A Single-Assignment Translation for Annotated Programs

Cláudio Belo Lourenço, Maria João Frade, and
Jorge Sousa Pinto

HASLab/INESC TEC & Universidade do Minho, Portugal

Abstract. We present a translation of While programs annotated with loop invariants into a dynamic single-assignment language with a dedicated iterating construct. We prove that the translation is sound and complete. This is a companion report to our paper *Formalizing Single-assignment Program Verification: an Adaptation-complete Approach* [6].

1 Introduction

Deductive verification tools typically rely on the conversion of code to a dynamic single-assignment (SA) form. In [6] we formalize this approach by proposing a sound and complete technique based on the translation of annotated programs to such an intermediate form, and the generation of verification conditions from it. We introduce a notion of SA iterating program, as well as an *adaptation-complete* program logic (adequate for adapting specifications to local contexts) and an *efficient verification conditions* generator (in the sense of Flanagan and Saxe).

In this report we define a translation function that transforms an annotated program into SA from. The program produced conforms to the syntactic restrictions of an SA annotated program and preserves the operational semantics of the program in a sense that will be made precise later. Moreover, the translation function is lifted to Hoare triples, and preserves derivability in a goal-directed system of Hoare logic, i.e, if a Hoare triple for an annotated program is derivable, then the translated triple is also derivable in that system (with derivations guided by the translated loop invariants).

The core result of the paper, which we prove in full detail, is that the defined translation conforms to the requirements, identified in [6], for the generation of verification conditions to be sound and complete for the initial program. More precisely, if a Hoare triple containing an annotated While program is translated into SA form by our translation, and verification conditions for this SA triple are then generated, then (i) if these verification conditions are successfully discharged, then the original triple is valid (the program is correct); and (ii) if the original program is correct, and moreover it is correctly annotated (i.e. its annotations allow for the Hoare triple to be derived) then the verification conditions are all valid.

The report is organised as follows. In the remaining of this section we introduce some preliminary notation. In Section 2 we recall the necessary background material about Hoare logic. In Section 3 we define the translation function T_{sa}, and illustrate its use by means of an example in Section 4. In Section 5 we prove that T_{sa} is sound and preserves Hg-derivability. Section 6 concludes the report.

1.1 Preliminary notation

First of all let us introduce the notation used for functions. Given a function f, $\mathsf{dom}(f)$ denotes the domain of f, $\mathsf{rng}(f)$ denotes the codomain of f. As usual, $f[x \mapsto a]$ denotes the function that maps x to a and any other value y to $f(y)$.

For finite functions we also use the following notation. $[]$ represents the empty function (whose domain is $\emptyset$), and $[x_1 \mapsto a_1, \ldots, x_n \mapsto a_n]$ represents the finite function whose domain is $\{x_1, \ldots, x_n\}$ that maps each x_i to a_i. We also use the notation $[x \mapsto g(x) \mid x \in A]$ to represent the function with domain A generated by g.

Given a total function $f : X \to Y$ and a partial functions $g : X \rightharpoonup Y$, we define $f \oplus g : X \to Y$ as follows

$$(f \oplus g)(x) = \begin{cases} g(x) & \text{if } \ x \in \mathsf{dom}(g) \\ f(x) & \text{if } \ x \notin \mathsf{dom}(g) \end{cases}$$

i.e., g overrides f but leaves f unchanged at points where g is not defined.

2 Hoare logic

We will work with Hoare logic for simple While programs. The logic deals with the notion of correction w.r.t. a *specification* that consists of a *precondition* – an assertion that is assumed to hold when the execution of the program starts – and a *postcondition* – an assertion that is required to hold when execution stops.

2.1 Syntax

We consider a typical While language whose commands $C \in \mathbf{Comm}$ are defined over a set of variables $x \in \mathbf{Var}$ in the following way:

$$\mathbf{Comm} \ni C \ ::= \ \mathbf{skip} \mid x := e \mid C\,;C \mid \mathbf{if}\ b\ \mathbf{then}\ C\ \mathbf{else}\ C \mid \mathbf{while}\ b\ \mathbf{do}\ C$$

We will not fix the language of program expressions $e \in \mathbf{Exp}$ and Boolean expressions $b \in \mathbf{Exp^{bool}}$, both constructed over variables from $\mathbf{Var}$ (a standard instantiation is for $\mathbf{Exp}$ to be a language of integer expressions and $\mathbf{Exp^{bool}}$ constructed from comparison operators over $\mathbf{Exp}$, together with Boolean operators). In addition to expressions and commands, we need formulas that express properties of particular states of the program. Program assertions $\phi, \theta, \psi \in \mathbf{Assert}$ (preconditions and postconditions in particular) are formulas of a first-order language obtained as an expansion of $\mathbf{Exp^{bool}}$.

1. $\langle \mathbf{skip}, s\rangle \rightsquigarrow s$
2. $\langle x := e, s\rangle \rightsquigarrow s[x \mapsto [\![e]\!](s)]$
3. if $\langle C_1, s\rangle \rightsquigarrow s'$ and $\langle C_2, s'\rangle \rightsquigarrow s''$, then $\langle C_1 \,;\, C_2, s\rangle \rightsquigarrow s''$
4. if $[\![b]\!](s) = \mathbf{T}$ and $\langle C_t, s\rangle \rightsquigarrow s'$, then $\langle \mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f, s\rangle \rightsquigarrow s'$
5. if $[\![b]\!](s) = \mathbf{F}$ and $\langle C_f, s\rangle \rightsquigarrow s'$, then $\langle \mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f, s\rangle \rightsquigarrow s'$
6. if $[\![b]\!](s) = \mathbf{T}$, $\langle C, s\rangle \rightsquigarrow s'$ and $\langle \mathbf{while}\ b\ \mathbf{do}\ C, s'\rangle \rightsquigarrow s''$, then $\langle \mathbf{while}\ b\ \mathbf{do}\ C, s\rangle \rightsquigarrow s''$
7. if $[\![b]\!](s) = \mathbf{F}$, then $\langle \mathbf{while}\ b\ \mathbf{do}\ C, s\rangle \rightsquigarrow s$

Fig. 1. Evaluation semantics for While programs

We also require a class of formulas for specifying the behaviour of programs. Specifications are pairs (ϕ, ψ), with $\phi, \psi \in \mathbf{Assert}$ intended as precondition and postcondition for a program. The precondition is an assertion that is assumed to hold when the program is executed, whereas the postcondition is required to hold when its execution stops. A *Hoare triple*, written as $\{\phi\}\, C\, \{\psi\}$, expresses the fact that the program C conforms to the specification (ϕ, ψ).

2.2 Semantics

We will consider an *interpretation structure* $\mathcal{M} = (D, I)$ for the vocabulary describing the concrete syntax of program expressions. This structure provides an interpretation domain D as well as a concrete interpretation of constants and operators, given by I. The interpretation of expressions depends on a *state*, which is a function that maps each variable into its value. We will write $\Sigma = \mathbf{Var} \to D$ for the set of states (note that this approach extends to a multi-sorted setting by letting Σ become a *generic function space*). For $s \in \Sigma$, $s[x \mapsto a]$ will denote the state that maps x to a and any other variable y to $s(y)$. The interpretation of $e \in \mathbf{Exp}$ will be given by a function $[\![e]\!]_{\mathcal{M}} : \Sigma \to D$, and the interpretation of $b \in \mathbf{Exp}^{\mathbf{bool}}$ will be given by $[\![b]\!]_{\mathcal{M}} : \Sigma \to \{\mathbf{F}, \mathbf{T}\}$. This reflects our assumption that an expression has a value at every state (evaluation always terminates without error) and that expression evaluation never changes the state (the language is free of *side effects*). For the interpretation of assertions we take the usual interpretation of first-order formulas, noting two facts: since assertions build on the language of program expressions their interpretation also depends on $\mathcal{M}$ (possibly extended to account for user-defined predicates and functions), and states from Σ can be used as *variable assignments* in the interpretation of assertions. The interpretation of the assertion $\phi \in \mathbf{Assert}$ is then given by $[\![\phi]\!]_{\mathcal{M}} : \Sigma \to \{\mathbf{F}, \mathbf{T}\}$, and we will write $s \models \phi$ as a shorthand for $[\![\phi]\!]_{\mathcal{M}}(s) = \mathbf{T}$. In the rest of the paper we will omit the $\mathcal{M}$ subscripts for the sake of readability; the interpretation structure will be left implicit.

For commands, we consider a standard operational, natural style semantics, based on a deterministic *evaluation relation* $\rightsquigarrow \subseteq \mathbf{Comm} \times \Sigma \times \Sigma$ (which again

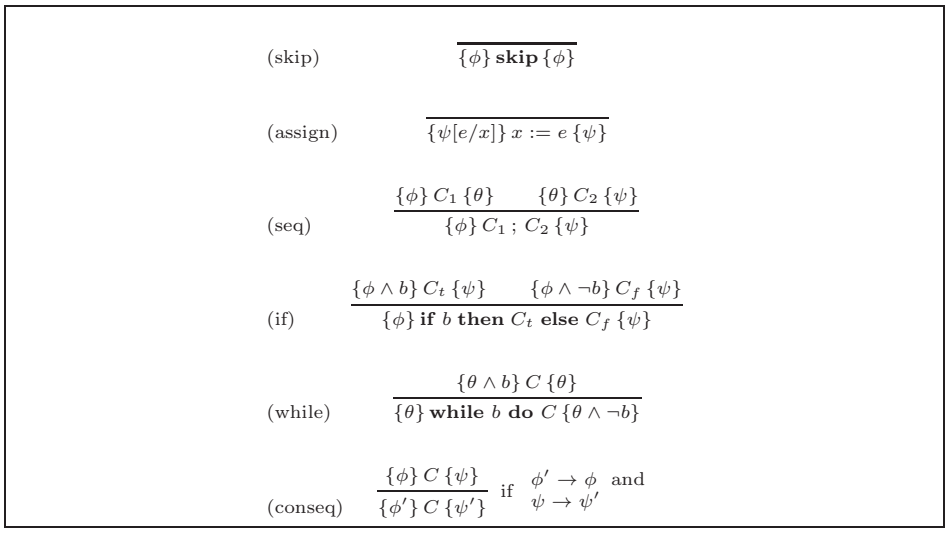

Fig. 2. System H

depends on an implicit interpretation of program expressions). We will write $\langle C, s\rangle \rightsquigarrow s'$ to denote the fact that if C is executed in the initial state s, then its execution terminates, and the final state is s'. The usual inductive definition of this relation is given in Figure 1.

The intuitive meaning of the triple $\{\phi\}\,C\,\{\psi\}$ is that if the program C is executed in an initial state in which the precondition ϕ is true, then either execution of C does not terminate or if it does, the postcondition ψ will be true in the final state. Because termination is not guaranteed, this is called a *partial correctness* specification. Let us now define formally the notion of validity for such a triple.

Definition 1. *The Hoare triple* $\{\phi\}\,C\,\{\psi\}$ *is said to be* valid*, denoted* $\models \{\phi\}\,C\,\{\psi\}$*, whenever for all* $s, s' \in \Sigma$*, if* $s \models \phi$ *and* $\langle C, s\rangle \rightsquigarrow s'$*, then* $s' \models \psi$*.*

2.3 Hoare Calculus

Hoare [5] introduced an inference system for reasoning about Hoare triples, which we will call system H - see Figure 3. Note that the system contains one rule (conseq) whose application is guarded by first-order conditions. We will consider that reasoning in this system takes place in the context of the *complete theory* $\mathrm{Th}(\mathcal{M})$ of the implicit structure $\mathcal{M}$, so that when constructing derivations in H one simply checks, when applying the (conseq) rule, whether the side conditions are elements of $\mathrm{Th}(\mathcal{M})$. We will write $\vdash_{\mathsf{H}} \{\phi\}\,C\,\{\psi\}$ to denote the fact that the triple is derivable in this system with $\mathrm{Th}(\mathcal{M})$.

System H is sound w.r.t. the semantics of Hoare triples; it is also complete as long as the assertion language is sufficiently expressive (a result due to Cook [2]). One way to ensure this is to force the existence of a *strongest post-condition* for every command and assertion. Let $C \in \mathbf{Comm}$ and $\phi \in \mathbf{Assert}$, and denote by $\mathrm{post}(\phi, C)$ the set of states $\{s' \in \Sigma \mid \langle C, s\rangle \rightsquigarrow s'$ for some $s \in \Sigma$ such that $[\![\phi]\!](s) = \mathbf{T}\}$. In what follows we will assume that the assertion language **Assert** is *expressive* with respect to the command language **Comm** and interpretation structure $\mathcal{M}$, i.e., for every $\phi \in \mathbf{Assert}$ and $C \in \mathbf{Comm}$ there exists $\psi \in \mathbf{Assert}$ such that $s \models \psi$ iff $s \in \mathrm{post}(\phi, C)$ for any $s \in \Sigma$. The reader is directed to [1] for details.

Proposition 1 (Soundness of system H). *Let $C \in \mathbf{Comm}$ and $\phi, \psi \in \mathbf{Assert}$. If $\vdash_{\mathsf{H}} \{\phi\}\, C\, \{\psi\}$, then $\models \{\phi\}\, C\, \{\psi\}$.*

Proposition 2 (Completeness of system H). *Let $C \in \mathbf{Comm}$ and $\phi, \psi \in \mathbf{Assert}$. With* **Assert** *expressive in the above sense, if $\models \{\phi\}\, C\, \{\psi\}$, then $\vdash_{\mathsf{H}} \{\phi\}\, C\, \{\psi\}$.*

Let $\mathsf{FV}(\phi)$ denote the set of free variables occurring in ϕ. The sets of *variables occurring* and *assigned* in the program C will be given by $\mathsf{Vars}(C)$ and $\mathsf{Asgn}(C)$ according to the next definition.

Definition 2. *Let $\mathsf{Vars}(e)$ be the set of variables occurring in expression e, and $C \in \mathbf{Comm}$.*

- *The set of* variables occurring *in C, $\mathsf{Vars}(C)$, is defined as follows:*

$$\begin{aligned}
\mathsf{Vars}(\mathbf{skip}) &= \emptyset \\
\mathsf{Vars}(x := e) &= \{x\} \cup \mathsf{Vars}(e) \\
\mathsf{Vars}(C_1\,;\, C_2) &= \mathsf{Vars}(C_1) \cup \mathsf{Vars}(C_2) \\
\mathsf{Vars}(\mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f) &= \mathsf{Vars}(b) \cup \mathsf{Vars}(C_t) \cup \mathsf{Vars}(C_f) \\
\mathsf{Vars}(\mathbf{while}\ b\ \mathbf{do}\ C) &= \mathsf{Vars}(b) \cup \mathsf{Vars}(C)
\end{aligned}$$

- *The set of* variables assigned *in C, $\mathsf{Asgn}(C)$, is defined as follows:*

$$\begin{aligned}
\mathsf{Asgn}(\mathbf{skip}) &= \emptyset \\
\mathsf{Asgn}(x := e) &= \{x\} \\
\mathsf{Asgn}(C_1\,;\, C_2) &= \mathsf{Asgn}(C_1) \cup \mathsf{Asgn}(C_2) \\
\mathsf{Asgn}(\mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f) &= \mathsf{Asgn}(C_t) \cup \mathsf{Asgn}(C_f) \\
\mathsf{Asgn}(\mathbf{while}\ b\ \mathbf{do}\ C) &= \mathsf{Asgn}(C)
\end{aligned}$$

- *We will write $\phi \# C$ to denote the fact that C does not assign the variables occurring free in ϕ, i.e. $\mathsf{Asgn}(C) \cap \mathsf{FV}(\phi) = \emptyset$.*

Triples in which the program do not assign variables from the precondition enjoy the following property in H:

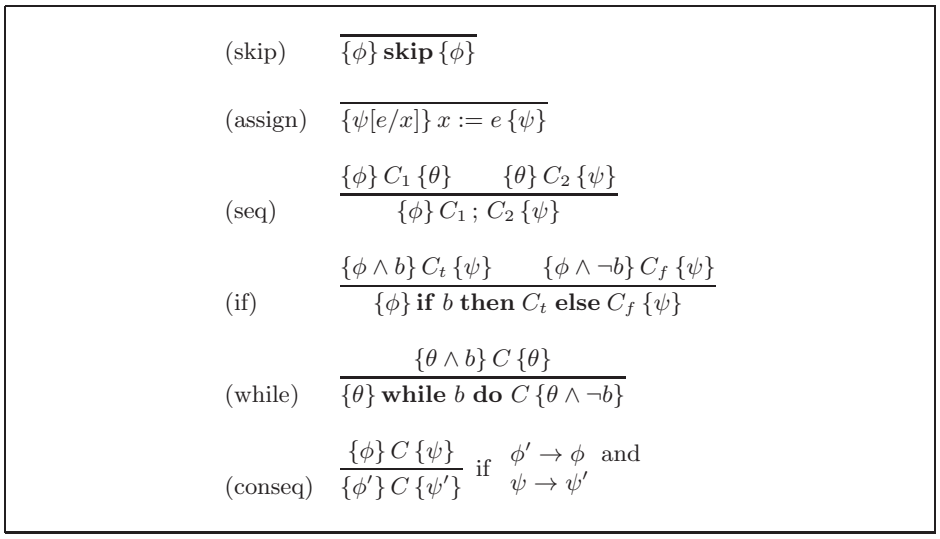

Fig. 3. Systems H

Lemma 1. *Let $\phi, \psi \in$ **Assert** and $C \in$ **Comm**, such that $\phi \# C$. If $\vdash_{\mathsf{H}} \{\phi\}\,C\,\{\psi\}$, then $\vdash_{\mathsf{H}} \{\phi\}\,C\,\{\phi \wedge \psi\}$.*

Proof. By induction on the structure of C one proves that $\vdash_{\mathsf{H}} \{\phi\}\,C\,\{\phi\}$. Then it follows from $\vdash_{\mathsf{H}} \{\phi\}\,C\,\{\psi\}$, by the conjunction assertions rule (see for instance [7]), that $\vdash_{\mathsf{H}} \{\phi \wedge \phi\}\,C\,\{\phi \wedge \psi\}$. We conclude by applying (conseq). □

2.4 Goal-directed logic

We introduce a syntactic class **AComm** of *annotated programs*, which differs from **Comm** only in the case of while commands, which are of the form **while** b **do** $\{\theta\}\,C$ where the assertion θ is an annotated loop invariant. Annotations do not affect the operational semantics. Note that for $C \in$ **AComm**, $\mathsf{Vars}(C)$ includes the free variables of the annotations in C. In what follows we will use the auxiliary function $\lfloor\cdot\rfloor$ that erases all annotations from a program defined in the following definition.

Definition 3. *The function $\lfloor\cdot\rfloor$: **AComm** $\to$ **Comm** is defined as follows:*

$$\begin{aligned}
\lfloor\mathbf{skip}\rfloor &= \mathbf{skip}\\
\lfloor x := e\rfloor &= x := e\\
\lfloor C_1\,;\,C_2\rfloor &= \lfloor C_1\rfloor\,;\,\lfloor C_2\rfloor\\
\lfloor\mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f\rfloor &= \mathbf{if}\ b\ \mathbf{then}\ \lfloor C_t\rfloor\ \mathbf{else}\ \lfloor C_f\rfloor\\
\lfloor\mathbf{while}\ b\ \mathbf{do}\,\{\theta\}\,C\rfloor &= \mathbf{while}\ b\ \mathbf{do}\ \lfloor C\rfloor
\end{aligned}$$

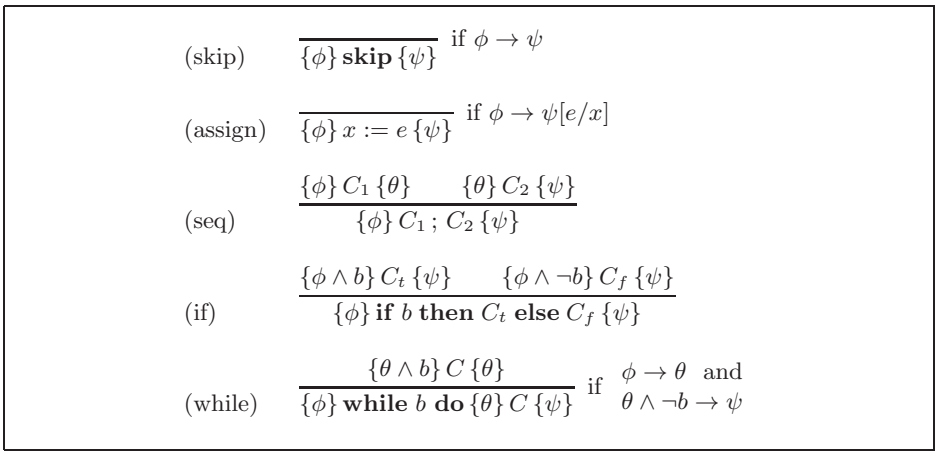

Fig. 4. System Hg

In Figure 4 we present system Hg, a *goal-directed* version of Hoare logic for triples containing annotated programs. This system is intended for mechanical construction of derivations: loop invariants are not invented but taken from the annotations, and there is no ambiguity in the choice of rule to apply, since a consequence rule is not present. The different derivations of the same triple in Hg differ only in the intermediate assertions used.

The following can be proved by induction on the derivation of $\vdash_{\mathsf{Hg}} \{\phi\}\,C\,\{\psi\}$.

Proposition 3 (Soundness of Hg). *Let* $C \in$ **AComm** *and* $\phi, \psi \in$ **Assert**. *If* $\vdash_{\mathsf{Hg}} \{\phi\}\,C\,\{\psi\}$ *then* $\vdash_{\mathsf{H}} \{\phi\}\,\lfloor C \rfloor\,\{\psi\}$.

The converse implication does not hold, since the annotated invariants may be inadequate for deriving the triple. Instead we need the following definition:

Definition 4. *Let* $C \in$ **AComm** *and* $\phi, \psi \in$ **Assert**. *We say that* C *is* correctly-annotated *w.r.t.* (ϕ, ψ) *if* $\vdash_{\mathsf{H}} \{\phi\}\,\lfloor C \rfloor\,\{\psi\}$ *implies* $\vdash_{\mathsf{Hg}} \{\phi\}\,C\,\{\psi\}$.

The following lemma states the admissibility of the consequence rule in Hg.

Lemma 2. *Let* $C \in$ **AComm** *and* $\phi, \psi, \phi', \psi' \in$ **Assert** *such that* $\vdash_{\mathsf{Hg}} \{\phi\}\,C\,\{\psi\}$, $\models \phi' \to \phi$, *and* $\models \psi \to \psi'$. *Then* $\vdash_{\mathsf{Hg}} \{\phi'\}\,C\,\{\psi'\}$.

It is possible to write an algorithm, known as a *verification conditions generator* (VCGen), that simply collects the side conditions of a derivation without actually constructing it. Hg is agnostic with respect to a strategy for propagating assertions, but the VCGen necessarily imposes one such strategy [4].

2.5 Single-assignment programs

Translation of code into Single-Assignment (SA) form has been part of the standard compilation pipeline for decades now; in such a program each variable is

assigned at most once. The fragment $x := 10\,;\, x := x+10$ could be translated as $x_1 := 10\,;\, x_2 := x_1+10$, using a different "version of x" variable for each assignment. In this paper we will use a dynamic notion of single-assignment (DSA) program, in which each variable may occur syntactically as the left-hand side of more than one assignment instruction, as long as it is not assigned more than once *in each execution*. For instance the fragment **if** $x > 0$ **then** $x := x + 10$ **else skip** could be translated into DSA form as **if** $x_0 > 0$ **then** $x_1 := x_0+10$ **else** $x_1 := x_0$. Note that the *else* branch cannot be simply **skip**, since it is necessary to have a single version variable (in this case x_1) representing x when exiting the conditional.

In the context of the guarded commands language, it has been shown that verification conditions for *passive programs* (essentially DSA programs without loops) can be generated avoiding the exponential explosion problem. However, a proper single-assignment imperative language in which programs with loops can be expressed for the purpose of verification does not exist. In what follows we will introduce precisely such a language based on DSA form.

Definition 5. *The set* $\mathbf{Rnm} \subseteq \mathbf{Comm}$ *of* renamings *consists of all programs of the form* $\{x_1 := y_1\,;\,\ldots\,;\,x_n := y_n\}$ *such that all* x_i *and* y_i *are distinct.*

A renaming $\mathcal{R} = \{x_1 := y_1\,;\,\ldots\,;\,x_n := y_n\}$ represents the finite bijection $[x_1 \mapsto y_1, \ldots, x_n \mapsto y_n]$, which we will also denote by $\mathcal{R}$. Furthermore, $\mathcal{R}(\phi)$ will denote the assertion that results from applying the substitution $[y_1/x_1, \ldots, y_n/x_n]$ to ϕ. Also, for $s \in \Sigma$ we define the state $\mathcal{R}(s)$ as follows: $\mathcal{R}(s)(x) = s(\mathcal{R}(x))$ if $x \in \mathsf{dom}(\mathcal{R})$, and $\mathcal{R}(s)(x) = s(x)$ otherwise.

Lemma 3. *Let* $\mathcal{R} \in \mathbf{Rnm}$, $\phi, \psi \in \mathbf{Assert}$ *and* $s \in \Sigma$.

1. $\langle \mathcal{R}, s\rangle \rightsquigarrow \mathcal{R}(s)$
2. $[\![\mathcal{R}(\phi)]\!](s) = [\![\phi]\!](\mathcal{R}(s))$
3. $\models \{\phi\}\,\mathcal{R}\,\{\psi\}$ *iff* $\models \phi \to \mathcal{R}(\psi)$.

Proof. 1. By inspection on the evaluation relation. 2. By structural induction on the interpretation assertions. 3. Follows directly from 1 and 2. □

Definition 6. *Let* $\mathbf{AComm}^{\text{SA}}$ *be the class of* annotated single-assignment programs. *Its abstract syntax is defined by*

$$C \;::=\; \mathbf{skip} \mid C\,;\,C \mid x := e \mid \mathbf{if}\ b\ \mathbf{then}\ C\ \mathbf{else}\ C \mid \mathbf{for}\ (\mathcal{I}, b, \mathcal{U})\ \mathbf{do}\,\{\theta\}\,C$$

with the following restrictions:

- $\mathbf{skip} \in \mathbf{AComm}^{\text{SA}}$
- $x := e \in \mathbf{AComm}^{\text{SA}}$ *if* $x \notin \mathsf{Vars}(e)$
- $C_1\,;\,C_2 \in \mathbf{AComm}^{\text{SA}}$ *if* $C_1, C_2 \in \mathbf{AComm}^{\text{SA}}$ *and* $\mathsf{Vars}(C_1) \cap \mathsf{Asgn}(C_2) = \emptyset$
- **if** b **then** C_t **else** $C_f \in \mathbf{AComm}^{\text{SA}}$ *if* $C_t, C_f \in \mathbf{AComm}^{\text{SA}}$ *and* $\mathsf{Vars}(b) \cap (\mathsf{Asgn}(C_t) \cup \mathsf{Asgn}(C_f)) = \emptyset$
- **for** $(\mathcal{I}, b, \mathcal{U})$ **do** $\{\theta\}\,C \in \mathbf{AComm}^{\text{SA}}$ *if* $C \in \mathbf{AComm}^{\text{SA}}$, $\mathcal{I}, \mathcal{U} \in \mathbf{Rnm}$, $\mathsf{Asgn}(\mathcal{I}) = \mathsf{Asgn}(\mathcal{U})$, $\mathsf{rng}(\mathcal{U}) \subseteq \mathsf{Asgn}(C)$, *and* $(\mathsf{Vars}(\mathcal{I}) \cup \mathsf{Vars}(b) \cup \mathsf{FV}(\theta)) \cap \mathsf{Asgn}(C) = \emptyset$

where the sets of variables occurring and assigned in the program are extended with the case of for-commands as follows:

$$\begin{aligned}\mathsf{Vars}(\textbf{for } (\mathcal{I},b,\mathcal{U}) \textbf{ do } \{\theta\}\, C) &= \mathsf{Vars}(\mathcal{I}) \cup \mathsf{Vars}(b) \cup \mathsf{FV}(\theta) \cup \mathsf{Vars}(C)\\ \mathsf{Asgn}(\textbf{for } (\mathcal{I},b,\mathcal{U}) \textbf{ do } \{\theta\}\, C) &= \mathsf{Asgn}(\mathcal{I}) \cup \mathsf{Asgn}(C)\end{aligned}$$

Note that the definition of $\phi\#C$ is naturally lifted to annotated programs.

The definition is straightforward except in the case of loops. In a strict sense it is not possible to write iterating programs in DSA form. So what we propose here is a syntactically controlled violation of the single-assignment constraints that allows for structured reasoning. Loop bodies are SA blocks, but loops contain a renaming $\mathcal{U}$ (to be executed after the body) that is free from single-assignment restrictions. The idea is that the "initial version" variables of the loop body (the ones used in the loop condition) are updated by the $\mathcal{U}$ renaming, which transports values from one iteration to the next, and is allowed to assign variables already assigned in $\mathcal{I}$ or occurring in C or b. The initialization code $\mathcal{I}$ on the other hand contains a renaming assignment that runs exactly once, even if no iterations take place. This ensures that the initial version variables always contain the output values of the loop: $\mathcal{U}$ is always executed after every iteration, and in the case of zero iterations they have been initialized by $\mathcal{I}$.

Consider the factorial program shown below on the left. The counter i ranges from 1 to n and the accumulator f contains at each step the factorial of $i-1$. The program is annotated with an appropriate loop invariant; it is easy to show that the program is correct w.r.t. the specification $(n \geq 0 \wedge n = n_{aux}, f = n_{aux}!)$.

$f := 1\,;$
$i := 1\,;$
while $i \leq n$ **do** $\{f = (i-1)! \wedge i \leq n+1\}$
{
  $f := f * i\,;$
  $i := i + 1$
}

$f_1 := 1\,;\ i_1 := 1\,;$
$\mathcal{I}$
while $(i_{a0} \leq n)$ **do**
{
  $f_{a1} := f_{a0} * i_{a0}\,;\ i_{a1} := i_{a0} + 1\,;$
  $\mathcal{U}$
}

On the right we show the same code with the blocks converted to SA form. The variables in the loop are indexed with an 'a', and then sequentially indexed with integer numbers as assignments take place (any fresh variable names would do). The initial version variables of the loop body f_{a0} and i_{a0} are the ones used in the Boolean expression, which is evaluated at the beginning of each iteration. We have placed in the code the required renamings $\mathcal{I}$ and $\mathcal{U}$, and it should be easy to instantiate them. $\mathcal{I}$ should be defined as $i_{a0} := i_1\,;\ f_{a0} := f_1$, and $\mathcal{U}$ as $i_{a0} := i_{a1}\,;\ f_{a0} := f_{a1}$. Note that without $\mathcal{U}$ the new values of the counter and of the accumulator would not be transported to the next iteration. The initial version variables i_{a0} and f_{a0} are the ones to be used after the loop to access the value of the counter and accumulator. A specification for this program could be written as $(n \geq 0 \wedge n = n_{aux}, f_{a0} = n_{aux}!)$.

It is straightforward to convert this pseudo-SA code to a program that is in accordance with Definition 6, with a *for* command encapsulating the structure of

the SA loop. The required invariant annotation uses the initial version variables:

$f_1 := 1\,;\, i_1 := 1\,;$
for $(\{i_{a0} := i_1\,;\, f_{a0} := f_1\}, i_{a0} \leq n, \{i_{a0} := i_{a1}\,;\, f_{a0} := f_{a1}\})$ **do** $\{f_{a0} = (i_{a0} - 1)! \wedge i_{a0} \leq n + 1\}$
{
  $f_{a1} := f_{a0} * i_{a0}\,;\, i_{a1} := i_{a0} + 1$
}

Definition 7. *The function* $\mathcal{W} : \mathbf{AComm}^{\text{SA}} \to \mathbf{AComm}$ *translates SA programs to (annotated) While programs as follows:*

$$\begin{aligned}
\mathcal{W}(\mathbf{skip}) &= \mathbf{skip} \\
\mathcal{W}(x := e) &= x := e \\
\mathcal{W}(C_1\,;\, C_2) &= \mathcal{W}(C_1)\,;\, \mathcal{W}(C_2) \\
\mathcal{W}(\mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f) &= \mathbf{if}\ b\ \mathbf{then}\ \mathcal{W}(C_t)\ \mathbf{else}\ \mathcal{W}(C_f) \\
\mathcal{W}(\mathbf{for}\ (\mathcal{I}, b, \mathcal{U})\ \mathbf{do}\ \{\theta\}\ C) &= \mathcal{I}\,;\, \mathbf{while}\ b\ \mathbf{do}\ \{\theta\}\ \{\mathcal{W}(C)\,;\, \mathcal{U}\}
\end{aligned}$$

A translation of annotated programs into SA form (as ilustrated by the factorial example) must of course abide by the syntactic restrictions of $\mathbf{AComm}^{\text{SA}}$, with additional requirements of a semantic nature. In particular, the translation will annotate the SA program with loop invariants (produced from those contained in the original program), and Hg-derivability guided by these annotations must be preserved. On the other hand, the translation must be sound: it will not translate invalid triples into valid triples. Both these notions are expressed by translating back to While programs.

Definition 8 (SA translation). *A function* $\mathcal{T} : \mathbf{Assert} \times \mathbf{AComm} \times \mathbf{Assert} \to \mathbf{Assert} \times \mathbf{AComm}^{\text{SA}} \times \mathbf{Assert}$ *is said to be a* single-assignment translation *if when* $\mathcal{T}(\phi, C, \psi) = (\phi', C', \psi')$ *we have* $\phi' \# C'$*, and the following both hold:*

1. *If* $\models \{\phi'\}\, \lfloor \mathcal{W}(C') \rfloor\, \{\psi'\}$*, then* $\models \{\phi\}\, \lfloor C \rfloor\, \{\psi\}$.
2. *If* $\vdash_{\mathsf{Hg}} \{\phi\}\, C\, \{\psi\}$*, then* $\vdash_{\mathsf{Hg}} \{\phi'\}\, \mathcal{W}(C')\, \{\psi'\}$.

The remmaning of this report is devoted to the definition of a concrete SA translation function and to proving that the function defined conforms Definition 8.

3 A translation to single-assigment form

In this section we define a translation function that transforms an annotated program into SA form. We start by introducing some auxiliary definitions to deal with variable versions. Without loss of generality, we will assume that the universe of variables of the SA programs consists of two parts: the *variable identifier* and a *version*. A version is a non-empty list of positive numbers. We let $\mathbf{Var}^{\text{SA}} = \mathbf{Var} \times \mathbb{N}^+$ be the set of SA variables, and we will write x_l to denote $(x, l) \in \mathbf{Var}^{\text{SA}}$. We write $\Sigma^{\text{SA}} = \mathbf{Var}^{\text{SA}} \to D$ for the set of states, with D being the interpretation domain.

$$
\begin{aligned}
T_{\mathsf{sa}} &: (\mathbf{Var} \to \mathbb{N}^+) \times \mathbf{AComm} \to (\mathbf{Var} \to \mathbb{N}^+) \times \mathbf{AComm}^{\mathrm{SA}} \\
T_{\mathsf{sa}}(\mathcal{V}, \mathbf{skip}) &= (\mathcal{V}, \mathbf{skip}) \\
T_{\mathsf{sa}}(\mathcal{V}, x := e) &= (\mathcal{V}[x \mapsto \mathsf{next}(\mathcal{V}(x))], x_{\mathsf{next}(\mathcal{V}(x))} := \widehat{\mathcal{V}}(e)) \\
T_{\mathsf{sa}}(\mathcal{V}, C_1; C_2) &= (\mathcal{V}'', C_1'; C_2') \\
&\quad \text{where } (\mathcal{V}', C_1') = T_{\mathsf{sa}}(\mathcal{V}, C_1) \\
&\qquad\quad (\mathcal{V}'', C_2') = T_{\mathsf{sa}}(\mathcal{V}', C_2) \\
T_{\mathsf{sa}}(\mathcal{V}, \mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f) &= (\mathsf{sup}(\mathcal{V}', \mathcal{V}''), \mathbf{if}\ \widehat{\mathcal{V}}(b)\ \mathbf{then}\ C_t'; \mathsf{merge}(\mathcal{V}', \mathcal{V}'')\ \mathbf{else}\ C_f'; \mathsf{merge}(\mathcal{V}'', \mathcal{V}')) \\
&\quad \text{where } (\mathcal{V}', C_t') = T_{\mathsf{sa}}(\mathcal{V}, C_t) \\
&\qquad\quad (\mathcal{V}'', C_f') = T_{\mathsf{sa}}(\mathcal{V}, C_f) \\
T_{\mathsf{sa}}(\mathcal{V}, \mathbf{while}\ b\ \mathbf{do}\ \{\theta\}\ C) &= (\mathcal{V}''', \mathbf{for}\ (\mathcal{I}, \widehat{\mathcal{V}'}(b), \mathcal{U})\ \mathbf{do}\ \{\widehat{\mathcal{V}'}(\theta)\}\ \{C'; \mathsf{upd}(\mathsf{dom}(\mathcal{U}))) \\
&\quad \text{where } \mathcal{I} = [x_{\mathsf{new}(\mathcal{V}(x))} := x_{\mathcal{V}(x)} \mid x \in \mathsf{Asgn}(C)] \\
&\qquad\quad \mathcal{V}' = \mathcal{V}[x \mapsto \mathsf{new}(\mathcal{V}(x)) \mid x \in \mathsf{Asgn}(C)] \\
&\qquad\quad (\mathcal{V}'', C') = T_{\mathsf{sa}}(\mathcal{V}', C) \\
&\qquad\quad \mathcal{U} = [x_{\mathsf{new}(\mathcal{V}(x))} := x_{\mathcal{V}''(x)} \mid x \in \mathsf{Asgn}(C)] \\
&\qquad\quad \mathcal{V}''' = \mathcal{V}''[x \mapsto \mathsf{jump}(l) \mid x_l \in \mathsf{dom}(\mathcal{U})]
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{next} : \mathbb{N}^+ \to \mathbb{N}^+ \\
&\mathsf{next}\ (h : t) = (h+1) : t \\
&\mathsf{new} : \mathbb{N}^+ \to \mathbb{N}^+ \\
&\mathsf{new}\ l = 1 : l \\
&\mathsf{jump} : \mathbb{N}^+ \to \mathbb{N}^+ \\
&\mathsf{jump}\ (i : j : t) = (j+1) : t \\
&(h : t) \prec (h' : t') = h < h'
\end{aligned}
\qquad
\begin{aligned}
&\mathsf{sup} : (\mathbf{Var} \to \mathbb{N}^+)^2 \to (\mathbf{Var} \to \mathbb{N}^+) \\
&\mathsf{sup}\ (\mathcal{V}, \mathcal{V}')(x) = \begin{cases} \mathcal{V}(x) & \text{if } \mathcal{V}'(x) \prec \mathcal{V}(x) \\ \mathcal{V}'(x) & \text{otherwise} \end{cases} \\
&\mathsf{merge} : (\mathbf{Var} \to \mathbb{N}^+)^2 \to \mathbf{Rnm} \\
&\mathsf{merge}\ (\mathcal{V}, \mathcal{V}') = [x_{\mathcal{V}'(x)} := x_{\mathcal{V}(x)} \mid x \in \mathbf{Var} \wedge \mathcal{V}(x) \prec \mathcal{V}'(x)] \\
&\mathsf{upd} : \mathcal{P}(\mathbf{Var}^{\mathrm{SA}}) \to \mathbf{Rnm} \\
&\mathsf{upd}\ (X) = [x_{\mathsf{jump}(l)} := x_l \mid x_l \in X]
\end{aligned}
$$

$$
\begin{aligned}
T_{\mathsf{sa}} &: \mathbf{Assert} \times \mathbf{AComm} \times \mathbf{Assert} \to \mathbf{Assert}^{\mathrm{SA}} \times \mathbf{AComm}^{\mathrm{SA}} \times \mathbf{Assert}^{\mathrm{SA}} \\
T_{\mathsf{sa}}(\phi, C, \psi) &= (\widehat{\mathcal{V}}(\phi), C', \widehat{\mathcal{V}'}(\psi)) \\
&\quad \text{where } (\mathcal{V}', C') = T_{\mathsf{sa}}(\mathcal{V}, C) \text{ , for some } \mathcal{V} \in \mathbf{Var} \to \mathbb{N}^+
\end{aligned}
$$

Fig. 5. SA translation function

Consider the *version function* $\mathcal{V} : \mathbf{Var} \to \mathbb{N}^+$. The function $\widehat{\mathcal{V}} : \mathbf{Var} \to \mathbf{Var}^{\text{SA}}$ is such that $\widehat{\mathcal{V}}(x) = x_{\mathcal{V}(x)}$. $\widehat{\mathcal{V}}$ is lifted to **Exp** and **Assert** in the obvious way, renaming the variables according to $\mathcal{V}$. Let $s \in \Sigma$ and $\mathcal{V} : \mathbf{Var} \to \mathbb{N}^+$. We define $\mathcal{V}(s) \in \mathbf{Var}^{\text{SA}} \rightharpoonup D$ as the partial function $[\widehat{\mathcal{V}}(x) \mapsto s(x) \mid x \in \mathbf{Var}]$.

The translation function T_{sa} is presented in Figure 5 (top). The function T_{sa} receives the initial version of the variable identifier and the annotated program, and returns a pair with the final version of each variable identifier and the SA translated program. The definition of T_{sa} relies on various auxiliary functions that deal with the version list and version functions, and also generate renaming commands. The functions are defined using Haskell-like syntax. We give a brief description of each one:

- next increments the first element of a variable version.
- new appends a new element at the head of a variable version.
- jump is used to merge the first two elements of a variable version. It is used when exiting a loop, in order to return to the previous context.
- sup receives two functions $\mathcal{V}$ and $\mathcal{V}'$, and returns a new one that returns the highest version for each variable (depending on whether it is in $\mathcal{V}$ or in $\mathcal{V}'$). A variable version $x : xs$ is higher than $y : ys$ if $x > y$.
- merge receives two functions $\mathcal{V}$ and $\mathcal{V}'$ and returns a **Rnm** (see Definition 5) containing assignments of the form $\widehat{\mathcal{V}'}(x) := \widehat{\mathcal{V}}(x)$, for each $x \in \mathsf{dom}(\mathcal{V})$ such that $\mathcal{V}(x) \prec \mathcal{V}'(x)$.
- upd fetches the appropriate version of a variable to be used after a loop.

For the sake of simplicity, we assume that the renaming sequences $\mathcal{I}$ and $\mathcal{U}$, defined in the case of while commands, follow some predefined order established over **Var** (any order will do).

4 Example

We will now illustrate the use of the SA translation by showing the result of applying it to the following Hoare triple containing the factorial program (let us call it FACT). To illustrate how nested loops are handled, only sum arithmetic instructions are used in the program, and multiplication is implemented by a loop.

$\{n \geq 0 \wedge aux = n\}$
$f := 1\,;$
$i := 1\,;$
while $i \leq n$ **do** $\{f = (i-1)! \wedge i \leq n+1\}$
{
  $j := 1\,;$
  $r := 0\,;$
  while $j \leq i$ **do** $\{j \leq i+1 \wedge r = f * (j-1)\}$
  {
    $r := r + f\,;$
    $j := j + 1$
  }$\,;$
  $f := r\,;$
  $i := i + 1$
}
$\{f = aux!\}$

Below we show the result of applying the program-level T_{sa} function to the above program, taking as initial version function $\mathcal{V}$ that maps every variable to the list containing the sole element 0 (note that any version function could be used). For the sake of presentation, index lists will be depicted using '.' as a separator and omitting the empty list constructor. The translated function, which we will call $\mathsf{FACT}^{\mathsf{sa}}$ is as follows:

$$
\begin{array}{l}
f_1 := 1\,; \\
i_1 := 1\,; \\
\textbf{for}\ (\ \{j_{1.0} := j_0\,;\ r_{1.0} := r_0\,;\ f_{1.1} := f_1\,;\ i_{1.1} := i_1\},\\
\qquad i_{1.1} \leq n_0, \\
\qquad \{j_{1.0} := j_{3.0}\,;\ r_{1.0} := r_{3.0}\,;\ f_{1.1} := f_{2.1}\,;\ i_{1.1} := i_{2.1}\} \\
\quad)\ \textbf{do}\,\{f_{1.1} = (i_{1.1} - 1)! \wedge i_{1.1} \leq n_0 + 1\} \\
\{ \\
\quad j_{2.0} := 1\,; \\
\quad r_{2.0} := 0\,; \\
\quad \textbf{for}\ (\ \{r_{1.2.0} := r_{2.0}\,;\ j_{1.2.0} := j_{2.0}\}, \\
\qquad j_{1.2.0} \leq i_{1.1}, \\
\qquad \{r_{1.2.0} := r_{2.2.0}\,;\ j_{1.2.0} := j_{2.2.0}\} \\
\quad)\ \textbf{do}\,\{j_{1.2.0} \leq i_{1.1} + 1 \wedge r_{1.2.0} = f_{1.1} * (j_{1.2.0} - 1)\} \\
\\
\quad \{ \\
\qquad r_{2.2.0} := r_{1.2.0} + f_{1.1}\,; \\
\qquad j_{2.2.0} := j_{1.2.0} + 1 \\
\quad \}\,; \\
\quad r_{3.0} := r_{1.2.0}\,; \\
\quad j_{3.0} := j_{1.2.0}\,; \\
\quad f_{2.1} := r_{3.0}\,; \\
\quad i_{2.1} := i_{1.1} + 1 \\
\}\,; \\
j_1 := j_{1.0}\,; \\
r_1 := r_{1.0}\,; \\
f_2 := f_{1.1}\,; \\
i_2 := i_{1.1}
\end{array}
$$

In addition to the SA program, T_{sa} returns a version function $\mathcal{V}'$ which is $\mathcal{V}[f \mapsto 2, i \mapsto 2, j \mapsto 1, r \mapsto 1]$. The initial and final versions functions $\mathcal{V}$ and $\mathcal{V}'$ will be applied to the precondition and postcondition to obtain the following triple.

$$\{n_0 \geq 0 \wedge aux_0 = n_0\}\,\mathsf{FACT}^{\mathsf{sa}}\,\{f_2 = aux_0!\}$$

As expected, the program does not assign free variables from the preconditions, that is, $\{n_0 \geq 0 \wedge aux_0 = n_0\}\#\mathsf{FACT}^{\mathsf{sa}}$.

5 Proving T_{sa} is an SA translation

We will now show that T_{sa} is indeed an SA translation.

Firstly, we prove that the T_{sa} translation preserves the operational semantics of the original programs in the following sense: if the translated program executes in a state where the values of the input version of the variables coincide with the values of the original variables in the initial state, then in the final state

the values of output versions of the variables are also equal to the values of the original variables in the final state.

Secondly, we prove that lifting the translation function to Hoare triples results in a sound translation, i.e., if the translated triple is valid then the original triple must also be valid.

Finally, we will show that Hg-derivability is preserved, i.e. if a Hoare triple for an annotated program is derivable in Hg, then the translated triple is also derivable in Hg.

We first consider some lemmas.

Lemma 4. *Let* $\mathcal{V} \in \mathbf{Var} \to \mathbb{N}^+$, $s \in \Sigma$ *and* $s' \in \Sigma^{\mathrm{SA}}$. *If* $\forall x \in \mathbf{Var}.\, s(x) = s'(\widehat{\mathcal{V}}(x))$, *then* $s' = s'_0 \oplus \mathcal{V}(s)$ *for some* $s'_0 \in \Sigma^{\mathrm{SA}}$.

Proof. Follows directly from the definitions. □

Lemma 5. *Let* $e \in \mathbf{Exp}$, $\phi \in \mathbf{Assert}$, $\mathcal{V} \in \mathbf{Var} \to \mathbb{N}^+$, $s \in \Sigma$ *and* $s' \in \Sigma^{\mathrm{SA}}$.

1. $[\![\widehat{\mathcal{V}}(e)]\!](s' \oplus \mathcal{V}(s)) = [\![e]\!](s)$
2. $[\![\widehat{\mathcal{V}}(\phi)]\!](s' \oplus \mathcal{V}(s)) = [\![\phi]\!](s)$

Proof. Both proofs follow directly from Lemma 4. □

Lemma 6. *Let* $C \in \mathbf{AComm}$ *and* $\mathcal{V} \in \mathbf{Var} \to \mathbb{N}^+$. *If* $T_{\mathsf{sa}}(\mathcal{V}, C) = (\mathcal{V}', C')$, *then for every* $x \notin \mathsf{Asgn}(C)$, $\mathcal{V}(x) = \mathcal{V}'(x)$.

Proof. By induction on the structure of C. □

The following lemma plays a central role in the proof of Proposition 4, for the case of a while command.

Lemma 7. *Let* $C_t \in \mathbf{AComm}$, $\mathcal{V} \in \mathbf{Var} \to \mathbb{N}^+$, $s_i, s_f \in \Sigma$, $s', s'_f \in \Sigma^{\mathrm{SA}}$ *and*

$$\begin{aligned}
&\mathcal{V}' = \mathcal{V}[x \mapsto \mathsf{new}(\mathcal{V}(x)) \mid x \in \mathsf{Asgn}(C_t)] \\
&T_{\mathsf{sa}}(\mathcal{V}', C_t) = (\mathcal{V}'', C'_t) \\
&\mathcal{U} = [x_{\mathsf{new}(\mathcal{V}(x))} := x_{\mathcal{V}''(x)} \mid x \in \mathsf{Asgn}(C_t)]
\end{aligned}$$

If $\langle \mathbf{while}\ b\ \mathbf{do}\ \lfloor C_t \rfloor, s_i \rangle \rightsquigarrow s_f$ *and* $\langle \mathbf{while}\ \mathcal{V}'(b)\ \mathbf{do}\ \{\lfloor \mathcal{W}(C'_t) \rfloor; \mathcal{U}\}; \mathsf{upd}(\mathsf{dom}(\mathcal{U})), s' \oplus \mathcal{V}(s_i) \rangle \rightsquigarrow s'_f$ *then* $\forall x \in \mathbf{Var}.\, s_f(x) = s'_f(\widehat{\mathcal{V}'}(x))$.

Proof. By induction on the derivation of the evaluation relation $\rightsquigarrow$. Assume

$$\begin{aligned}
&\langle \mathbf{while}\ b\ \mathbf{do}\ \lfloor C_t \rfloor, s_i \rangle \rightsquigarrow s_f \text{ and} \\
&\langle \mathbf{while}\ \mathcal{V}'(b)\ \mathbf{do}\ \{\lfloor \mathcal{W}(C'_t) \rfloor; \mathcal{U}\}; \mathsf{upd}(\mathsf{dom}(\mathcal{U})), s' \oplus \mathcal{V}(s_i) \rangle \rightsquigarrow s'_f
\end{aligned}$$

Two cases can occur:

- Case $[\![b]\!](s_i) = \mathbf{F}$, then $[\![\widehat{\mathcal{V}'}(b)]\!](s' \oplus \mathcal{V}'(s_i)) = [\![b]\!](s_i) = \mathbf{F}$ by Lemma 5. In this case we have $s_f = s_i$ and $s'_f = s' \oplus \mathcal{V}'(s_i)$. Hence, $\forall x \in \mathbf{Var}.\, s'_f(\widehat{\mathcal{V}'}(x)) = s_i(x) = s_f(x)$.

– Case $[\![b]\!](s_i) = \mathbf{T}$, then $[\![\widehat{\mathcal{V}'}(b)]\!](s' \oplus \mathcal{V}'(s_i)) = [\![b]\!](s_i) = \mathbf{T}$ by Lemma 5. In this case we must have, for some $s_1 \in \Sigma$,

$$\langle \lfloor C_t \rfloor, s_i \rangle \rightsquigarrow s_1 \tag{1}$$
$$\langle \textbf{while } b \textbf{ do } \lfloor C_t \rfloor, s_1 \rangle \rightsquigarrow s_f \tag{2}$$

and also, for some $s'_0, s'_1 \in \Sigma^{\text{SA}}$,

$$\langle \lfloor \mathcal{W}(C'_t) \rfloor, s' \oplus \mathcal{V}'(s_i) \rangle \rightsquigarrow s'_0 \tag{3}$$
$$s'_0 = s'_2 \oplus \mathcal{V}''(s_i) \tag{4}$$
$$\langle \mathcal{U}, s'_0 \rangle \rightsquigarrow s'_1 \tag{5}$$
$$s'_1 = s'_0[x_{\mathsf{new}(x_{\mathcal{V}(x)})} \mapsto [\![x_{\mathcal{V}''(x)}]\!](s'_0) \mid x \in \mathsf{Asgn}(C_t)] = s'_0 \oplus \mathcal{V}'(s_1) \tag{6}$$
$$\langle \textbf{while } \widehat{\mathcal{V}'}(b) \textbf{ do } \{\lfloor \mathcal{W}(C'_t) \rfloor; \mathcal{U}\}, s'_1 \rangle \rightsquigarrow s'_f \tag{7}$$

Note that (4) follows from (3) by Lemma 6, and that justifies (6). From (2), (7) and (6), by induction hypothesis, we get $\forall x \in \mathbf{Var}.\, s_f(x) = s'_f(\widehat{\mathcal{V}'}(x))$.

□

We will know prove that the T_{sa} translation preserves the operational semantics of the original programs. i.e., if the translated program executes in a state where the values of the input version of the variables coincide with the values of the original variables in the initial state, then in the final state the values of output versions of the variables are also equal to the values of the original variables in the final state.

Proposition 4. *Let $C \in \mathbf{AComm}$, $\mathcal{V} \in \mathbf{Var} \to \mathbb{N}^+$, $s_i, s_f \in \Sigma$, $s', s'_f \in \Sigma^{\text{SA}}$ and $T_{\mathsf{sa}}(\mathcal{V}, C) = (\mathcal{V}', C')$. If $\langle \lfloor C \rfloor, s_i \rangle \rightsquigarrow s_f$ and $\langle \lfloor \mathcal{W}(C') \rfloor, s' \oplus \mathcal{V}(s_i) \rangle \rightsquigarrow s'_f$, then $\forall x \in \mathbf{Var}.\, s_f(x) = s'_f(\widehat{\mathcal{V}'}(x))$.*

Proof. By induction on the structure of C.

– Case $C \equiv \textbf{skip}$. The hypothesis are:

$$T_{\mathsf{sa}}(\mathcal{V}, \textbf{skip}) = (\mathcal{V}, \textbf{skip})$$
$$\langle \lfloor \textbf{skip} \rfloor, s_i \rangle \rightsquigarrow s_i$$
$$\langle \lfloor \mathcal{W}(\textbf{skip}) \rfloor, s' \oplus \mathcal{V}(s_i) \rangle \rightsquigarrow s' \oplus \mathcal{V}(s_i)$$

As for every $x \in \mathbf{Var}$, we have $(s' \oplus \mathcal{V}(s_i))(\widehat{\mathcal{V}}(x)) = s_i(x)$, we are done.

– Case $C \equiv x := e$. The hypothesis are:

$$T_{\mathsf{sa}}(\mathcal{V}, x := e) = (\mathcal{V}[x \mapsto \mathsf{next}(\mathcal{V}(x))], x_{\mathsf{next}(\mathcal{V}(x))} := \widehat{\mathcal{V}}(e))$$
$$\langle \lfloor x := e \rfloor, s_i \rangle \rightsquigarrow s_f \quad \text{with} \quad s_f = s_i[x \mapsto [\![e]\!](s_i)]$$
$$\langle \lfloor \mathcal{W}(x_{\mathsf{next}(\mathcal{V}(x))} := \widehat{\mathcal{V}}(e))) \rfloor, s' \oplus \mathcal{V}(s_i) \rangle \rightsquigarrow s'_f \quad \text{with}$$
$$s'_f = (s' \oplus \mathcal{V}(s_i))[x_{\mathsf{next}(\mathcal{V}(x))} \mapsto [\![\widehat{\mathcal{V}}(e)]\!](s' \oplus \mathcal{V}(s_i))]$$

We want to prove that $\forall y \in \mathbf{Var}.\, s_f(y) = s'_f(\widehat{\mathcal{V}[x \mapsto \mathsf{next}(\mathcal{V}(x))]}(y))$. Two cases can occur:

- If $y = x$, we have $s_f(y) = s_f(x) = [\![e]\!](s_i)$ and

$$s'_f(\widehat{\mathcal{V}[x \mapsto \mathsf{next}(\mathcal{V}(x))]}(x)) = s'_f(x_{\mathsf{next}(\mathcal{V}(x))}) = [\![\widehat{\mathcal{V}}(e)]\!](s' \oplus \mathcal{V}(s_i)) = [\![e]\!](s_i)$$

- If $y \neq x$, we have $s_f(y) = s_i(y)$ and

$$s'_f(\widehat{\mathcal{V}[x \mapsto \mathsf{next}(\mathcal{V}(x))]}(y)) = s'_f(\widehat{\mathcal{V}}(y)) = (s' \oplus \mathcal{V}(s_i))(\widehat{\mathcal{V}}(y)) = s_i(y)$$

– Case $C \equiv C_1; C_2$. The hypothesis are:

$$\begin{aligned}
&T_{\mathsf{sa}}(\mathcal{V}, C_1; C_2) = (\mathcal{V}'', C'_1; C'_2) \\
&\qquad\qquad \text{with } T_{\mathsf{sa}}(\mathcal{V}, C_1) = (\mathcal{V}', C'_1) \text{ and } T_{\mathsf{sa}}(\mathcal{V}', C_2) = (\mathcal{V}'', C'_2) \\
&\langle \lfloor C_1; C_2 \rfloor, s_i \rangle \rightsquigarrow s_f \\
&\langle \lfloor \mathcal{W}(C'_1; C'_2) \rfloor, s \oplus \mathcal{V}(s_i) \rangle \rightsquigarrow s'_f
\end{aligned}$$

We must have, for some $s_0 \in \Sigma$ and $s'_0 \in \Sigma^{\mathrm{SA}}$

$$\langle \lfloor C_1 \rfloor, s_i \rangle \rightsquigarrow s_0 \tag{8}$$
$$\langle \lfloor C_2 \rfloor, s_0 \rangle \rightsquigarrow s_f \tag{9}$$
$$\langle \lfloor \mathcal{W}(C'_1) \rfloor, s \oplus \mathcal{V}(s_i) \rangle \rightsquigarrow s'_0 \tag{10}$$
$$\langle \lfloor \mathcal{W}(C'_2) \rfloor, s'_0 \rangle \rightsquigarrow s'_f \tag{11}$$

From (8) and (10), by induction hypothesis, we have $\forall x \in \mathbf{Var}.\, s_0(x) = s'_0(\widehat{\mathcal{V}'}(x))$. Therefore, by Lemma 4, we have $s'_0 = s'_1 \oplus \mathcal{V}'(s_0)$ for some $s'_i \in \Sigma^{\mathrm{SA}}$. Consequently, from (9) and (11), by induction hypothesis, we conclude that $\forall x \in \mathbf{Var}.\, s_f(x) = s'_f(\widehat{\mathcal{V}''}(x))$.

– Case $C \equiv \mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f$. The hypothesis are:

$$\begin{aligned}
T_{\mathsf{sa}}(\mathcal{V}, C) = (\mathsf{sup}(\mathcal{V}', \mathcal{V}''), &\mathbf{if}\ \widehat{\mathcal{V}}(b)\ \mathbf{then}\ \{C'_t; \mathsf{merge}(\mathcal{V}', \mathcal{V}'')\} \\
&\mathbf{else}\ \{C'_f; \mathsf{merge}(\mathcal{V}'', \mathcal{V}')\}\ ) \\
\text{with } T_{\mathsf{sa}}(\mathcal{V}, C_t) = (\mathcal{V}', C'_t) &\text{ and } T_{\mathsf{sa}}(\mathcal{V}, C_f) = (\mathcal{V}'', C'_f)
\end{aligned}$$

$$\begin{aligned}
&\langle \lfloor \mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f \rfloor, s_i \rangle \rightsquigarrow s_f \\
&\langle \mathbf{if}\ \widehat{\mathcal{V}}(b)\ \ \mathbf{then}\ \{C'_t; \mathsf{merge}(\mathcal{V}', \mathcal{V}'')\} \\
&\qquad\quad \mathbf{else}\ \{C'_f; \mathsf{merge}(\mathcal{V}'', \mathcal{V}')\}, s' \oplus \mathcal{V}(s_i) \rangle \rightsquigarrow s'_f
\end{aligned}$$

- Case $[\![b]\!](s_i) = \mathbf{T}$, then $[\![\widehat{\mathcal{V}}(b)]\!](s' \oplus \mathcal{V}(s_i)) = \mathbf{T}$ by Lemma 5. Therefore one must have, for some $s'_0 \in \Sigma^{\mathrm{SA}}$,

$$\langle \lfloor C_t \rfloor, s_i \rangle \rightsquigarrow s_f \tag{12}$$
$$\langle \lfloor \mathcal{W}(C'_t) \rfloor, s' \oplus \mathcal{V}(s_i) \rangle \rightsquigarrow s'_0 \tag{13}$$
$$\langle \mathsf{merge}(\mathcal{V}', \mathcal{V}''), s'_0 \rangle \rightsquigarrow s'_f \tag{14}$$

From (12) and (13), by induction hypothesis we have that

$$\forall x \in \mathbf{Var}.\, s_f(x) = s'_0(\widehat{\mathcal{V}'}(x)) \tag{15}$$

$\mathsf{merge}\ (\mathcal{V}', \mathcal{V}'') = [x_{\mathcal{V}''(x)} := x_{\mathcal{V}'(x)} \mid x \in \mathbf{Var} \wedge \mathcal{V}'(x) \prec \mathcal{V}''(x)]$ so, $s'_f = s'_0[x_{\mathcal{V}''(x)} \mapsto [\![x_{\mathcal{V}'(x)}]\!](s'_0) \mid x \in \mathbf{Var} \wedge \mathcal{V}'(x) \prec \mathcal{V}''(x)]$. Moreover,

$$\mathsf{sup}\ (\mathcal{V}', \mathcal{V}'')(x) = \begin{cases} \mathcal{V}'(x) \text{ if } \mathcal{V}''(x) \prec \mathcal{V}'(x) \\ \mathcal{V}''(x) \text{ otherwise} \end{cases}$$

We will now prove that $\forall x \in \mathbf{Var}.\, s_f(x) = s'_f(\widehat{\mathsf{sup}(\mathcal{V}', \mathcal{V}'')}(x))$. Let $x \in \mathbf{Var}$.

* If $\mathcal{V}'(x) \prec \mathcal{V}''(x)$, then $s'_f(\widehat{\mathsf{sup}(\mathcal{V}', \mathcal{V}'')}(x)) = s'_f(\widehat{\mathcal{V}''}(x)) = [\![x_{\mathcal{V}'(x)}]\!](s'_0) = s'_0(\widehat{\mathcal{V}'}(x)) = s_f(x)$ by (15).
* If $\mathcal{V}'(x) \not\prec \mathcal{V}''(x)$, then $s'_f(\widehat{\mathsf{sup}(\mathcal{V}', \mathcal{V}'')}(x)) = s'_f(\widehat{\mathcal{V}'}(x)) = s'_0(\widehat{\mathcal{V}'}(x)) = s_f(x)$ by (15).

• Case $[\![b]\!](s_i) = \mathbf{F}$. Analogous to the previous case.

– Case $C \equiv \mathbf{while}\ b\ \mathbf{do}\ \{\theta\}\ C_t$. The hypothesis are:

$$T_{\mathsf{sa}}(\mathcal{V}, \mathbf{while}\ b\ \mathbf{do}\ \{\theta\}\ C_y) = (\mathcal{V}''', \mathbf{for}\ (\mathcal{I}, \widehat{\mathcal{V}'}(b), \mathcal{U})\ \mathbf{do}\ \{\widehat{\mathcal{V}'}(\theta)\}\ C'_t) \tag{16}$$

$$\text{with } \mathcal{I} = [x_{\mathsf{new}(\mathcal{V}(x))} | x \in \mathsf{Asgn}(C_t)] \tag{17}$$

$$\mathcal{V}' = \mathcal{V}[x \mapsto \mathsf{new}(\mathcal{V}(x)) \mid x \in \mathsf{Asgn}(C_t)] \tag{18}$$

$$(\mathcal{V}'', C'_t) = T_{\mathsf{sa}}(\mathcal{V}', C_t) \tag{19}$$

$$\mathcal{U} = [x_{\mathsf{new}(\mathcal{V}(x))} := x_{\mathcal{V}''(x)} \mid x \in \mathsf{Asgn}(C_t)] \tag{20}$$

$$\mathcal{V}''' = \mathcal{V}''[x \mapsto \mathsf{jump}(l) \mid x_l \in \mathsf{dom}(\mathcal{U})] \tag{21}$$

$$\langle \mathbf{while}\ b\ \mathbf{do}\ \lfloor C_t \rfloor, s_i \rangle \rightsquigarrow s_f \tag{22}$$

$$\langle \mathcal{I}; \mathbf{while}\ \mathcal{V}'(b)\ \mathbf{do}\ \{\lfloor \mathcal{W}(C'_t) \rfloor; \mathcal{U}\}; \mathsf{upd}(\mathsf{dom}(\mathcal{U})), s' \oplus \mathcal{V}(s_i) \rangle \rightsquigarrow s'_f \tag{23}$$

From (17), (18) and (27), we must have for some $s'_0, s'_1 \in \Sigma^{\mathrm{SA}}$ that:

$$\begin{array}{l} \langle \mathcal{I}, s' \oplus s_i \rangle \rightsquigarrow s'_1 \\ \langle \mathbf{while}\ \mathcal{V}'(b)\ \mathbf{do}\ \{\lfloor \mathcal{W}(C'_t) \rfloor; \mathcal{U}\}; \mathsf{upd}(\mathsf{dom}(\mathcal{U})), s'_1 \rangle \rightsquigarrow s'_f \\ s'_1 = s'_0 \oplus \mathcal{V}'(s_i) \end{array} \tag{24}$$

There are the following cases to consider:

• Case $[\![b]\!](s_i) = \mathbf{F}$, then $s_f = s_i$ and $[\![\widehat{\mathcal{V}'}(b)]\!](s' \oplus \mathcal{V}'(s_i)) = \mathbf{F}$, by Lemma 5. Therefore one must have

$$\begin{array}{l} \langle \mathbf{while}\ b\ \mathbf{do}\ \lfloor C_t \rfloor, s_i \rangle \rightsquigarrow s_i \\ \langle \mathbf{while}\ \mathcal{V}'(b)\ \mathbf{do}\ \{\lfloor \mathcal{W}(C'_t) \rfloor; \mathcal{U}\}, s'_1 \rangle \rightsquigarrow s'_1 \\ \langle \mathsf{upd}(\mathsf{dom}(\mathcal{U})), s'_1 \rangle \rightsquigarrow s'_f \end{array}$$

Moreover, we know that $\mathsf{upd}(\mathsf{dom}(\mathcal{U})) = [x_{\mathsf{jump}(l)} := x_l \mid x_l \in \mathsf{dom}(\mathcal{U})]$ so

$$\langle \mathsf{upd}(\mathsf{dom}(\mathcal{U})), s'_0 \oplus \mathcal{V}'(s_i) \rangle \rightsquigarrow s'_f \tag{25}$$

$$s'_f = (s'_0 \oplus \mathcal{V}'(s_i))[x_{\mathsf{jump}(l)} \mapsto [\![x_l]\!](s'_0 \oplus \mathcal{V}'(s_i)) \mid x_l \in \mathsf{dom}(\mathcal{U})] \tag{26}$$

We now prove that $\forall x \in \mathbf{Var}.\, s_f(x) = s'_f(\widehat{\mathcal{V}'}(x))$.

* If $y \notin \mathsf{Asgn}(C_t)$, then $s'_f(\widehat{\mathcal{V}'''}(y)) = s'_f(\widehat{\mathcal{V}''}(y)) = s'_f(\widehat{\mathcal{V}'}(y)) = s_i(y)$, using Lemma 6.
* If $x \in \mathsf{Asgn}(C_t)$, then $s'_f(\widehat{\mathcal{V}'''}(x)) = s'_f(x_{\mathsf{jump}(l)})$ for some $x_l \in \mathsf{dom}(\mathcal{U})$ and $s'_f(x_{\mathsf{jump}(l)}) = [\![x_l]\!](s'_0 \oplus \mathcal{V}'(s_i)) = [\![\widehat{\mathcal{V}'}(x)]\!](s'_0 \oplus \mathcal{V}'(s_i)) = [\![x]\!](s_i) = s_i(x)$, using Lemma 4.

- Case $[\![b]\!](s_i) = \mathbf{T}$, then $[\![\widehat{\mathcal{V}'}(b)]\!](s'_0 \oplus \mathcal{V}'(s_i)) = \mathbf{T}$, by Lemma 5. From (25), we must have, for some $s_2 \in \Sigma$,

$$\langle \lfloor C_t \rfloor, s_i \rangle \rightsquigarrow s_2 \tag{27}$$
$$\langle \mathbf{while}\ b\ \mathbf{do}\ \lfloor C_t \rfloor, s_2 \rangle \rightsquigarrow s_f \tag{28}$$

and also, from (24), for some $s'_2, s'_3, s'_4 \in \Sigma^{\mathrm{SA}}$,

$$\langle \lfloor \mathcal{W}(C'_t) \rfloor, s'_0 \oplus \mathcal{V}'(s_i) \rangle \rightsquigarrow s'_4 \tag{29}$$
$$\langle \mathcal{U}, s'_4 \rangle \rightsquigarrow s'_2 \tag{30}$$
$$s'_2 = s'_4[x_{\mathsf{new}(x_{\mathcal{V}(x)})} \mapsto [\![x_{\mathcal{V}''(x)}]\!](s'_4) \mid x \in \mathsf{Asgn}(C_t)] \tag{31}$$
$$\langle \mathbf{while}\ \widehat{\mathcal{V}'}(b)\ \mathbf{do}\ \{\lfloor \mathcal{W}(C'_t) \rfloor; \mathcal{U}\}, s'_2 \rangle \rightsquigarrow s'_3 \tag{32}$$
$$\langle \mathsf{upd}(\mathsf{dom}(\mathcal{U})), s'_3 \rangle \rightsquigarrow s'_f \tag{33}$$

From (27), (29) and (19), by induction hypothesis, we have that $\forall x \in \mathbf{Var}.\, s_2(x) = s'_4(\widehat{\mathcal{V}''}(x))$. Using this fact, (31) and (18), we can conclude that $s'_2 = s'_4 \oplus \mathcal{V}'(s_2)$. Because of this, (28) and (32), we get by Lemma 7

$$\forall x \in \mathbf{Var}.\, s_f(x) = s'_3(\widehat{\mathcal{V}'}(x)) \tag{34}$$

$\mathsf{upd}(\mathsf{dom}(\mathcal{U})) = [x_{\mathsf{jump}(l)} := x_l \mid x_l \in \mathsf{dom}(\mathcal{U})]$ and $\langle \mathsf{upd}(\mathsf{dom}(\mathcal{U})), s'_3 \rangle \rightsquigarrow s'_f$, so $s'_f = s'_3[x_{\mathsf{jump}(l)} \mapsto [\![x_l]\!](s'_3) \mid x_l \in \mathsf{dom}(\mathcal{U})]$. We will now prove that $\forall x \in \mathbf{Var}.\, s_f(x) = s'_f(\widehat{\mathcal{V}'''}(x))$.

* If $y \notin \mathsf{Asgn}(C_t)$, then $s'_f(\widehat{\mathcal{V}'''}(y)) = s'_f(\widehat{\mathcal{V}''}(y)) = s'_3(\widehat{\mathcal{V}'}(y)) = s_f(y)$, using Lemma 6 and (34).
* If $x \in \mathsf{Asgn}(C_t)$, then $s'_f(\widehat{\mathcal{V}'''}(x)) = s'_f(x_{\mathsf{jump}(l)})$ for some $x_l = x_{\mathsf{new}(\mathcal{V}(c)} \in \mathsf{dom}(\mathcal{U})$, and $s'_f(x_{\mathsf{jump}(l)}) = [\![x_l]\!](s'_3) = s_f(x)$, using (34).

□

We now prove that the translation function is sound, i.e., if the translated triple is valid then the original triple must also be valid. We need the following lemma.

Lemma 8. *Let $C \in \mathbf{AComm}$, $\mathcal{V} \in \mathbf{Var} \to \mathbb{N}^+$, and $s_i, s_f \in \Sigma$. If $\langle \lfloor C \rfloor, s_i \rangle \rightsquigarrow s_f$ and $T_{\mathsf{sa}}(\mathcal{V}, C) = (\mathcal{V}', C')$, then $\langle \lfloor \mathcal{W}(C') \rfloor, s'_1 \oplus \mathcal{V}(s_i) \rangle \rightsquigarrow s'_2 \oplus \mathcal{V}'(s_f)$, for some $s'_1, s'_2 \in \Sigma^{\mathrm{SA}}$.*

Proof. By induction on the structure of C using Proposition 4. □

Proposition 5. *Let* $C \in \mathbf{AComm}$, $\phi, \psi \in \mathbf{Assert}$, $\mathcal{V} \in \mathbf{Var} \to \mathbb{N}^+$ *and* $T_{\mathsf{sa}}(\mathcal{V}, C) = (\mathcal{V}', C')$. *If* $\models \{\widehat{\mathcal{V}}(\phi)\}\, \lfloor \mathcal{W}(C') \rfloor\, \{\widehat{\mathcal{V}'}(\psi)\}$, *then* $\models \{\phi\}\, \lfloor C \rfloor\, \{\psi\}$.

Proof. Let $T_{\mathsf{sa}}(\mathcal{V}, C) = (\mathcal{V}', C')$ and

$$\models \{\widehat{\mathcal{V}}(\phi)\}\, \lfloor \mathcal{W}(C') \rfloor\, \{\widehat{\mathcal{V}'}(\psi)\} \tag{35}$$

We want to prove that $\models \{\phi\}\, \lfloor C \rfloor\, \{\psi\}$, so assume, for some $s_i, s_f \in \Sigma$, that

$$[\![\phi]\!](s_i) = \mathbf{T} \tag{36}$$

$$\langle \lfloor C \rfloor, s_i \rangle \rightsquigarrow s_f \tag{37}$$

From (37), by Lemma 8, we have for some $s_1', s_2' \in \Sigma^{\mathrm{SA}}$

$$\langle \lfloor \mathcal{W}(C') \rfloor, s_1' \oplus \mathcal{V}(s_i) \rangle \rightsquigarrow s_2' \oplus \mathcal{V}'(s_f) \tag{38}$$

From (36), by Lemma 5, we have $[\![\widehat{\mathcal{V}}(\phi)]\!](s_1' \oplus \mathcal{V}(s_i)) = \mathbf{T}$. Thus, since we have (35) and (38), we get $[\![\widehat{\mathcal{V}'}(\psi)]\!](s_2' \oplus \mathcal{V}'(s_f)) = \mathbf{T}$ and, by Lemma 5, it follows that $[\![\psi]\!](s_f) = \mathbf{T}$. Hence $\models \{\phi\}\, \lfloor C \rfloor\, \{\psi\}$ holds. □

Finally we will show that the translation T_{sa} preserves Hg-derivations, i.e., if a Hoare triple for an annotated program is derivable in Hg, then the translated triple is also derivable in Hg. Again we start by proving some auxiliary lemmas.

Lemma 9. *Let* $\mathcal{V}, \mathcal{V}' \in \mathbf{Var} \to \mathbb{N}^+$ *and* $\psi \in \mathbf{Assert}$. *The following derivations hold*

1. $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\psi)\}\, \mathsf{merge}(\mathcal{V}, \mathcal{V}')\, \{\widehat{\mathsf{sup}(\mathcal{V}, \mathcal{V}')}(\psi)\}$
2. $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\psi)\}\, \mathsf{merge}(\mathcal{V}', \mathcal{V})\, \{\widehat{\mathsf{sup}(\mathcal{V}, \mathcal{V}')}(\psi)\}$

Proof. 1. We have $\mathsf{merge}\ (\mathcal{V}, \mathcal{V}') = [x_{\mathcal{V}'(x)} := x_{\mathcal{V}(x)} \mid x \in \mathbf{Var} \wedge \mathcal{V}(x) \prec \mathcal{V}'(x)]$ and

$$\mathsf{sup}\ (\mathcal{V}, \mathcal{V}')(x) = \begin{cases} \mathcal{V}(x) & \text{if } \mathcal{V}'(x) \prec \mathcal{V}(x) \\ \mathcal{V}'(x) & \text{otherwise} \end{cases}$$

$\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\psi)\}\, \mathsf{merge}(\mathcal{V}, \mathcal{V}')\, \{\widehat{\mathsf{sup}(\mathcal{V}, \mathcal{V}')}(\psi)\}$ follows from successively applying the (assign) and (seq) rules, using as precondition the postcondition with the substitution $[x_{\mathcal{V}(x)}/x_{\mathcal{V}'(x)}]$ for each assigment $x_{\mathcal{V}'(x)} := x_{\mathcal{V}(x)}$ of the renaming sequence, since $\widehat{\mathsf{sup}(\mathcal{V}, \mathcal{V}')}(\psi)[x_{\mathcal{V}(x)}/x_{\mathcal{V}'(x)} \mid x \in \mathbf{Var} \wedge \mathcal{V}(x) \prec \mathcal{V}'(x)] = \widehat{\mathcal{V}}(\psi)$.

2. Similar. □

Lemma 10. *Let* $\mathcal{I} \in \mathbf{Rnm}$ *and* $\phi \in \mathbf{Assert}$.

$$\vdash_{\mathsf{Hg}} \{\phi\}\, \mathcal{I}\, \{\mathcal{I}^{-1}(\phi)\}$$

Proof. Follows directly from Lemma 3. □

Lemma 11. *Let* $\mathcal{V} \in \mathbf{Var} \to \mathbb{N}^+$, $C \in \mathbf{AComm}$, $\mathcal{V}' = \mathcal{V}[x \mapsto \mathsf{new}(\mathcal{V}(x)) \mid x \in \mathsf{Asgn}(C)]$, $T_{\mathsf{sa}}(\mathcal{V}', C) = (\mathcal{V}'', C')$ *and* $\mathcal{U} = [x_{\mathsf{new}(\mathcal{V}(x))} := x_{\mathcal{V}''(x)} \mid x \in \mathsf{Asgn}(C)]$.

$$\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}''}(\theta)\}\, \mathcal{U}\, \{\widehat{\mathcal{V}'}(\theta)\}$$

Proof. $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}''}(\theta)\}\, \mathcal{U}\, \{\widehat{\mathcal{V}'}(\theta)\}$ follows from successively applying the (assign) and (seq) rules, using as precondition the postcondition with the substitution $[x_{\mathcal{V}''(x)}/x_{\mathsf{new}(\mathcal{V}(x))}]$ for each assigment $x_{\mathsf{new}(\mathcal{V}(x))} := x_{\mathcal{V}''(x)}$ of the renaming sequence, since

$$\begin{aligned}
&(\widehat{\mathcal{V}'}(\theta))[x_{\mathcal{V}''(x)}/x_{\mathsf{new}(\mathcal{V}(x))} \mid x \in \mathsf{Asgn}(C)] \\
&= (\mathcal{V}[x \mapsto \widehat{\mathsf{new}(\mathcal{V}(x)) \mid x \in \mathsf{Asgn}(C)}](\theta))[x_{\mathcal{V}''(x)}/x_{\mathsf{new}(\mathcal{V}(x))} \mid x \in \mathsf{Asgn}(C)] \\
&= \widehat{\mathcal{V}''}(\theta)
\end{aligned}$$

□

Lemma 12. *Let* $\mathcal{V} \in \mathbf{Var} \to \mathbb{N}^+$, $C \in \mathbf{AComm}$, $\mathcal{V}' = \mathcal{V}[x \mapsto \mathsf{new}(\mathcal{V}(x)) \mid x \in \mathsf{Asgn}(C)]$, $T_{\mathsf{sa}}(\mathcal{V}', C) = (\mathcal{V}'', C')$, $\mathcal{U} = [x_{\mathsf{new}(\mathcal{V}(x))} := x_{\mathcal{V}''(x)} \mid x \in \mathsf{Asgn}(C)]$ *and* $\mathcal{V}''' = \mathcal{V}''[x \mapsto \mathsf{jump}(l) \mid x_l \in \mathsf{dom}(\mathcal{U})]$.

$$\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}'}(\psi)\}\, \mathsf{upd}(\mathsf{dom}(\mathcal{U}))\, \{\widehat{\mathcal{V}'''}(\psi)\}$$

Proof. Remember that $\mathsf{upd}\ (\mathsf{dom}(\mathcal{U})) = [x_{\mathsf{jump}(l)} := x_l \mid x_l \in \mathsf{dom}(\mathcal{U})]$.

$\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}'}(\psi)\}\, \mathsf{upd}(\mathsf{dom}(\mathcal{U}))\, \{\widehat{\mathcal{V}'''}(\psi)\}$ follows from successively applying the (assign) and (seq) rules, using as precondition the postcondition with the substitution $[x_l/x_{\mathsf{jump}(l)}]$ for each assigment $x_{\mathsf{jump}(l)} := x_l$ of the renaming sequence, since

$$\begin{aligned}
&(\widehat{\mathcal{V}'''}(\psi)[x_l/x_{\mathsf{jump}(l)} \mid x_l \in \mathsf{dom}(\mathcal{U})]) \\
&= (\mathcal{V}''[x \mapsto \widehat{\mathsf{jump}(l) \mid x_l \in \mathsf{dom}(\mathcal{U})}](\psi)[x_l/x_{\mathsf{jump}(l)} \mid x_l \in \mathsf{dom}(\mathcal{U})]) \\
&= \mathcal{V}'(\psi)
\end{aligned}$$

□

Proposition 6. *Let* $C \in \mathbf{AComm}$, $\phi, \psi \in \mathbf{Assert}$, $\mathcal{V} \in \mathbf{Var} \to \mathbb{N}^+$ *and* $T_{\mathsf{sa}}(\mathcal{V}, C) = (\mathcal{V}', C')$. *If* $\vdash_{\mathsf{Hg}} \{\phi\}\, C\, \{\psi\}$, *then* $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\phi)\}\, \mathcal{W}(C')\, \{\widehat{\mathcal{V}'}(\psi)\}$.

Proof. By induction on the structure of $\vdash_{\mathsf{Hg}} \{\phi\}\, C\, \{\psi\}$.

– Assume the last step is:

$$\frac{}{\{\phi\}\, \mathbf{skip}\, \{\psi\}} \quad \text{with } \phi \to \psi$$

We have $T_{\mathsf{sa}}(\mathcal{V}, \mathbf{skip}) = (\mathcal{V}, \mathbf{skip})$. As $\models \phi \to \psi$ we have $\models \widehat{\mathcal{V}}(\phi) \to \widehat{\mathcal{V}}(\psi)$. So $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\phi)\}\, \mathbf{skip}\, \{\widehat{\mathcal{V}}(\psi)\}$ by applying the (skip) rule.

– Assume the last step is:

$$\frac{}{\{\phi\}\, x := e\, \{\psi\}} \quad \text{with } \phi \to \psi[e/x]$$

We have $T_{\mathsf{sa}}(\mathcal{V}, x := e) = (\mathcal{V}[x \mapsto \mathsf{next}(\mathcal{V}(x))], x_{\mathsf{next}(\mathcal{V}(x))} := \widehat{\mathcal{V}}(e))$. Since $\models \phi \to \psi[e/x]$, it follows that $\models \widehat{\mathcal{V}}(\phi \to \psi[e/x])$. Moreover,

$$\begin{aligned}
&\widehat{\mathcal{V}}(\phi) \to \widehat{\mathcal{V}[x \mapsto \mathsf{next}(\mathcal{V}(x))]}(\psi)[\widehat{\mathcal{V}}(e)/x_{\mathsf{next}(\mathcal{V}(x))}] \\
&= \widehat{\mathcal{V}}(\phi) \to \widehat{\mathcal{V}}(\psi[x_{\mathsf{next}(\mathcal{V}(x))}/x])[\widehat{\mathcal{V}}(e)/x_{\mathsf{next}(\mathcal{V}(x))}] \\
&= \widehat{\mathcal{V}}(\phi) \to \widehat{\mathcal{V}}(\psi[x_{\mathsf{next}(\mathcal{V}(x))}/x][e/x_{\mathsf{next}(\mathcal{V}(x))}]) \quad \text{, because } x_{\mathsf{next}(\mathcal{V}(x))} \notin \mathsf{FV}(\psi) \\
&= \widehat{\mathcal{V}}(\phi) \to \widehat{\mathcal{V}}(\psi[e/x]) \\
&= \widehat{\mathcal{V}}(\phi \to \psi[e/x])
\end{aligned}$$

Hence $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\phi)\}\, x_{\mathsf{next}(\mathcal{V}(x))} := \widehat{\mathcal{V}}(e)\, \{\widehat{\mathcal{V}[x \mapsto \mathsf{next}(\mathcal{V}(x))](\psi)}\}$ follows by the (assign) rule.

– Assume the last step is:

$$\frac{\{\phi\}\, C_1\, \{\theta\} \quad \{\theta\}\, C_2\, \{\psi\}}{\{\phi\}\, C_1\, ;\, C_2\, \{\psi\}}$$

We have $T_{\mathsf{sa}}(\mathcal{V}, C_1\, ;\, C_2) = (\mathcal{V}'', C_1'; C_2')$
with $T_{\mathsf{sa}}(\mathcal{V}, C_1) = (\mathcal{V}', C_1')$ and $T_{\mathsf{sa}}(\mathcal{V}', C_2) = (\mathcal{V}'', C_2')$.

By induction hypothesis, we have $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\phi)\}\, \mathcal{W}(C_1')\, \{\widehat{\mathcal{V}'}(\theta)\}$ and also $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}'}(\theta)\}\, \mathcal{W}(C_2')\, \{\widehat{\mathcal{V}''}(\psi)\}$. Hence, applying the (seq) rule, we get

$$\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\phi)\}\, \mathcal{W}(C_1'\, ;\, C_2')\, \{\widehat{\mathcal{V}''}(\psi)\}$$

– Assume the last step is:

$$\frac{\{\phi \wedge b\}\, C_t\, \{\psi\} \quad \{\phi \wedge \neg b\}\, C_f\, \{\psi\}}{\{\phi\}\, \mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f\, \{\psi\}}$$

We have $T_{\mathsf{sa}}(\mathcal{V}, C) = (\mathsf{sup}(\mathcal{V}', \mathcal{V}''), \mathbf{if}\ \widehat{\mathcal{V}}(b)\ \mathbf{then}\ \{C_t'; \mathsf{merge}(\mathcal{V}', \mathcal{V}'')\}$
$\mathbf{else}\ \{C_f'; \mathsf{merge}(\mathcal{V}'', \mathcal{V}')\}\ )$
with $T_{\mathsf{sa}}(\mathcal{V}, C_t) = (\mathcal{V}', C_t')$ and $T_{\mathsf{sa}}(\mathcal{V}, C_f) = (\mathcal{V}'', C_f')$

- By induction hypothesis we have that $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\phi \wedge b)\}\, \mathcal{W}(C_t')\, \{\widehat{\mathcal{V}'}(\psi)\}$. By Lemma 9 we have $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}'}(\psi)\}\, \mathsf{merge}(\mathcal{V}', \mathcal{V}'')\, \{\widehat{\mathsf{sup}(\mathcal{V}', \mathcal{V}'')(\psi)}\}$. So, applying rule (seq), we get

$$\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\phi \wedge b)\}\, \mathcal{W}(C_t')\, ;\, \mathsf{merge}(\mathcal{V}', \mathcal{V}'')\, \{\widehat{\mathsf{sup}(\mathcal{V}', \mathcal{V}'')(\psi)}\} \qquad (39)$$

- By induction hypothesis we have that $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\phi \wedge \neg b)\}\, \mathcal{W}(C'_f)\, \{\widehat{\mathcal{V}'}(\psi)\}$.
  By Lemma 9 we have $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}'}(\psi)\}\, \mathsf{merge}(\mathcal{V}'', \mathcal{V}')\, \widehat{\{\mathsf{sup}(\mathcal{V}', \mathcal{V}'')(\psi)\}}$. So, applying rule (seq), we get

$$\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\phi \wedge \neg b)\}\, \mathcal{W}(C'_f)\, ;\, \mathsf{merge}(\mathcal{V}'', \mathcal{V}')\, \widehat{\{\mathsf{sup}(\mathcal{V}', \mathcal{V}'')(\psi)\}} \tag{40}$$

Finally, from (39) and (40), by rule (if) we get

$$\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\phi)\}\, \mathbf{if}\ \widehat{\mathcal{V}}(b)\ \mathbf{then}\ \{\mathcal{W}(C'_t); \mathsf{merge}(\mathcal{V}', \mathcal{V}'')\}\ \mathbf{else}\ \{\mathcal{W}(C'_f); \mathsf{merge}(\mathcal{V}'', \mathcal{V}')\}\, \widehat{\{\mathsf{sup}(\mathcal{V}', \mathcal{V}'')(\psi))\}}$$

– Assume the last step is:

$$\frac{\{\theta \wedge b\}\, C\, \{\theta\}}{\{\phi\}\, \mathbf{while}\ b\ \mathbf{do}\, \{\theta\}\, C\, \{\psi\}} \text{ with } \phi \to \theta \text{ and } \theta \wedge \neg b \to \psi$$

We have

$$\begin{aligned} T_{\mathsf{sa}}(\mathcal{V}, \mathbf{while}\ b\ \mathbf{do}\, \{\theta\}\, C) = (\mathcal{V}''', \mathbf{for}\ (\mathcal{I}, \widehat{\mathcal{V}'}(b), \mathcal{U})\ \mathbf{do}\, \{\widehat{\mathcal{V}'}(\theta)\}\, C'; \mathsf{upd}(\mathsf{dom}(\mathcal{U}))) \\ \text{with } \mathcal{I} = [x_{\mathsf{new}(\mathcal{V}(x))} := x_{\mathcal{V}(x)} \mid x \in \mathsf{Asgn}(C)] \\ \mathcal{V}' = \mathcal{V}[x \mapsto \mathsf{new}(\mathcal{V}(x)) \mid x \in \mathsf{Asgn}(C)] \\ (\mathcal{V}'', C') = T_{\mathsf{sa}}(\mathcal{V}', C) \\ \mathcal{U} = [x_{\mathsf{new}(\mathcal{V}(x))} := x_{\mathcal{V}''(x)} \mid x \in \mathsf{Asgn}(C)] \\ \mathcal{V}''' = \mathcal{V}''[x \mapsto \mathsf{jump}(l) \mid x_l \in \mathsf{dom}(\mathcal{U})] \end{aligned}$$

$$\begin{aligned} \text{and} \quad & \mathcal{W}(\mathbf{for}\ (\mathcal{I}, \widehat{\mathcal{V}'}(b), \mathcal{U})\ \mathbf{do}\, \{\widehat{\mathcal{V}'}(\theta)\}\, C'\, ;\, \mathsf{upd}(\mathsf{dom}(\mathcal{U}))) \\ & = \mathcal{I}\, ;\, \mathbf{while}\ \widehat{\mathcal{V}'}(b)\ \mathbf{do}\, \{\widehat{\mathcal{V}'}(\theta)\}\, \{\mathcal{W}(C')\, ;\, \mathcal{U}\}\, ;\, \mathsf{upd}(\mathsf{dom}(\mathcal{U})) \end{aligned}$$

We must prove that
$\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\phi)\}\, \mathcal{I}\, ;\, \mathbf{while}\ \widehat{\mathcal{V}'}(b)\ \mathbf{do}\, \{\widehat{\mathcal{V}'}(\theta)\}\, \{\mathcal{W}(C')\, ;\, \mathcal{U}\}\, ;\, \mathsf{upd}(\mathsf{dom}(\mathcal{U}))\, \widehat{\{\mathcal{V}'''}(\psi)\}$
This follows from applying rule (seq) twice, to the following premisses:

$$\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\phi)\}\, \mathcal{I}\, \{\mathcal{I}^{-1}(\widehat{\mathcal{V}}(\phi))\} \tag{41}$$

$$\vdash_{\mathsf{Hg}} \{\mathcal{I}^{-1}(\widehat{\mathcal{V}}(\phi))\}\, \mathbf{while}\ \widehat{\mathcal{V}'}(b)\ \mathbf{do}\, \{\widehat{\mathcal{V}'}(\theta)\}\, \{\mathcal{W}(C')\, ;\, \mathcal{U}\}\, \{\widehat{\mathcal{V}'}(\psi)\} \tag{42}$$

$$\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}'}(\psi)\}\, \mathsf{upd}(\mathsf{dom}(\mathcal{U}))\, \{\widehat{\mathcal{V}'''}(\psi)\} \tag{43}$$

We have that (41) follows from Lemma 10, and (43) follows from Lemma 12.
We will now prove (42). By induction hypotesis, we have $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}'}(\theta) \wedge \widehat{\mathcal{V}'}(b)\}\, \mathcal{W}(C')\, \{\widehat{\mathcal{V}''}(\theta)\}$. Moreover, by Lemma 11, $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}''}(\theta)\}\, \mathcal{U}\, \{\widehat{\mathcal{V}'}(\theta)\}$. Thus, by rule (seq), $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}'}(\theta) \wedge \widehat{\mathcal{V}'}(b)\}\, \mathcal{W}(C')\, ;\, \mathcal{U}\, \{\widehat{\mathcal{V}'}(\theta)\}$.
Since $(\mathcal{I}^{-1}(\widehat{\mathcal{V}}(\phi)) \to \widehat{\mathcal{V}'}(\theta))$ and $(\widehat{\mathcal{V}'}(\theta) \wedge \neg\widehat{\mathcal{V}'}(b) \to \widehat{\mathcal{V}'}(\psi))$ both hold, we can now apply rule (while), and obtain

$$\vdash_{\mathsf{Hg}} \{\mathcal{I}^{-1}(\widehat{\mathcal{V}}(\phi))\}\, \mathbf{while}\ \widehat{\mathcal{V}'}(b)\ \mathbf{do}\, \{\widehat{\mathcal{V}'}(\theta)\}\, \{\mathcal{W}(C')\, ;\, \mathcal{U}\}\, \{\widehat{\mathcal{V}'}(\psi)\}$$

□

It is now immediate that T_{sa} conforms th Definition 8.

Proposition 7. *The T_{sa} function of Figure 5 is an SA translation.*

Proof. Let $C \in \mathbf{AComm}$, $\phi, \psi \in \mathbf{Assert}$, $\mathcal{V} \in \mathbf{Var} \to \mathbb{N}^+$, $T_{\mathsf{sa}}(\mathcal{V}, C) = (\mathcal{V}', C')$ and $T_{\mathsf{sa}}(\phi, C, \psi) = (\widehat{\mathcal{V}}(\phi), C', \widehat{\mathcal{V}'}(\psi))$. We want to prove that

1. If $\models \{\widehat{\mathcal{V}}(\phi)\} \lfloor \mathcal{W}(C') \rfloor \{\widehat{\mathcal{V}'}(\psi)\}$, then $\models \{\phi\} \lfloor C \rfloor \{\psi\}$.
2. If $\vdash_{\mathsf{Hg}} \{\phi\} C \{\psi\}$, then $\vdash_{\mathsf{Hg}} \{\widehat{\mathcal{V}}(\phi)\} \mathcal{W}(C') \{\widehat{\mathcal{V}'}(\psi)\}$.

1. follows directly from Proposition 5.
2. follows directly from Proposition 6. □

6 Conclusion

We have proposed a translation of programs annotated with loop invariants into a dynamic single-assignment form. The translation extends to Hoare triples and is adequate for program verification using the efficient VCGen presented in [6]. Together with that VCGen and the corresponding program logic for single-assignment programs, the translation is part of a workflow for the deductive verification of imperative programs in a way that is efficient and allows for adaptation, since the logic is adaptation-complete. We remark that the workflow, also described in that paper, does not depend on this specific translation, but instead defines semantic requirements that a translation should comply to. The core result of the present report, proved in detail here, is precisely that the specific translation introduced here conforms to those requirements.

References

1. Krzysztof R. Apt. Ten years of Hoare's logic: A survey - part 1. *ACM Trans. Program. Lang. Syst.*, 3(4):431–483, 1981.
2. Stephen A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM J. Comput.*, 7(1):70–90, 1978.
3. Ron Cytron, Jeanne Ferrante, BK Rosen, Mark N Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.
4. Mike Gordon and Hélène Collavizza. Forward with Hoare. In *Reflections on the Work of C.A.R. Hoare*, History of Computing, pages 101–121. Springer, 2010.
5. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.
6. Cláudio Belo Lourenço, Maria João Frade, and Jorge Sousa Pinto. Formalizing Single-assignment Program Verification: an Adaptation-complete Approach. To appear in Proceedings of *ESOP 2015*.
7. John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, Cambridge, England, 1998.