1}(x)$.
   - Values of the I/O sequence variable in value stores σ_{1_1} and σ_{2_1} are equivalent, $\sigma_{1_1}(id_{IO}) \equiv \sigma_{2_1}(id_{IO})$;

By definition, variables used in the predicate expression e of S_1 and S_2 are used in S_1 and S_2, $\text{Use}(e) \subseteq \text{Use}(S_1) \cap \text{Use}(S_2)$. By assumption, value stores σ_1 and σ_2 agree on values of variables in $\text{Use}(e)$, the predicate expression e evaluates to the same value w.r.t value stores σ_1 and σ_2 by Lemma D.2. There are three possibilities.

1. The evaluation of e crashes,
   $\mathcal{E}'[\![e]\!]\sigma_1 = \mathcal{E}'[\![e]\!]\sigma_2 = (\text{error}, v_{\mathfrak{of}})$.
   The execution of S_1 continues as follows:

   $(\text{while}_{\langle n_1 \rangle}(e) \, \{S_1'\}, m_1(\sigma_1))$
   $\rightarrow (\text{while}_{\langle n_1 \rangle}((\text{error}, v_{\mathfrak{of}})) \, \{S_1'\}, m_1(\sigma_1))$
   by the rule EEval'
   $\rightarrow (\text{while}_{\langle n_1 \rangle}(0) \, \{S_1'\}, m_1(1/\mathfrak{f}))$
   by the ECrash rule
   $\xrightarrow{i} (\text{while}_{\langle n_1 \rangle}(0) \, \{S_1'\}, m_1(1/\mathfrak{f}))$ for any $i > 0$
   by the Crash rule.

   Similarly, the execution of S_2 started from the state $m_2(\sigma_2)$ crashes. Therefore S_1 and S_2 terminate in the same way when started from m_1 and m_2 respectively. Because $\sigma_1(id_{IO}) \equiv \sigma_2(id_{IO})$, the lemma holds.
2. The evaluation of e reduces to zero, $\mathcal{E}'[\![e]\!]\sigma_1 = \mathcal{E}'[\![e]\!]\sigma_2 = (0, v_{\mathfrak{of}})$.
   The execution of S_1 continues as follows.

   $(\text{while}_{\langle n_1 \rangle}(e) \, \{S_1'\}, m_1(\sigma_1))$
   $= (\text{while}_{\langle n_1 \rangle}((0, v_{\mathfrak{of}})) \, \{S_1'\}, m_1(\sigma_1))$
   by the rule EEval'
   $\rightarrow (\text{while}_{\langle n_1 \rangle}(0) \, \{S_1'\}, m_1(\sigma_1))$
   by the E-Oflow1 or E-Oflow2 rule
   $\rightarrow (\text{skip}, m_1(\sigma_1))$ by the Wh-F rule.

   Similarly, the execution of S_2 gets to the configuration $(\text{skip}, m_2(\sigma_2))$. Loop counters of S_1 and S_2 are less than 1 and value stores agree on values of used/defined variables in both S_1 and S_2

as well as the input sequence variable and the I/O sequence variable.

3. The evaluation of e reduces to the same nonzero integer value, $\mathcal{E}'[\![e]\!]\sigma_1 = \mathcal{E}'[\![e]\!]\sigma_2 = (v, v_{\mathtt{of}})$ where $v \neq 0$.
Then the execution of S_1 proceeds as follows:

$$\begin{aligned}
&(\text{while}_{\langle n_1\rangle}(e)\,\{S_1'\}, m_1(\sigma_1))\\
=\ &(\text{while}_{\langle n_1\rangle}((v, v_{\mathtt{of}}))\,\{S_1'\}, m_1(\sigma_1))\\
&\text{by the rule EEval'}\\
\to&(\text{while}_{\langle n_1\rangle}(v)\,\{S_1'\}, m_1(\sigma_1))\\
&\text{by the E-Oflow1 or E-Oflow2 rule}\\
\to&(S_1'; \text{while}_{\langle n_1\rangle}(e)\,\{S_1'\}, m_1(\text{loop}_c^1[1/n_1], \sigma_1))\\
&\text{by the Wh-T rule.}
\end{aligned}$$

Similarly, the execution of S_2 proceeds to the configuration $(S_2'; \text{while}_{\langle n_2\rangle}(e)\,\{S_2'\}, m_2(\text{loop}_c^2[1/n_2], \sigma_2))$.
By the assumption, we show that S_1' and S_2' terminate in the same way and produce the equivalent I/O sequence when started in the state $m_1(\text{loop}_c^{1_1}, \sigma_1)$ and $m_2(\text{loop}_c^{2_1}, \sigma_2)$ respectively, and S_1' and S_2' have equivalent computation of variables defined in both statement sequences if both terminate. We need to show that all conditions are satisfied for the application of the assumption.

- Values of the I/O sequence variable in value stores σ_1 and σ_2 are equivalent, $\sigma_1(id_{IO}) \equiv \sigma_2(id_{IO})$;
The above two conditions are by assumption.
- Value stores σ_1 and σ_2 agree on values of used variables in S_1' and S_2' as well as the input sequence variable.
By definition, Use(S_1') $\subseteq$ Use(S_1). So are the cases to S_2' and S_2. In addition, value stores σ_1 and σ_2 are not changed in the evaluation of the predicate expression e. The condition holds.

By assumption, S_1' and S_2' terminate in the same way and produce the equivalent output sequence when started in states $m_1(\text{loop}_c', \sigma_1)$ and $m_2(\text{loop}_c', \sigma_2)$. In addition, S_1' and S_2' have equivalent computation of variables used or defined in S_1' and S_2' when started in states $m_1(\text{loop}_c', \sigma_1)$ and $m_2(\text{loop}_c', \sigma_2)$.
Then there are two cases.

(a) S_1' and S_2' both do not terminate and produce the equivalent I/O sequence.
By Lemma E.2, $S_1'; S_1$ and $S_2'; S_2$ both do not terminate and produce the equivalent I/O sequence.

(b) S_1' and S_2' both terminate and have equivalent computation of variables defined in S_1' and S_2'.
By assumption, $(S_1', m_1(\text{loop}_c', \sigma_1)) \xrightarrow{*} (\text{skip}, m_1'(\text{loop}_c'', \sigma_1'))$; $(S_2', m_2(\text{loop}_c', \sigma_2)) \xrightarrow{*} (\text{skip}, m_2'(\text{loop}_c'', \sigma_2'))$ where $\forall x \in (\text{Def}(S_1') \cap \text{Def}(S_2')) \cup \{id_I\}, \sigma_1'(x) = \sigma_2'(x)$.
By assumption,Use(S_1') = Use(S_2') and Def(S_1') = Def(S_2'). Then variables used in the predicate expression of S_1 and S_2 are either in variables used or defined in both S_1' and S_2' or not. Therefore value stores σ_2' and σ_1' agree on values of variables used in the expression e and even variables used or defined in S_1 and S_2.
By assumption, S_1 and S_2 produce the equivalent output sequence.

Induction step on iterations

The induction hypothesis (IH) is that, when $i \geq 1$, one of the following holds:

1. Loop counters for S_1 and S_2 are always less than i if any is present, $\forall m_1'(\text{loop}_c^{1'})\, m_2'(\text{loop}_c^{2'})\ :\ (S_1, m_1(\text{loop}_c^1, \sigma_1)) \xrightarrow{*} (S_1'', m_1'(\text{loop}_c^{1'})), \text{loop}_c^{1'}(n_1) < i, (S_2, m_2(\text{loop}_c^2, \sigma_2)) \xrightarrow{*} (S_2'', m_2'(\text{loop}_c^{2'})), \text{loop}_c^{2'}(n_2) < i$, S_1 and S_2 terminate in the same way, and have equivalent computation of defined variables in both S_1 and S_2 and the input sequence variable, $(S_1, m_1) \equiv_H (S_2, m_2)$ and $\forall x \in \text{Def}(S) \cup \{id_I\}\ :\ (S_1, m_1) \equiv_x (S_2, m_2)$; S_1 and S_2 produce the equivalent I/O sequence;
2. The loop counter of S_1 and S_2 are of value less than or equal to i, and there are no reachable configurations $(S_1, m_1(\text{loop}_c^{1_i}, \sigma_{1_i}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(\text{loop}_c^{2_i}, \sigma_{2_i}))$ from $(S_2,\ m_2(\sigma_2))$ where all of the following hold:
   - The loop counters of S_1 and S_2 are of value i, $\text{loop}_c^{1_i}(n_1) = \text{loop}_c^{2_i}(n_2) = i$.
   - Value stores σ_{1_i} and σ_{2_i} agree on values of used variables in both S_1 and S_2 as well as the input sequence variable, $\forall x \in$ (Use$(S_1) \cap$ Use$(S_2)) \cup \{id_I\}\ :\ \sigma_{1_i}(x) = \sigma_{2_i}(x)$.
   - Values of the I/O sequence variable in value stores σ_{1_i} and σ_{2_i}, $\sigma_{1_i}(id_{IO}) \equiv \sigma_{2_i}(id_{IO})$;
3. There are reachable configurations $(S_1, m_1(\text{loop}_c^{1_i}, \sigma_{1_i}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(\text{loop}_c^{2_i}, \sigma_{2_i}))$ from $(S_2,\ m_2(\sigma_2))$ where all of the following hold:
   - The loop counter of S_1 and S_2 are of value i, $\text{loop}_c^{1_i}(n_1) = \text{loop}_c^{2_i}(n_2) = i$.
   - Value stores σ_{1_i} and σ_{2_i} agree on values of used variables in both S_1 and S_2 as well as the input sequence variable, $\forall x \in$ (Use$(S_1) \cap$ Use$(S_2)) \cup \{id_I\}\ :\ \sigma_{1_i}(x) = \sigma_{2_i}(x)$.
   - Values of the I/O sequence variable in value stores σ_{1_i} and σ_{2_i} are equivalent, $\sigma_{1_i}(id_{IO}) \equiv \sigma_{2_i}(id_{IO})$;

Then we show that, when $i+1$, one of the following holds: The induction hypothesis (IH) is that, when $i \geq 1$, one of the following holds:

1. Loop counters for S_1 and S_2 are always less than $i+1$ if any is present, $\forall m_1'(\text{loop}_c^{1'})\, m_2'(\text{loop}_c^{2'})\ :\ (S_1, m_1(\text{loop}_c^1, \sigma_1)) \xrightarrow{*} (S_1'', m_1'(\text{loop}_c^{1'})), \text{loop}_c^{1'}(n_1) < i{+}1, (S_2, m_2(\text{loop}_c^2, \sigma_2)) \xrightarrow{*} (S_2'', m_2'(\text{loop}_c^{2'})), \text{loop}_c^{2'}(n_2) < i+1$, S_1 and S_2 terminate in the same way, produce the equivalent I/O sequence, and have equivalent computation of defined variables in both S_1 and S_2 and the input sequence variable, $(S_1, m_1) \equiv_H (S_2, m_2)$ and $(S_1, m_1) \equiv_O (S_2, m_2)$ and $\forall x \in (\text{Def}(S) \cup \{id_I\}\ :\ (S_1, m_1) \equiv_x (S_2, m_2)$; S_1 and S_2 produce the "equivalent" I/O sequence variable;
2. The loop counter of S_1 and S_2 are of value less than or equal to $i+1$, and there are no reachable configurations $(S_1, m_1(\text{loop}_c^{1_{i+1}}, \sigma_{1_{i+1}}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(\text{loop}_c^{2_{i+1}}, \sigma_{2_{i+1}}))$ from $(S_2, m_2(\sigma_2))$ where all of the following hold:
   - The loop counters of S_1 and S_2 are of value $i+1$, $\text{loop}_c^{1_{i+1}}(n_1) = \text{loop}_c^{2_{i+1}}(n_2) = i+1$.
   - Value stores $\sigma_{1_{i+1}}$ and $\sigma_{2_{i+1}}$ agree on values of used variables in both S_1 and S_2 as well as the input sequence variable, $\forall x \in$ (Use$(S_1) \cap$ Use$(S_2)) \cup \{id_I\}\ :\ \sigma_{1_{i+1}}(x) = \sigma_{2_{i+1}}(x)$.
   - Values of the I/O sequence variable in value stores $\sigma_{1_{i+1}}$ and $\sigma_{2_{i+1}}$ are equivalent, $\sigma_{1_{i+1}}(id_{IO}) \equiv \sigma_{2_{i+1}}(id_{IO})$;
3. There are reachable configurations $(S_1, m_1(\text{loop}_c^{1_{i+1}}, \sigma_{1_i}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(\text{loop}_c^{2_{i+1}}, \sigma_{2_i}))$ from $(S_2,\ m_2(\sigma_2))$ where all of the following hold:
   - The loop counter of S_1 and S_2 are of value i, $\text{loop}_c^{1_{i+1}}(n_1) = \text{loop}_c^{2_{i+1}}(n_2) = i+1$.
   - Value stores $\sigma_{1_{i+1}}$ and $\sigma_{2_{i+1}}$ agree on values of used variables in both S_1 and S_2 as well as the input sequence variable, $\forall x \in$ (Use$(S_1) \cap$ Use$(S_2)) \cup \{id_I\}\ :\ \sigma_{1_{i+1}}(x) = \sigma_{2_{i+1}}(x)$;
   - Values of the I/O sequence variable in value stores $\sigma_{1_{i+1}}$ and $\sigma_{2_{i+1}}$ are equivalent, $\sigma_{1_{i+1}}(id_{IO}) \equiv \sigma_{2_{i+1}}(id_{IO})$;

By hypothesis IH, there is no configuration where loop counters of S_1 and S_2 are of value $i+1$ when any of the following holds:

1. Loop counters for S_1 and S_2 are always less than i if any is present, $\forall m_1'(\text{loop}_c^{1'})\, m_2'(\text{loop}_c^{2'}) \,:\, (S_1, m_1(\text{loop}_c^1, \sigma_1)) \xrightarrow{*} (S_1'', m_1'(\text{loop}_c^{1'})), \text{loop}_c^{1'}(n_1) < i, (S_2, m_2(\text{loop}_c^2, \sigma_2)) \xrightarrow{*} (S_2'', m_2'(\text{loop}_c^{2'})), \text{loop}_c^{2'}(n_2) < i$, S_1 and S_2 terminate in the same way, produce the equivalent I/O sequence, and have equivalent computation of used/defined variables in both S_1 and S_2 and the input sequence variable, , the I/O sequence variable, $(S_1, m_1) \equiv_H (S_2, m_2)$ and $(S_1, m_1) \equiv_O (S_2, m_2)$ and $\forall x \in (\text{Def}(S_1) \cap \text{Def}(S_2)) \cup \{id_I\} \,:\, (S_1, m_1) \equiv_x (S_2, m_2)$; S_1 and S_2 produce , and the I/O sequence variable;
2. The loop counter of S_1 and S_2 are of value less than or equal to i, and there are no reachable configurations $(S_1, m_1(\text{loop}_c^{1_i}, \sigma_{1_i}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(\text{loop}_c^{2_i}, \sigma_{2_i}))$ from $(S_2,\, m_2(\sigma_2))$ where all of the following hold:
   - The loop counters of S_1 and S_2 are of value i, $\text{loop}_c^{1_i}(n_1) = \text{loop}_c^{2_i}(n_2) = i$.
   - Value stores σ_{1_i} and σ_{2_i} agree on values of used variables in both S_1 and S_2 as well as the input sequence variable, $\forall x \in (\text{Use}(S_1) \cap \text{Use}(S_2)) \cup \{id_I, , id_{IO}\} \,:\, \sigma_{1_i}(x) = \sigma_{2_i}(x)$.
   - Values of the I/O sequence variable in value stores σ_{1_i} and σ_{2_i} are equivalent, $\sigma_{1_i}(id_{IO}) \equiv \sigma_{2_i}(id_{IO})$;

When there are reachable configurations $(S_1, m_1(\text{loop}_c^{1_i}, \sigma_{1_i}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(\text{loop}_c^{2_i}, \sigma_{2_i}))$ from $(S_2,\, m_2(\sigma_2))$ where all of the following hold:

- The loop counter of S_1 and S_2 are of value i, $\text{loop}_c^{1_i}(n_1) = \text{loop}_c^{2_i}(n_2) = i$.
- Value stores σ_{1_i} and σ_{2_i} agree on values of used variables in both S_1 and S_2 as well as the input sequence variable, $\forall x \in (\text{Use}(S_1) \cap \text{Use}(S_2)) \cup \{id_I\} \,:\, \sigma_{1_i}(x) = \sigma_{2_i}(x)$.
- Values of the I/O sequence variable in value stores σ_{1_i} and σ_{2_i} are equivalent, $\sigma_{1_i}(id_{IO}) \equiv \sigma_{2_i}(id_{IO})$;

By similar argument in base case, we have one of the following holds:

1. Loop counters for S_1 and S_2 are always less than $i+1$ if any is present, $\forall m_1'(\text{loop}_c^{1'})\, m_2'(\text{loop}_c^{2'}) \,:\, (S_1, m_1(\text{loop}_c^1, \sigma_1)) \xrightarrow{*} (S_1'', m_1'(\text{loop}_c^{1'})), \text{loop}_c^{1'}(n_1) < i{+}1, (S_2, m_2(\text{loop}_c^2, \sigma_2)) \xrightarrow{*} (S_2'', m_2'(\text{loop}_c^{2'})), \text{loop}_c^{2'}(n_2) < i$, S_1 and S_2 terminate in the same way, produce the equivalent I/O sequence, and have equivalent computation of defined variables in both S_1 and S_2 and the input sequence variable, $(S_1, m_1) \equiv_H (S_2, m_2)$ and $\forall x \in (\text{Def}(S)) \cup \{id_I\} \,:\, (S_1, m_1) \equiv_x (S_2, m_2)$;
2. The loop counter of S_1 and S_2 are of value less than or equal to $i+1$, and there are no reachable configurations $(S_1, m_1(\text{loop}_c^{1_i}, \sigma_{1_i}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(\text{loop}_c^{2_i}, \sigma_{2_i}))$ from $(S_2,\, m_2(\sigma_2))$ where all of the following hold:
   - The loop counters of S_1 and S_2 are of value i, $\text{loop}_c^{1_{i+1}}(n_1) = \text{loop}_c^{2_{i+1}}(n_2) = i+1$.
   - Value stores $\sigma_{1_{i+1}}$ and $\sigma_{2_{i+1}}$ agree on values of used variables in both S_1 and S_2 as well as the input sequence variable, $\forall x \in (\text{Use}(S_1) \cap \text{Use}(S_2)) \cup \{id_I\} \,:\, \sigma_{1_{i+1}}(x) = \sigma_{2_{i+1}}(x)$.
   - Values of the I/O sequence variable in value stores $\sigma_{1_{i+1}}$ and $\sigma_{2_{i+1}}$ are equivalent, $\sigma_{1_{i+1}}(id_{IO}) \equiv \sigma_{2_{i+1}}(id_{IO})$;
3. There are reachable configurations $(S_1, m_1(\text{loop}_c^{1_{i+1}}, \sigma_{1_{i+1}}))$ from $(S_1, m_1(\sigma_1))$, $(S_2, m_2(\text{loop}_c^{2_{i+1}}, \sigma_{2_{i+1}}))$ from $(S_2,\, m_2(\sigma_2))$ where all of the following hold:
   - The loop counter of S_1 and S_2 are of value i, $\text{loop}_c^{1_{i+1}}(n_1) = \text{loop}_c^{2_{i+1}}(n_2) = i$.
   - The loop counter of S_1 and S_2 are of value i, $\text{loop}_c^{1_{i+1}}(, n_1) = \text{loop}_c^{2_{i+1}}(, n_2) = i$.
   - Value stores $\sigma_{1_{i+1}}$ and $\sigma_{2_{i+1}}$ agree on values of used variables in both S_1 and S_2 as well as the input sequence variable, $\forall x \in (\text{Use}(S_1) \cap \text{Use}(S_2)) \cup \{id_I\} \,:\, \sigma_{1_{i+1}}(x) = \sigma_{2_{i+1}}(x)$;
   - Values of the I/O sequence variable in value stores $\sigma_{1_{i+1}}$ and $\sigma_{2_{i+1}}$ are equivalent, $\sigma_{1_{i+1}}(id_{IO}) \equiv \sigma_{2_{i+1}}(id_{IO})$;

□

```
1:                               1':  b := 2
2:  If (a > 0) then              2':  If (a > 0) then
3:      b := c + 1               3':      b := c + 1
4:  output b + c                 4':  output b + c

    old                               new
```

Figure 18: Missing initialization

6.6 Proof rule for missing variable initializations

A kind of bugfix we call *missing-initialization* includes variable initialization for those in the imported variables relative to the I/O sequence variable in the old program. Figure 18 shows an example of missing-initializations. The initialization $b := 2$ ensures the value used in "output $b + c$" is not to be undefined. In general, new variable initializations only affect rare buggy executions of the old program, where there are uses of undefined imported variables relative to the I/O sequence variable in the program. Because DSU is not starting in error state, we assume that, in the proof of backward compatibility, there are no uses of variables with undefined variables in executions of the old program.

The following is the definition of the update class "missing initializations".

Definition 32. (Missing initializations) *A statement sequence S_2 includes updates of missing initializations compared with a statement sequence S_1, written $S_2 \approx^S_{\mathit{Init}} S_1$, iff $S_2 = S_{\mathit{Init}}; S_1$ where S_{Init} is a sequence of assignment statements of form "$lval := v$" and $\mathit{Def}(S_{\mathit{Init}}) \subseteq \mathit{Imp}(S_1, \{id_{IO}\})$;*

Though the bugfix in the update of missing initializations are not in rare execution in the first case in Definition 32, the definition shows the basic form of bugfix clearly.

We show that two statement sequences terminate in the same way, produce the same output sequence, and have equivalent computation of defined variables in both programs in valid executions if there are updates of missing initializations between them.

Lemma 6.14. *Let S_1 and S_2 be two statement sequences respectively where there are updates of "missing initializations" in S_2 compared with S_1, $S_2 \approx^S_{\mathit{Init}} S_1$. If S_1 and S_2 start in states $m_1(\sigma_1)$ and $m_2(\sigma_2)$ respectively such that both of the following hold:*

- *Value stores σ_1 and σ_2 agree on values of variables used in both S_1 and S_2 as well as the input sequence variable and the I/O sequence variable, $\forall id \in (\mathit{Use}(S_1) \cap \mathit{Use}(S_2)) \cup \{id_I, id_{IO}\} \,:\, \sigma_1(id) = \sigma_2(id)$;*
- *defined variables in S_{Init} are of undefined value in value stores σ_1, σ_2, $\forall id \in \mathit{Def}(S_{\mathit{Init}}) \,:\, \sigma_1(id) = \sigma_2(id) = \mathit{Udf}[\![\tau]\!]$ where τ is the type of the variable id;*
- *There are no use of variables with undefined values in the execution of S_1;*
- *There are no crash in execution of S_{Init};*

then S_1 and S_2 terminate in the same way, produce the same output sequence, and when S_1 and S_2 both terminate, they have equivalent computation of used variables and defined variables in both S_1 and S_2 as well as the input sequence variable and the I/O sequence variable,

- $(S_1, m_1) \equiv_H (S_2, m_2)$;
- $(S_1, m_1) \equiv_O (S_2, m_2)$;
- $\forall x \in (Def(S_1) \cup Def(S_2)) \cup \{id_I, id_{IO}\} : (S_1, m_1) \equiv_x (S_2, m_2)$;

Proof. By induction on the sum of the program size of S_1 and S_2, $\text{size}(S_1) + \text{size}(S_2)$.
Base case. $S_1 = s$ and $S_2 = S_{\text{Init}}; s$ where $S_{\text{Init}} =$ "$lval := v$" and $\text{Def}(lval) \in \text{Use}(s)$;

There are cases regarding $lval$ in S_{Init}.

1. $lval = id$.
   Then the execution of S_2 proceeds as follows.

   $(id := v; s, m_2(\sigma_2))$
   $\rightarrow(\text{skip}; s, m_2(\sigma_2[v/(id)]))$
   by the rule As-Scl
   $\rightarrow(s, m_2(\sigma_2[v/(id)]))$ by the rule Seq.

   By assumption, $id \in \text{Use}(e)$. By assumption, the value of id is undefined in value store σ_1. Then there is no valid execution of S_1. Then it holds that, in valid executions of S_1, S_1 and S_2 terminate in the same way, produce the same output sequence, and have equivalent computation of defined variables in both S_1 and S_2. Then this lemma holds.
2. $lval = id[n]$.
   Then the execution of S_2 proceeds as follows.

   $(id[n] := v; s, m_2(\sigma_2))$
   $\rightarrow(\text{skip}; s, m_2(\sigma_2[v/(id, n)]))$
   by the rule As-Err
   $\rightarrow(s, m_2(\sigma_2[v/(id, n)]))$ by the rule Seq.

   By similar argument above, this lemma holds.
3. $lval = id_1[id_2]$.
   Then the execution of S_2 proceeds as follows.

   $(id_1[id_2] := v; s, m_2(\sigma_2))$
   $\rightarrow(id_1[v_1] := v; s, m_2(\sigma_2[v/(id, n)]))$
   by the rule Var
   $\rightarrow(\text{skip}; s, m_2(\sigma_2[v/(id, v_1)]))$ by the rule As-Arr.
   $\rightarrow(s, m_2(\sigma_2[v/(id, v_1)]))$ by the rule Seq.

   By similar argument above, this lemma holds.

Induction step.

The hypothesis is that this lemma holds when the sum k of the program size of S_1 and S_2 are great than or equal to 3, $k \geq 3$.

We then show that this lemma holds when the sum of the program size of S_1 and S_2 is $k + 1$.

$S_2 = S_{\text{Init}}; S_1$ where S_{Init} is a sequence of assignment statements and $\text{Def}(S_{\text{Init}}) \in \text{Imp}(S_1, \{id_{IO}\})$;

The proof is similar to that in the base case. By assumption, the execution of S_{Init} does not crash, $(S_{\text{Init}}, m_2(\sigma_2)) \overset{*}{\rightarrow} (\text{skip}, m_2(\sigma_2'))$ where $\sigma_2' = \sigma_2[v_1/x_1]...[v_k/x_k]$ and $\forall 1 \leq i \leq k : x_k \in \text{Def}(S_{\text{Init}})$.

By assumption, there are no use of variables with undefined values in the execution of S_1 by Theorem 5 and Theorem 4, this lemma holds. □

7. Related Work

We discuss related work on DSU safety and program equivalence in order.

Existing studies on DSU safety could be roughly divided into high level studies and low level ones. There are a few studies on high level DSU safety. In [19], Kramer and Magee defined the DSU correctness that the updated system shall "operate as normal instead of progressing to an error state". This is covered by our requirement that hybrid executions conform to the old program's specification and our accommodation for bug fixes. Moreover, our backward compatibility includes I/O behavior, which is more concrete than the behavior in [19]. In [9], Bloom and Day proposed a DSU correctness which allows functionality extension that could not produce past behavior. This is probably because Bloom and Day considered updated environment. On the contrary, we assume that the environment is not updated. In addition, we explicitly present the error state, which is not mentioned in [9]. Panzica La Manna [28] presented a high level correctness only considering scenario-based specifications for controller systems instead of general programs.

There are also studies on low level DSU safety. Hayden et al. [15] discussed DSU correctness and concluded that there is only client-oriented correctness. Zhang et al. [32] asked the developers to ensure DSU correctness. Magill et al. [24] did ad-hoc program correlation without definitions of any correctness. We consider that there is general principle of DSU safety. The difference lies at the abstraction of the program behavior. We model program behavior by concrete I/O while others [15, 24, 32] consider a general program behavior.

We next discuss existing work on program equivalence. There is a rich literature on program equivalence and we compare our work only with most related work. Our study of program equivalence is inspired by original work of Horwitz et al. [17] on program dependence graphs, but we take a much more formal approach and we consider terminating as well as non-terminating programs with recurring I/O. In [13], Godlin and Strichman have a structured study of program equivalence similar to that of ours. Godlin and Strichman [13] restricted the equivalence to corresponding functions and therefore weakens the applicability to general transformations affecting loops such as loop fission, loop fusion and loop invariant code motion. However, our program equivalence allows loop optimizations such as loop fusion and loop fission. Furthermore, our syntactic conditions imply more program point mapping because we allow corresponding program point in arbitrary nested statements and in the middle of program that does not include function call.

8. Conclusion

In this paper, we propose a formal and practical general definition of DSU correction based on I/O sequences, backward compatibility. We devised a formal language and adapt the general definition of DSU correctness for executable programs based on our language. Based on the adapted backward compatibility, we proposed syntactic conditions that help guarantee correct DSUs for both terminating and nonterminating executions. In addition, we formalize typical program updates that are provably backward compatible, covering both new feature and bugfix.

In the future, we plan to identify more backward compatible update patterns by studying more open source programs. Though it is dubious if open source programs' evolution history includes typical update patterns, open source programs are the most important source of widely-used programs for our study of DSU. In addition, we plan to develop an algorithm for automatic state mapping based on our syntactic condition of program equivalence and definition of update classes.

References

[1] Software Engineering - Software Life Cycle Processes - Maintenance. Technical Report ISO/IEC 14764:2006(E). 2.4

[2] Openssh users. `http://www.openssh.com/users.html`, 2015. [Online; accessed 15-Jan-2015]. 3

[3] Vsftpd wikipage. `http://en.wikipedia.org/wiki/Vsftpd`, 2015. [Online; accessed 15-Jan-2015]. 3

[4] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. 5.2.1

[5] J. Arnold and M. F. Kaashoek. KSplice: Automatic Rebootless Kernel Updates. In *EuroSys 2009*, April 2009. 1

[6] N. Benton. Simple relational correctness proofs for static analyses and program transformations. *SIGPLAN Not.*, 39(1):14–25, Jan. 2004. 1

[7] D. Binkley, S. Horwitz, and T. Reps. The multi-procedure equivalence theorem. Technical report, 1989. 1

[8] S. Blazy and X. Leroy. Mechanized semantics for the clight subset of the c language. *Journal of Automated Reasoning*, 43(3), 2009. 4.2

[9] T. Bloom and M. Day. Reconfiguration and module replacement in argus: theory and practice. *Software Engineering Journal*, Mar 1993. 1, 7

[10] L. Cardelli. Type systems. *ACM Computing Surveys*, 28(1):263–264, 1996. A

[11] R. Cartwright and M. Felleisen. The semantics of program dependence. *SIGPLAN Not.*, 24(7):13–27, 1989. 1

[12] W. Dietz, P. Li, J. Regehr, and V. Adve. Understanding integer overflow in c/c++. ICSE 2012, pages 760–770, 2012. 2

[13] B. Godlin and O. Strichman. Inference rules for proving the equivalence of recursive procedures. *Acta Informatica*, 45(6):403–439, 2008. 1, 7

[14] A. D. Gordon. *Functional programming and input/output*. Number 8. Cambridge University Press, 1994. 4.1, 4.2

[15] C. M. Hayden, S. Magill, M. Hicks, N. Foster, and J. S. Foster. Specifying and verifying the correctness of dynamic software updates. VSTTE '12, pages 278–293, Jan. 2012. 1, 7

[16] M. Hicks. *Dynamic Software Updating*. PhD thesis, University of Pennsylvania, August 2001. 1

[17] S. Horwitz, J. Prins, and T. Reps. On the adequacy of program dependence graphs for representing programs. POPL '88, pages 146–157. ACM, 1988. 1, 7

[18] C. Karfa, K. Banerjee, D. Sarkar, and C. Mandal. Verification of loop and arithmetic transformations of array-intensive behaviors. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 32(11):1787–1800, Nov 2013. 1

[19] J. Kramer and J. Magee. The evolving philosophers problem: dynamic change management. *Software Engineering, IEEE Transactions on*, 16, Nov 1990. 1, 7

[20] S. Kundu, Z. Tatlock, and S. Lerner. Proving optimizations correct using parameterized program equivalence. *SIGPLAN Not.*, 44(6). 1

[21] D. Lacey, N. D. Jones, E. Van Wyk, and C. C. Frederiksen. Proving correctness of compiler optimizations by temporal logic. *SIGPLAN Not.*, 37(1):283–294, Jan. 2002. ISSN 0362-1340. 1

[22] Y.-F. Lee and R.-C. Chang. Hotswapping linux kernel modules. *Journal of Systems and Software*, 79(2):163–175, February 2006. 1

[23] D. Lucanu and V. Rusu. Program equivalence by circular reasoning. In *Integrated Formal Methods*, volume 7940 of *Lecture Notes in Computer Science*, pages 362–377. Springer Berlin Heidelberg, 2013. 1

[24] S. Magill, M. Hicks, S. Subramanian, and K. S. McKinley. Automating object transformations for dynamic software updating. *SIGPLAN Not.*, 47(10):265–280, Oct. 2012. 1, 7

[25] K. Makris. *Whole-Program Dynamic Software Updating*. PhD thesis, Arizona State University, December 2009. 1, 3

[26] K. Makris and R. A. Bazzi. Immediate Multi-Threaded Dynamic Software Updates Using Stack Reconstruction. In *Proceedings of the USENIX '09 Annual Technical Conference*, June 2009. 3

[27] I. Neamtiu. *Practical Dynamic Software Updating*. PhD thesis, University of Maryland, August 2008. 3

[28] V. Panzica La Manna, J. Greenyer, C. Ghezzi, and C. Brenner. Formalizing correctness criteria of dynamic updates derived from specification changes. SEAMS '13, pages 63–72, 2013. 1, 7

[29] D. L. Parnas. Software aging. ICSE '94, Los Alamitos, CA, USA. IEEE Computer Society Press. 2.4

[30] B. C. Pierce. *Types and programming languages*. MIT press, 2002. 4.2

[31] S. Verdoolaege, G. Janssens, and M. Bruynooghe. Equivalence checking of static affine programs using widening to handle recurrences. *ACM Trans. Program. Lang. Syst.*, 34(3):11:1–11:35, Nov. 2012. 1

[32] M. Zhang, K. Ogata, and K. Futatsugi. Formalization and verification of behavioral correctness of dynamic software updates. *Electronic Notes in Theoretical Computer Science*, 294:12 – 23, 2013. 1, 7

A. Type system

Figure 19 shows an almost standard unsound and incomplete type system. The type system is unsound because of three reasons, (a) the possible value mismatch due to the subtype rule from hte type Int to Long, (b) the implicit subtype between enumeration types and the type Long allowed by our semantics and (c) the possible array index out of bound. The type system is incomplete due to the parameterized "other" expressions. The notation Dom(Γ) borrowed from Cardelli [10] in rules Tvar1, Tvar2, Tlabels an Tfundecl refers to the domain of the typing environment Γ, which are identifiers bound to a type in Γ.

B. Syntactic definitions

The syntax-directed definitions listed below make our argument independent of existing program analysis partially.

Definition 33. (Idx($lval$)) *The used variables in index of a left value $lval$, written Idx($lval$), are listed as follows:*

1. *Idx(id) = $\emptyset$;*
2. *Idx($id[n]$) = $\emptyset$;*
3. *Idx($id_1[id_2]$) = $\{id_2\}$;*

Definition 34. (Base($lval$)) *The base of a left value $lval$, written Base($lval$), is listed as follows:*

1. *Base(id) = $\{id\}$;*
2. *Base($id[n]$) = $\{id\}$;*
3. *Base($id_1[id_2]$) = $\{id_1\}$;*

Definition 35. (Use(e)) *The set of used variables in an expression e, written Use(e), are listed as follows:*

1. *Use($lval$) = Base($lval$) $\cup$ Idx($lval$);*
2. *Use($id == l$) = $\{id\}$;*
3. *Use($other$) $=$ Use($other$) where function Use : other $\rightarrow \{id\}$ is parameterized;*

Definition 36. (Use(S)) *The used variables in a sequence of statements S, written Use(S), are listed as follows:*

1. *Use($skip$) $= \emptyset$;*
2. *Use($lval := e$) = Use(e) $\cup$ Idx($lval$);*
3. *Use($output\ e$) = Use(e) $\cup \{id_{IO}\}$;*
4. *Use($input\ id$) = $\{id_I, id_{IO}\}$;*
5. *Use(If (e) then $\{S_t\}$ else $\{S_f\}$) = Use(e)$\cup$Use(S_t)$\cup$Use(S_f);*
6. *Use(while$_{\langle n\rangle}$$(e)\{S'\}$) = Use($e$) $\cup$ Use(S');*

$\boxed{\Gamma \vdash \diamond}$

$$\textbf{TInit}\ \frac{}{\Gamma \vdash \diamond}$$

$\boxed{\Gamma \to \Gamma'}$

(Tvar1)

$$\frac{\Gamma \vdash \diamond \qquad V = V', \tau\, id \quad id \notin \text{Dom}(\Gamma)}{\Gamma, id : \tau \vdash \diamond}$$

(Tlabels)

$$\frac{\Gamma \vdash \diamond \quad k \geq 1 \quad id \notin \text{Dom}(\Gamma) \qquad EN = EN', \text{enum}\, id\{l_1, ..., l_k\}}{\Gamma, id : \{l_1, ..., l_k\} \vdash \diamond}$$

(Tprompt)

$$\frac{\Gamma \vdash \diamond \qquad Pmpt = \{l_1 : n_1, \ldots, l_k : n_k\} \quad \text{pmpt} \notin \text{Dom}(\Gamma)}{\Gamma, \text{pmpt} : \{l_1 : n_1, \ldots, l_k : n_k\} \vdash \diamond}$$

(Tvar2)

$$\frac{\Gamma \vdash \diamond \quad id \notin \text{Dom}(\Gamma) \qquad V = V', \tau\, id[n] \quad n > 0}{\Gamma, id : \text{array}(\tau, n) \vdash \diamond}$$

$\boxed{\Gamma \vdash \tau}$

(Tint)

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \text{Int}}$$

(Tlong)

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \text{Long}}$$

(Tenum)

$$\frac{\Gamma \vdash id : \{l_1, ..., l_k\}}{\Gamma \vdash \text{enum}\ id}$$

$\boxed{\Gamma \vdash e : \tau}$

(Topnd)

$$\frac{}{\Gamma, id : \tau \vdash id : \tau}$$

(Tequiv)

$$\frac{\Gamma \vdash id' : \{l_1, ..., l, ..., l_k\} \qquad \Gamma \vdash id : \text{enum}\, id'}{\Gamma \vdash (id == l) : \text{Long}}$$

(TSub)

$$\frac{\Gamma \vdash e : \text{Int}}{\Gamma \vdash e : \text{Long}}$$

(Tarray1)

$$\frac{\Gamma \vdash id : \text{array}(\tau, n) \qquad \Gamma \vdash id' : \text{Long}}{\Gamma \vdash id[id'] : \tau}$$

(Tarray2)

$$\frac{\Gamma \vdash id : \text{array}(\tau, n) \qquad 1 \leq k \leq n}{\Gamma \vdash id[k] : \tau}$$

$\boxed{\Gamma \vdash S}$

(Tassign)

$$\frac{\Gamma \vdash lval : \tau \qquad \Gamma \vdash e : \tau}{\Gamma \vdash lval := e}$$

(Tinput)

$$\frac{\Gamma \vdash id : \tau \qquad \tau \neq \text{pmpt}}{\Gamma \vdash \text{input}\ id}$$

(Toutput)

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{output}\, e}$$

(Tseq)

$$\frac{\Gamma \vdash S_1 \qquad \Gamma \vdash S_2}{\Gamma \vdash S_1; S_2}$$

(Tif)

$$\frac{\Gamma \vdash e : \text{Long} \qquad \Gamma \vdash S_1 \quad \Gamma \vdash S_2}{\Gamma \vdash \text{If}(e)\ \text{then}\, \{S_1\}\ \text{else}\, \{S_2\}}$$

(Twhile)

$$\frac{\Gamma \vdash e : \text{Long},\ \Gamma \vdash S}{\Gamma \vdash\ \text{while}(e)\{S\}}$$

$\boxed{\Gamma \vdash P}$

(Tprog)

$$\frac{\begin{array}{c} Pmpt = \{l_1 : n_1, ..., l_k : n_k\} \\ EN = \text{enum}\, id_1\{l_1, ..., l_r\}, ..., \text{enum}\, id_k\{l'_1, ..., l'_r\} \\ \Gamma \vdash \text{enum}\, id_i, 1 \leq i \leq k \qquad V = \tau'_1\, id'_1, ..., \tau'_k id'_k[n] \\ \Gamma \vdash id'_j : \tau'_j, 1 \leq j \leq k' - 1 \qquad \Gamma \vdash id'_k : \text{array}(\tau'_k, n) \\ \Gamma \vdash S_{entry} \end{array}}{\Gamma \vdash Pmpt; EN; V; S_{entry}}$$

Figure 19: Typing rules

7. *For $k > 0$, Use$(s_1; ...; s_{k+1})$ = Use$(s_1; ...; s_k) \cup$ Use(s_{k+1});*

Definition 37. (Def(S)) *The defined variables in a sequence of statements S, written Def(S), are listed as follows:*

1. *Def$(skip)$ = $\emptyset$;*
2. *Def$(id := e)$ = $\{id\}$;*
3. *Def$(input\ id)$ = $\{id_I, id_{IO}, id\}$;*
4. *Def$(output\ e)$ = $\{id_{IO}\}$;*
5. *Def$(If\ (e)\ then\ \{S_t\}\ else\ \{S_f\})$ = Def$(S_t) \cup$ Def(S_f);*
6. *Def$(while_{\langle n \rangle}(e)\{S\})$ = Def(S);*
7. *For $k > 0$, Def$(s_1; ...; s_{k+1})$ = Def$(s_1; ...; s_k) \cup$ Def(s_{k+1});*

Definition 38. *($s \in S$) We say a statement s is in a sequence of statements S of a program P, written $s \in S$, if one of the following holds:*

1. *$S = s$;*
2. *If S = "If(e) then $\{S_t\}$ else $\{S_f\}$", $(s \in S_t) \vee (s \in S_f)$;*
3. *If S = "while(e) $\{S'\}$", $s \in S'$;*
4. *For $k > 0$, if $S = s_1; ...; s_k; s_{k+1}$, $(s \in s_{k+1}) \vee (s \in s_1; ...; s_k)$;*

We write $s \notin S$ if $s \in S$ does not hold.

We show the definition of program size, which is based of our induction proof.

Definition 39. (*size*(S)) *The program size of a statement sequence S, written size(S), is listed as follows:*

1. *size("skip") = size("$id := e$") = size("$id_1 :=$ call $id_2(e^*)$") = size("input id") = size("output e") = 1;*
2. *size("If(e) then $\{S_t\}$ else $\{S_f\}$") = 1 + size(S_t) + size(S_f);*
3. *size("while(e) $\{S'\}$") = 1 + size(S');*
4. *For $k > 0$, size$(s_1; ...; s_k$ = $\sum_{i=1}^{k}$ size(s_i);*

C. Properties of imported variables

Lemma C.1. *Imp$(S_1; S_2, X)$ = Imp$(S_1$, Imp$(S_2, X))$.*

Proof. Let statement sequence S_2 = $s_1; s_2; ...; s_k$ for some $k > 0$. The proof is by induction on k. ☐

Corollary C.1. $\forall i \in \mathbb{Z}_+$, *Imp$(S^{i+1}, X)$ = Imp$(S$, Imp$(S^i, X))$.*

This is by lemma C.1.

Lemma C.2. *Imp$(S, A \cup B)$ = Imp$(S, A) \cup$ Imp(S, B).*

Proof. By structural induction on abstract syntax of statement sequence S. ☐

Lemma C.3. *For statement s = "while(e)$\{S\}$" and a set of finite number of variables X such that $X \cap$ Def $(s) \neq \emptyset$, there is $\beta > 0$ such that $\bigcup_{0 \leq i \leq (\beta+1)}$ Imp $(S^i, X) \subseteq \bigcup_{1 \leq j \leq \beta}$ Imp(S^j, X).*

Proof. By contradiction against the fact that is finite number of variables redefined in statement s. ☐

D. Properties of expression evaluation

We wrap the two properties of expression evaluation, which is based on the two properties of "other" expression evaluation. In the following, we use the notation $\mathcal{E}'$ to expand the domain of the expression meaning function $\mathcal{E}' : e \to \sigma \to (v_{\text{error}}, \{0, 1\})$.

Lemma D.1. *If every variable in Use(e) of an expression e has the same value w.r.t two value stores, the expression e evaluates to same value against the two value stores, $(\forall x \in$ Use$(e) : \sigma_1(x) = \sigma_2(x)) \Rightarrow (\mathcal{E}'[\![e]\!]\sigma_1 = \mathcal{E}'[\![e]\!]\sigma_2)$.*

Proof. The proof is a case analysis of the expression e.

1. $e = lval$;
   There are further cases regarding $lval$.
   (a) $lval = id$;
   By definition, Use$(e) = \{id\}$. Besides, there is no integer overflow in both evaluations. The lemma holds trivially.
   (b) $lval = id[n]$;
   By definition, Use$(e) = \{id\}$. Because the array has fixed size, by assumption, $\sigma_1(id, n) = \sigma_2(id, n)$ or $(id, n, *) \notin \sigma_1, (id, n, *) \notin \sigma_2$. Besides, there is no integer overflow in both evaluations. The lemma holds.

(c) $lval = id_1[id_2]$;
By definition, Use$(e) = \{id_1, id_2\}$. By assumption, $\sigma_1(id_2) = \sigma_2(id_2) = n$ By similar argument to the case $lval = id[n]$, the lemma holds.

2. $e =$ "$id == l$";
By definition, Use$(e) = \{id\}$. W.l.o.g, id is a global variable. By assumption, $\sigma_1(id) = \sigma_2(id) = l'$. If $l' = l$, by rule Eq-T, $(l' == l, m(\sigma)) \to (1, m)$. If $l' \neq l$, by rule Eq-F, $(l' == l, m(\sigma)) \to (0, m)$. Besides, there is no integer overflow in both evaluations. The lemma holds.
3. $e =$ other;
By definition, Use(e) = Use(e). By assumption, $\forall x \in$ Use$(e) : \sigma_1(x) = \sigma_2(x) \vee \sigma_1(x) = \sigma_2(x)$. The lemma holds by parameterized expression meaning function for "other" expression.

□

Lemma D.2. *If every variable in Err(e) of an expression e has same value w.r.t two pairs of (block, value store), $\forall x \in$ Err(e) : $\sigma_1(x) = \sigma_2(x)$ then one of the following holds:*

1. *the expression evaluates to crash against the two value stores, $(\mathcal{E}'[\![e]\!]\sigma_1 = (error, v_{\mathfrak{of}})) \wedge (\mathcal{E}'[\![e]\!]\sigma_2 = (error, v_{\mathfrak{of}}))$;*
2. *the expression evaluates to no crash against the two pairs of (block, value store) $(\mathcal{E}'[\![e]\!]\sigma_1 \neq (error, v^1_{\mathfrak{of}})) \wedge (\mathcal{E}'[\![e]\!]\sigma_2 \neq (error, v^2_{\mathfrak{of}}))$.*

Proof. The proof is a case analysis of the expression e.

1. $e = lval$;
There are further cases regarding $lval$.
(a) $lval = id$;
By definition, Err(e) = Idx$(e) = \emptyset$. By our semantic, the evaluation of id never crash. Besides, there is no integer overflow in both evaluations. The lemma holds.
(b) $lval = id[n]$;
By definition, Err(e) = Idx$(e) = \emptyset$. Because the array id_1 has a fixed array size, by assumption, either $((id_1, n, v_1) \in \sigma_1) \wedge ((id_1, n, v_2) \in \sigma_2)$ or $((id_1, n, v_1) \notin \sigma_1) \wedge ((id_1, n, v_2) \notin \sigma_2)$. Besides, there is no integer overflow in both evaluations. The lemma holds.
(c) $lval = id_1[id_2]$
By definition, Err(e) = Idx$(e) = \{id_2\}$. By assumption, $\sigma_1(id_2) = \sigma_2(id_2) = n$ or $\sigma_1(id_2) = \sigma_2(id_2) = n$. By similar argument to the case $lval = id[n]$, the lemma holds.
2. $e =$ "$id == l$";
By definition, Err$(e) = \emptyset$. W.l.o.g, id is a global variable. Let $\sigma_1(id) = l_1, \sigma_2(id) = l_2$. W.l.o.g., $l_1 = l$ and $l_2 \neq l$, by rule Eq-T, $(l_1 == l, m(\sigma)) \to (1, m)$ and, by rule Eq-F, $(l_2 == l, m(\sigma)) \to (0, m)$. Besides, there is no integer overflow in both evaluations. The lemma holds.
3. $e =$ other;
By definition, Err(e) = Err(e). By assumption, $\forall x \in$ Err(e) : $\sigma_1(x) = \sigma_2(x)$. The lemma holds by the property of parameterized expression meaning function for "other" expression.

□

With respect to Lemma D.1 and Lemma D.2, we extend semantic rule for expression evaluation as follows.

$$\boxed{(r, m) \to (r', m')}$$

$$\mathcal{E}' : e \to \sigma \to (v_{\text{error}} \times \{0, 1\})$$

$$\text{EEval'}\ \frac{\mathfrak{f} = 0}{(e, m(\mathfrak{f}, \sigma)) \to (\mathcal{E}'[\![e]\!]\sigma, m)}$$

Figure 20: An extended SOS rule for expressions

E. Properties of remaining execution

We assume that crash flag $\mathfrak{f}$ = 0 in given execution state $m(\mathfrak{f})$.

Lemma E.1. $(S_1, m) \to (S_1', m') \Rightarrow (S_1; S_2, m) \to (S_1'; S_2, m')$.

Proof. The proof is by structural induction on abstract syntax of S_1.

Case 1. S_1 = "skip".
By rule Seq, $(skip; S_2, m) \to (S_2, m)$ where $m = m'$.
Case 2. $S_1 =$ "$id := e$".
There are two subcases.
Case 2.1. $(e, m) \xrightarrow{*} (v, m)$ for some value v.
By rule Assign,
$(id := v, m) \to (\text{skip}, m(\sigma[v/x]))$.
Then, by contextual (semantic) rule,
$(id := v; S_2, m) \to (\text{skip}; S_2, m(\sigma[v/x]))$.
Case 2.2. $(e, m) \xrightarrow{*} (e', m(1/\mathfrak{f}))$ for some expression e'.
Then, by rule crash,
$(id := e', m(1/\mathfrak{f})) \to (id := e', m(1/\mathfrak{f}))$.
Then, by contextual rule,
$(id := e'; S_2, m(1/\mathfrak{f})) \to (id := e'; S_2, m(1/\mathfrak{f}))$.
Case 3. S_1= "output e"
Case 4. S_1= "input id"
By similar argument in Case 2, the lemma holds for case 3 and 4.
Case 5. S_1= "If (e) then $\{S_t\}$ else $\{S_f\}$".
Case 5.1. W.l.o.g., expression e in predicate of S_1 evaluates to nonzero in state m, written $(e, m) \xrightarrow{*} (0, m)$.
By rule If-T, (If (0) then $\{S_t\}$ else $\{S_f\}, m) \to (S_t, m)$.
By contextual (semantic) rule,
(If (0) then $\{S_t\}$ else $\{S_f\}; S_2, m) \to (S_t; S_2, m)$.
Case 5.2. Evaluation of expression e in predicate of S_1 crashes, written $(e, m) \xrightarrow{*} (e', m(1/\mathfrak{f}))$.
Then, by rule crash,
(If (e') then $\{S_t\}$ else $\{S_f\}, m(1/\mathfrak{f})) \to$
(If (e') then $\{S_t\}$ else $\{S_f\}, m(1/\mathfrak{f}))$.
Then, by contextual rule,
(If (e') then $\{S_t\}$ else $\{S_f\}; S_2, m(1/\mathfrak{f})) \to$
(If (e') then $\{S_t\}$ else $\{S_f\}; S_2, m(1/\mathfrak{f}))$.
Case 6. S_1 = "while$_{\langle n \rangle}$ (e) $\{S\}$".
Case 6.1 When expression e in predicate of S_1 evaluates to nonzero value, written $(e, m) \xrightarrow{*} (v, m)$ for some $v \neq 0$, then, by rule Wh-T,
(while$_{\langle n \rangle}$ $(e)\{S\}, m) \to (S;$ while$_{\langle n \rangle}$ $(e)\{S\}, m(m_c[(k+1)/n]))$ for some nonnegative integer k.
Then, by contextual rule,
(while$_{\langle n \rangle}$ (e) $\{S\}; S_2, m) \to$
$(S;$ while $_{\langle n \rangle}$ (e) $\{S\}; S_2, m(m_c[(k+1)/n]))$.
Case 6.2 When expression e in predicate of S_1 evaluates to zero, written $(e, m) \xrightarrow{*} (0, m)$, then, by rule Wh-F,
(while$_{\langle n \rangle}$ $(e)\{S\}, m) \to (\text{skip}, m(m_c[0/n]))$.
By contextual rule,
(while$_{\langle n \rangle}$ (e) $\{S\}; S_2, m) \to (\text{skip}; S_2, m(m_c[0/n]))$.
Case 6.3 Evaluation of expression e in predicate of S_1 crashes, written $(e, m) \xrightarrow{*} (e', m)$.
By rule crash,
(while$_{\langle n \rangle}$ $(e')\{S\}, m(1/\mathfrak{f})) \to$ (while$_{\langle n \rangle}$ $(e')\{S\}, m(1/\mathfrak{f}))$.
Then, by contextual rule,

(while$_{\langle n\rangle}$ $(e')\{S\}; S_2, m(1/\mathfrak{f})) \rightarrow$
(while$_{\langle n\rangle}$ $(e')\{S\}; S_2, m(1/\mathfrak{f}))$. ☐

Lemma E.2. $(S_1, m) \xrightarrow{*} (S_1', m') \Rightarrow (S_1; S_2, m) \xrightarrow{*} (S_1'; S_2, m')$.

Proof. By induction on number of steps k in execution $(S_1, m) \xrightarrow{k} (S_1', m')$.

Base case. k = 0 and 1.
By definition,
$(S_1, m) \xrightarrow{0} (S_1, m)$, and $(S_1; S_2, m) \xrightarrow{0} (S_1; S_2, m)$.
By lemma E.1,
$(S_1, m) \rightarrow (S_1', m') \Rightarrow (S_1; S_2, m) \rightarrow (S_1'; S_2, m')$.
Induction step.
The induction hypothesis IH is that, for $k \geq 1$,
$(S_1, m) \xrightarrow{k} (S_1', m') \Rightarrow (S_1; S_2, m) \xrightarrow{k} (S_1'; S_2, m')$.
Then we show that,
$(S_1, m) \xrightarrow{k+1} (S_1', m') \Rightarrow (S_1; S_2, m) \xrightarrow{k+1} (S_1'; S_2, m')$.
We decompose the k+1 step execution into
$(S_1, m) \rightarrow (S_1'', m'') \xrightarrow{k} (S_1', m')$.
By lemma E.1,
$(S_1; S_2, m) \rightarrow (S_1''; S_2, m'')$.
Next, by IH,
$(S_1''; S_2, m'') \xrightarrow{k} (S_1'; S_2, m')$. ☐

Corollary E.1. $(S_1, m) \xrightarrow{*} (\textit{skip}, m') \Rightarrow (S_1; S_2, m) \xrightarrow{*} (S_2, m')$.

Proof. By lemma E.2,
$(S_1, m) \xrightarrow{*} (\text{skip}, m') \Rightarrow (S_1; S_2, m) \xrightarrow{*} (\text{skip}; S_2, m')$.
Then, by rule Seq,
$(\text{skip}; S_2, m') \rightarrow (S_2, m')$. ☐

Lemma E.3. *If one statement s is not in S, then, after one step of execution $(S, m) \rightarrow (S', m')$, s is not in the S', $(s \notin S) \wedge ((S, m) \rightarrow (S', m')) \Rightarrow (s \notin S')$.*

Proof. By induction on abstract syntax of S. ☐

Lemma E.4. *If one statement s is not in S, then, after the execution $(S, m) \xrightarrow{*} (S', m')$, s is not in the S', $(s \notin S) \wedge ((S, m) \xrightarrow{*} (S', m')) \Rightarrow (s \notin S')$.*

Proof. By induction on the number k of the steps in the execution $(S, m) \xrightarrow{k} (S', m')$. ☐

Lemma E.5. *If a variable x is not defined in a statement sequence S, then, after one step execution of S, the value of x is not redefined, $(x \notin \textit{Def}(S)) \wedge ((S, m(\sigma)) \rightarrow (S', m'(\sigma'))) \Rightarrow (x \notin \textit{Def}(S')) \wedge (\sigma'(x) = \sigma(x))$*

Proof. By structural induction on abstract syntax of statement sequence S, we show the lemma holds.

Case 1. S = "$id := e$".
By definition, Def(S) = $\{id\}$. Then $id \neq x$ by condition that $x \notin$ Def (S).
Then there are two subcases.

Case 1.1 Expression e evaluates to some value v, written $(e, m) \xrightarrow{*} (v, m)$.
Then, by rule Assign, $(S, m(\sigma)) \rightarrow (\text{skip}, m(\sigma[v/id]))$ where $m' = m(\sigma[v/id])$.
Hence, $\sigma'(x) = \sigma(x)$. Besides, $x \notin$ Def (skip) by definition.

Case 1.2 Evaluation of expression e crashes, written $(e, m) \xrightarrow{*} (e', m(1/\mathfrak{f}))$.
Then, by rule crash,
$(id := e', m(1/\mathfrak{f}, \sigma)) \rightarrow (id := e', m(1/\mathfrak{f}, \sigma))$ where $m' = m(1/\mathfrak{f}, \sigma)$.
Hence, $\sigma'(x) = \sigma(x)$. Besides, $x \notin$ Def$(id := e')$ by definition.

Case 2. S = " output e".
Case 3. S = " input id".
By similar argument in case 1.

Case 4. S= "If (e) then $\{S_t\}$ else $\{S_f\}$".
Def (S) = Def $(S_f) \cup$ Def (S_t) by definition. Then $x \notin$ Def $(S_f) \cup$ Def (S_t).
There are two subcases.

Case 4.1 W.l.o.g., expression e in predicate of S evaluates to nonzero value, written $(e, m) \xrightarrow{*} (v, m)$ where $v \neq 0$.
Then by rule If-T, (If (v) then $\{S_t\}$ else $\{S_f\}, m(\sigma)) \rightarrow (S_t, m(\sigma))$ where $m' = m$.
Therefore, $\sigma'(x) = \sigma(x)$. By argument above, $x \notin$ Def (S_t).

Case 4.2 Evaluation of expression e in predicate of S crashes, written $(e, m) \xrightarrow{*} (e', m(1/\mathfrak{f}))$.
Then, by rule crash,
(If (e') then $\{S_t\}$ else $\{S_f\}, m(1/\mathfrak{f}, \sigma)) \rightarrow$
(If (e') then $\{S_t\}$ else $\{S_f\}, m(1/\mathfrak{f}, \sigma))$ where $m' = m(1/\mathfrak{f}, \sigma)$.
Therefore, $\sigma'(x) = \sigma(x)$.
Besides, $x \notin$ Def (If (e') then $\{S_t\}$ else $\{S_f\}$) = Def $(S_f) \cup$ Def (S_t).

Case 5. S = "while$_{\langle n\rangle}$ (e) $\{S'\}$".
Def (S) = Def (S') by definition. Then $x \notin$ Def (S') by condition $x \notin$ Def (S).
There are subcases.

Case 5.1 Expression e evaluates to nonzero value, written $(e, m) \xrightarrow{*} (v, m)$ where $v \neq 0$.
By rule Wh-T, (while$_{\langle n\rangle}$ (v) $\{S'\}, m(\sigma)) \rightarrow$
$(S'$; while$_{\langle n\rangle}$ (e) $\{S'\}, m(m_c[(k+1)/n]), \sigma)$ for some non-negative integer k.
Let $m' = m(m_c[(k+1)/n], \sigma)$. Then $\sigma'(x)$ = $\sigma(x)$.
Besides, $x \notin$ Def $(S'$; while$_{\langle n\rangle}$ (e) $\{S'\})$ = Def $(S') \cup$ Def (S), because $x \notin$ Def (S').

Case 5.2 Expression e evaluates to zero in state m, written $(e, m) \xrightarrow{*} (0, m)$.
By rule Wh-F,
(while$_{\langle n\rangle}$ (0) $\{S'\}, m(\sigma)) \rightarrow (\text{skip}, m(m_c[0/n], \sigma))$ where $m' = m(m_c[0/n], \sigma)$.
Therefore, $\sigma'(x) = \sigma(x)$. Besides, $x \notin$ Def (skip).

Case 5.3 Evaluation of expression e crashes, written $(e, m) \xrightarrow{*} (e', m(1/\mathfrak{f}))$.
By rule crash
(while$_{\langle n\rangle}$ (e') $\{S'\}, m(1/\mathfrak{f}, \sigma)) \rightarrow$
(while$_{\langle n\rangle}$ (e') $\{S'\}, m(1/\mathfrak{f}, \sigma))$ where $m' = m(1/\mathfrak{f}, \sigma)$.
Therefore, $\sigma'(x) = \sigma(x)$. Besides, $x \notin$ Def(while$_{\langle n\rangle}$ (e') $\{S'\}$) = Def (S') by definition.

Case 6. S = $S_1; S_2$.
By argument in Case 1 to 5, after one step execution $((S_1, m(\sigma)) \rightarrow (S', m'(\sigma)))$, $\sigma'(x) = \sigma(x)$.
By contextual rule, the lemma holds.

☐

Corollary E.2. *If a variable x is not defined in a statement sequence S, then, after an execution of S, the value of x is not redefined, $(x \notin \textit{Def}(S) \wedge (S, m(\sigma)) \xrightarrow{*} (S', m'(\sigma')) \Rightarrow (x \notin \textit{Def}(S')) \wedge \sigma'(x) = \sigma(x))$.*

Proof. Let $(S, m) \xrightarrow{k} (S', m')$. The proof is by induction on k using lemma E.5. ☐

Based on Corollary E.2, we extend the result to array variable elements.

Corollary E.3. *If an element in an array variable $x[i]$ is not defined in a statement sequence S in a program $P = EN; V; S_{entry}$, then, after an execution of S, the value of $x[i]$ is not redefined, $(x \notin Def(S)) \wedge ((x, i, *) \in \sigma) \wedge (S, m(\sigma)) \xrightarrow{*} (S', m'(\sigma')) \Rightarrow (x \notin Def(S')) \wedge \sigma'(x, i) = \sigma(x, i))$.*

Lemma E.6. *If all of the following hold:*

1. *There is no loop of label n in statements S, "while$_{\langle n \rangle}(e)\{S'\}$" $\notin S$;*
2. *The crash flag is not set, $\mathfrak{f} = 0$;*
3. *There is an entry n in loop counter, $(n, *) \in loop_c$;*
4. *There is one step execution, $(S, m(\mathfrak{f}, loop_c)) \rightarrow (S', m'(loop'_c))$;*

then, $loop'_c(n) = loop_c(n)$.

Proof. The proof is by induction on abstract syntax of S, similar to that for lemma E.5. □

Corollary E.4. *If all of the following hold:*

1. *There is no loop of label n in statements S, "while$_{\langle n \rangle}(e)\{S'\}$" $\notin S$;*
2. *The crash flag is not set, $\mathfrak{f} = 0$;*
3. *There is an entry n in loop counter, $(n, *) \in loop_c$;*
4. *There is multiple steps execution of stack depth $d = 0$, $(S, m(\mathfrak{f}, loop_c)) \xrightarrow{*} (S', m'(loop'_c))$;*

then, $loop'_c(n) = loop_c(n)$.

Lemma E.7. *If all of the following hold:*

1. *A non-skip statement s is not in S, $(s \neq skip) \wedge (s \notin S)$;*
2. *There is one step execution of stack depth $d = 0$, $(S, m) \rightarrow (S', m')$,*

then, $s \notin S'$.

Proof. By structural induction on abstract syntax of statement sequence S, we show the lemma holds.

Case 1. S = "$id := e$".

Then there are two subcases.

Case 1.1 Expression e evaluates to some value v, written $(e, m) \xrightarrow{*} (v, m)$.

Then, by rule Assign, $(S, m) \rightarrow$ (skip, $m(\sigma[v/id])$).

Hence, $s \notin$ skip by definition.

Case 1.2 Evaluation of expression e crashes, written $(e, m) \xrightarrow{*} (e', m(1/\mathfrak{f}))$.

By parameterized type rule TExpr, $\Gamma \nvdash e'$. Then, by type rule TAssign, $\Gamma \nvdash id := e'$.

Then, by rule crash,
$(id := e', m(1/\mathfrak{f})) \rightarrow (id := e', m(1/\mathfrak{f}))$.

Because $\Gamma \vdash s$, then $s \neq id := e'$. Hence, $s \notin id := e'$ by definition.

Case 2. S = " output e".

Case 3. S = " input id".

By similar argument in case 1.

Case 4. S= "If (e) then $\{S_t\}$ else $\{S_f\}$".

$s \notin S_f, s \notin S_t$ by definition. There are two subcases.

Case 4.1 W.l.o.g., expression e in predicate of S evaluates to nonzero value, written $(e, m) \xrightarrow{*} (v, m)$ where $v \neq 0$.

Then by rule If-T, (If (v) then $\{S_t\}$ else $\{S_f\}, m) \rightarrow (S_t, m)$.

Therefore, $s \notin S_t$.

Case 4.2 Evaluation of expression e in predicate of S crashes, written $(e, m) \xrightarrow{*} (e', m(1/\mathfrak{f}))$.

Then, by rule crash,
(If (e') then $\{S_t\}$ else $\{S_f\}, m(1/\mathfrak{f})) \rightarrow$
(If (e') then $\{S_t\}$ else $\{S_f\}, m(1/\mathfrak{f})$).

By parameterized type rule TExpr, $\Gamma \nvdash e'$. By type rule Tif, $\Gamma \nvdash$ If (e') then $\{S_t\}$ else $\{S_f\}$.

Because $\Gamma \vdash s$, then $s \neq$ If (e') then $\{S_t\}$ else $\{S_f\}$.

Besides, $s \notin S_t, s \notin S_f$ by condition. Therefore, $s \notin$ If (e') then $\{S_t\}$ else $\{S_f\}$.

Case 5. S = "while$_{\langle n \rangle}$ (e) $\{S'\}$".

$s \notin S'$ by definition. There are subcases.

Case 5.1 Expression e evaluates to nonzero value, written $(e, m) \xrightarrow{*} (v, m)$ where $v \neq 0$.

By rule Wh-T, (while$_{\langle n \rangle}$ (v) $\{S'\}, m) \rightarrow$
$(S';$while$_{\langle n \rangle}$ (e) $\{S'\}, m(m_c[(k+1)/n]))$ for some nonnegative integer k.

Then $s \notin S';$while$_{\langle n \rangle}$ (e) $\{S'\}$ by definition.

Case 5.2 Expression e evaluates to zero in state m, written $(e, m) \xrightarrow{*} (0, m)$.

By rule Wh-F,
(while$_{\langle n \rangle}$ (0) $\{S'\}, m) \rightarrow (skip, m(m_c[0/n]))$.

Therefore, $s \notin$ skip.

Case 5.3 Evaluation of expression e crashes, written $(e, m) \xrightarrow{*} (e', m(1/\mathfrak{f}))$.

By rule crash,
(while$_{\langle n \rangle}$ (e') $\{S'\}, m(1/\mathfrak{f})) \rightarrow$
(while$_{\langle n \rangle}$ (e') $\{S'\}, m(1/\mathfrak{f})$).

Then, by type rule Twhile, $\Gamma \nvdash$ while$_{\langle n \rangle}$ (e') $\{S'\}$. Because $\Gamma \vdash s$, then $s \neq$ while$_{\langle n \rangle}$ (e') $\{S'\}$.

Besides $s \notin S'$, then $s \notin$ while$_{\langle n \rangle}$ (e') $\{S'\}$ by definition.

Case 6. $S = S_1; S_2$.

By argument in Case 1 to 5, after one step execution $(S_1, m) \rightarrow (S', m')$, $s \notin S'$.

By contextual rule, $(S_1; S_2, m) \rightarrow (S'; S_2, m')$.

By definition, $s \notin S_2$.

Then, by definition, $s \notin S'; S_2$

□

Lemma E.8. *Let s = "while$_{\langle n \rangle}$ (e) $\{S''\}$". If both of the following hold:*

- *$s \in S$;*
- *$(S, m(loop_c)) \rightarrow (S', m'(loop'_c))$;*

then one of the following holds:

1. *The loop counter of label n is incremented by one, $loop'_c(n) - loop_c(n)$ = 1;*
2. *There is no entry for label n in loop counter, $(n, v) \notin loop'_c$;*
3. *The loop counter of label n is not changed, $loop'_c(n) - loop_c(n)$ = 0;*

Proof. Let $S = s'; S''$. The proof is by induction on abstract syntax of s'. □