Universidade do Minho
Escola de Engenharia
Departamento de Informática

Marta Vasconcelos Castro Azevedo

Correct Translation of Imperative Programs to Single Assignment Form

January 2017

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Marta Vasconcelos Castro Azevedo

Correct Translation of Imperative Programs to Single Assignment Form

Master dissertation
Master Degree in Computer Science

Dissertation supervised by
Professor Maria João Frade
Professor Luís Pinto

January 2017

ABSTRACT

A common practice in compiler design is to have an intermediate representation of the source code in Static Single-Assignment (SSA) form in order to simplify the code optimization process and make it more efficient. Generally, one says that an imperative program is in SSA form if each variable is assigned exactly once.

In this thesis we study the central ideas of SSA-programs in the context of a simple imperative language including jump instructions. The focus of this work is the proof of correctness of a translation from programs of the source imperative language into the SSA format. In particular, we formally introduce the syntax and the semantics of the source imperative language ($\mathcal{GL}$) and the SSA language; we define and implement a function that translates from source imperative programs into SSA-programs; we develop an alternative operational semantics, in order to be able to relate the execution of a source program and of its SSA translation; we prove soundness and completeness results for the translation, relatively to the alternative operational semantics, and from these results we prove correctness of the translation relatively to the initial small-step semantics.

RESUMO

Uma prática comum no design de compiladores é ter uma representação intermédia do código fonte em formato *Static Single Assigment* (SSA), de modo a facilitar o processo subsequente de análise e optimização de código. Em termos gerais, diz-se que um programa imperativo está no formato SSA se cada variável do programa é atribuída exatamente uma vez.

Nesta tese estudamos as ideias principais do SSA no contexto de uma linguagem imperativa simples com instruções de salto. O foco deste trabalho é a prova de correcção de uma tradução dos programas fonte para formato SSA. Concretamente, definimos formalmente a sintaxe e a semântica da linguagem fonte (GL) e da linguagem em formato SSA; definimos e implementamos a função de tradução; desenvolvemos semânticas operacionais alternativas de modo a permitir relacionar a execução do programa fonte com a sua tradução SSA; provamos a idoneidade e a completude do processo de tradução relativamente às semânticas alternativas definidas; e a partir destes resultados demostramos a correcção da tradução em ordem às semânticas operacionais estruturais definidas inicialmente.

CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

CPS	Continuation Passing Style
SA	Single Assignment
SSA	Static Single Assignment
CFG	Control Flow Graph

1 INTRODUCTION

1.1 IMPERATIVE AND FUNCTIONAL PROGRAMMING

A programming language is a notation for writing programs, which are specifications of a computation to be performed by a machine. Different approaches to programming have been developed over time. There are many different programming paradigms that allow the possibility to determine the programmer's view of the problem. Despite some languages are designed to support one particular paradigm, there are others that support multiple paradigms.

This work focuses on two of these paradigms, which have drawn programmers and computer scientists' attention since early times: the imperative programming paradigm and the functional programming paradigm. Both paradigms are built upon different ideas.

In imperative languages computation is specified in an imperative form (i.e., as a sequence of operations to perform). The model of computation in the imperative paradigm relies on the notion of *state* (seen as the content of the program variables which represent storage/memory locations, at any given point in the program's execution) and *statements/commands* that change a program's state. So, an imperative program consists of a sequence of commands for the computer to perform. A program describes *how* a program operates. In Figure 1 we can see a snippet of a imperative program. This program sums all the numbers from 0 to n.

The first imperative programming languages were machine languages with simple instructions. Gradually, some more complex languages were introduced such as *C*, *Pascal* or *Java*.

Contrasting with imperative programming, functional programming focuses on *what* the program should accomplish without specifying how the program should achieve the result. Functional programming has its roots in λ-*calculus* (Church, 1932). The functional programming paradigm is based on the notion of mathematical function (a program is a collection of functions) and program execution corresponds to the evaluation of expressions (involving the functions defined in the program). Functional programs deal only with mathematical variables (the formal arguments of the functions). The output value of a function depends only on the arguments that are inputs to the function. So, calling a function twice with the same argument values will produce the same result each time. This property is called *referential transparency* and make it much easier to understand and predict the behavior of a program.

```
        goto $l$;
$l$ :      $i := 1$;
        $s := 0$;
        goto $l'$
$l'$ :     branch $(i \leq n)$   $l''$   $l'''$
$l''$ :    $s := s + 1$;
        $i := i + 1$;
        goto $l'$
$l'''$ :   return $s$;
```

Figure 1.: Example of a imperative program

Note that in imperative programming, as for instance in *C*, one usually call "functions" to subroutines that are not really functions (in the mathematical sense), since they can have side effects that may change the value of the program state and the value they produce may depend on the state also. Because of this, they lack referential transparency, i.e. the same language expression may have different values at different times depending on the state of the program.

An expression is said to be *referentially transparent* if it can be replaced with its value without changing the behavior of a program. This allows the programmer and the compiler to reason about program behavior as a rewrite system, and is very helpful in the static analysis of the code, proving its correctness or optimizing it.

Functional programming languages have long been popular in academia. More recently, several functional programming languages are being used in industry and became popular for building a range of applications. The first functional-flavoured language was Lisp (developed in 1950s). Since then many different languages emerged (some of them multi-paradigm). Scheme, SML, Ocaml and Haskell are examples of more recent functional programming languages..

We can see, in Figure 2 a program fragment in Haskell with the functional approach to the sum function displayed on Figure 1.

```
fun :: Int → Int
fun n = if n ≤ 0 then 0 else n + (fun n − 1)
```

Figure 2.: Example of a functional program

1.2 COMPILATION PROCESS

For a computer to be able to execute the instructions given by a program written in a (high-level) programming language, it is necessary to translate the source program into an equivalent target program in machine language (for the specific hardware). This translation is the task of the *compiler*.

Original Program	SA Program
$x := 5;$	$x_1 := 5;$
$y := x + 1;$	$y_1 := x_1 + 1;$
$x := y * 10;$	$x_2 := y_1 * 10;$
$y := 3 + y;$	$y_2 := 3 + y_1;$
$z := x + y;$	$z_1 := x_2 + y_2;$

Figure 3.: Translation in a simple program in SA

Compilers are complex programs that are organized in several phases usually grouped as follows: The *front-end* performs the syntactic and semantic processing and generates an intermediate representation of the source code to be then processed by the middle-end. The *middle-end* performs optimizations over the intermediate code generates another intermediate representation for the back-end. The *back-end* performs more analysis and optimizations for a particular hardware and then generates machine code for a particular processor and operating system. This approach makes it possible to combine front ends for different languages with back ends for different hardware. Examples of this approach are the GNU Compiler Collection (Stallman, 2001) and LLVM (Lattner and Adve, 2004), which have multiple front-ends, shared analysis and multiple back-ends.

In compiler design, a common practice is to have the intermediate representation of the source code in static single assignment form in order to simplify the code optimization process.

Another style used as an intermediate representation to perform optimizations and program transformations is the continuation-passing style. Despite continuations can be used to compile most programming languages they are more common on the compilation process of functional languages.

1.3 SINGLE ASSIGNMENT FORM

One says that an imperative program is in *static single assignment* (SSA) form if each variable is assigned exactly once, and every variable is defined before it is used. This restriction makes explicit in the program syntax the link between the program point where a variable is defined and read.

Converting into SSA form is usually done splitting the (original) variables into "versions", with the new variables typically denoted by the original name with an index, so that every definition gets its own version. The transformation of a program into SSA form has to be semantic preserving. One simply tags each variable definition with an index, and each variable use with the index corresponding to the last definition of this variable. You can see an example in Figure 3.

This transformation preserves the semantics of the program in the sense that the final values of x, y and z in the first program coincide with the values of the final vertions of those variables: x_2, y_2 and z_1. However for more complex programs (with jumps) the transformation into SSA is more challenging.

The SSA form was introduced in the 1980s by Ron Cytron *et al* (Cytron et al., 1991) and it is a popular intermediate representation for compiler optimizations. The considerable strength of the SSA form, where variables are statically assigned exactly once, simplifies the definition of many optimizations, and improves their efficiency and the quality of their results. The SSA format plays a central role in a range of optimization algorithms relying on data flow information, and hence, to the correctness of compilers employing those algorithms. Examples of optimization algorithms which benefit from the use of SSA form include, among others, constant propagation, value range propagation, dead code elimination and register allocation. Many modern optimizing compilers, such as the GNU Compiler Collection and LLVM Compiler Infrastructure rely heavily on SSA form.

The apparent simplicity of the SSA conversion of the code snippet example given above is misleading. For program with control flow commands (for instance, `if` and `while` or `goto` commands) the translation to SSA form cannot be done solely by tagging variables: to handle it one must insert the so-called *ϕ-functions*, which control the merging of data flow edges entering code blocks. For instance, where two control-flow edges join together, carrying different values of some variable, one must somehow merge the two values. This is done with the help of a ϕ-function, which is a notational trick. In some node with two in-edges, the expression $\phi(a, b)$ has the value a if one reaches this node on the first in-edge, and b if one comes in on the second in-edge. The semantics of ϕ-functions is in the seminal paper by Cytron et al. (1991):

> ”If control reaches node j from its kth predecessor, then the run-time support remembers k while executing the ϕ-functions in j. The value of $\phi(x_1, x_2, ...)$ is just the value of the kth operand. Each execution of a ϕ-function uses only one of the operands, but which one depends on the flow of control just before entering j.”

Let us illustrate the translation to SSA form with an example in Figure 4.

There may be several SSA forms for a single control-flow graph program. As the number of ϕ-functions directly impacts size of the SSA form and the quality of the subsequent optimisations it is important that SSA generators for real compilers produce a SSA form with a minimal number of ϕ-functions. Implementations of minimal SSA generally rely on the notion of *dominance frontier* to choose where to insert ϕ-functions, as indicated, for instance, in (Cytron et al., 1991).

The Control Flow Graph (CFG) is a representation, using graph notation, of all paths that might be traversed a program during execution. In a CFG, node A *dominates* node B if any path from the start to node B must go though A. There is a vast body of work on efficient methods of computing minimal SSA form and SSA-based optimisations. General references are Appel (1998b) and Muchnick (1997a).

	Original imperative program		SSA correspondent imperative program
	goto l		goto l_1
l :	$i := 1$; $s := 0$; goto l'	l :	$i_1 := 1$; $s_1 := 0$; goto l'_1
l' :	branch $(i \leq n)\ l''\ l'''$	l' :	$i_3 := \phi(i_1, i_4)$; $s_3 := \phi(s_1, s_4)$; branch $(i_3 \leq n_1)\ l''_1\ l'''_1$
l''	$s := s + 1$; $i := i + 1$; goto l'	l'' :	$s_4 := s_3 + 1$; $i_4 := i_3 + 1$; goto l'_2
l''' :	return s	l'' :	return s_3

Figure 4.: Example of a transformation into SSA form

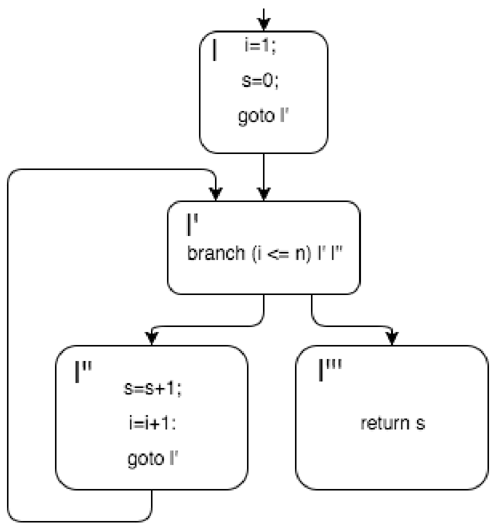


Figure 5.: CFG corresponding to example presented in Figure 1

1.4 CONTINUATION PASSING STYLE

In the functional paradigm, *continuation-passing style* (CPS) is a style of programming in which control is passed explicitly in the form of *a continuation*. This is not the style usually used by programmers (generally called the *direct style*). Continuation-passing style is used as an intermediate representation to perform optimisations and program transformations. CPS has been used as an intermediate language in compiler for functional languages, see Appel (2007).

A function written in CPS takes an extra argument, the *continuation function*, which is a function of one argument. When the CPS function comes to a result value, it returns it by calling the continuation function with this value as the argument. The *continuation* represents what should be done with the result of the function being calculated. This feature, along with some other restrictions on the form of expressions, makes a number of things explicit (such as function returns, intermediate values, the order of argument evaluation and tail calls) which are implicit in direct style. λ-*calculus* in CPS, as an intermediate representation, is used to expose the semantics of programs, making them easier to analyze and the optimization process more efficient for functional-language compilers.

It is well known that the SSA form is closely related to λ-*terms* . (Kelsey, 1995) pointed to a correspondence between programs in SSA form and λ-*terms* in CPS. In CPS there is exactly one binding form for every variable and variable uses are lexically scoped. In SSA there is exactly one assignment statement for every variable and that statement dominates all uses of the variable.

In Kelsey (1995), Kelsey define syntactic transformation that convert CPS programs into SSA and vice-versa. Some CPS programs cannot be converted to SSA but these are not pro duced by the usual CPS transformation. The transformations from CPS into SSA is especially helpful for compiling functional programs. Many optimizations that normally require flow analysis can be performed directly on functional CPS programs by viewing them as SSA programs.

1.5 CONTRIBUTIONS AND DOCUMENT STRUCTURE

In this work we study the central ideas of static single assignment programs in a context of a simple imperative language including jump instructions. The fundamental contribution of this thesis is the proof of correctness of a translation from programs of the base imperative language into the SSA format. In particular,

- we defined small-step operational semantics for the base imperative language and for the corresponding SSA language;
- we defined a translation from programs of the base language into SSA-programs, and prototyped it in Haskell in `https://goo.gl/PtfqGJ`;
- we developed alternative operational semantics, both for the base imperative language and for the SSA language, which split the computation, according to whether it involves a jump or not, and keeps track of information about the variables in order to identify the adequate version of each variable, when relating execution of a base program and of its SSA translation;
- we proved each of these alternative semantics equivalent to the corresponding small-step semantics;

- we proved soundness and completeness results for the translation, relatively to the alternative operational semantics, and from these results we proved correctness of the translation relatively to the initial small-step semantics.

The thesis is organized as follows: In Chapter 2 we formally introduce the syntax and the semantics of the base imperative language and the SSA language that are used in this work. In Chapter 3 we present in every detail a function that translates from base imperative programs into SSA-programs. Chapter 4 is entirely devoted to the proof of correctness of the translation defined. In Chapter 5 we discuss some possible improvements to the translation function and the relation of SSA-programs and the functional programming paradigm. Finally we draw some conclusions and directions for future work.

2

IMPERATIVE LANGUAGE AND SINGLE ASSIGNMENT FORM

In this chapter we introduce a base imperative language and a SSA language that will be used in our study. For the sake of simplicity we will refer to SSA format just as SA form. We will call $\mathcal{GL}$ (short for *Goto Language*) to the imperative language. Let us begin by introducing some general notation used for functions and lists.

NOTATION: Given a function f, $\mathsf{dom}(f)$ denotes the domain of f and $f[x \mapsto a]$ denotes the function that maps x to a and any other value y to $f(y)$. We also use the notation $[x \mapsto g(x) | \ x \in A]$ to represent the function with domain A generated by g. For a set $B \subseteq \mathsf{dom}(f)$, $f(B) = \{f(x) \mid x \in B\}$.
For every A, we let A^* denote the set of lists of elements of A, and $[\,]$ denote the empty list. $\vec{a}$ ranges over A^+, if a ranges over A. $\#\vec{a}$ denotes the length of $\vec{a}$. $\vec{a}[n]$ with $n \leq \#\vec{a}$ denotes the n^{th} element of $\vec{a}$. For convenience, we will sometimes write non-empty lists in the form $h : t$, where h denotes the first element of the list (its *head*) and t the rest of it (its *tail*).
We use the $+\!\!+$ infix operator for the concatenation of two lists.
Given a total function $f : X \to Y$ and a partial function $g : X \rightharpoonup Y$, 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.

2.1 THE GOTO LANGUAGE $\mathcal{GL}$

$\mathcal{GL}$ is a simple "goto" language whose programs are defined as sequences of labeled blocks with an entry point. Blocks are sequences of assignments that end with a jump or a return instruction, if the program ends in the block. Labels are used as names for the blocks.

2.1.1 *Syntax*

For our imperative language, $\mathcal{GL}$, we have various categories and for each one we have meta-variables that will be used to range over constructs of each category. **Num** denotes numerals, *i.e.*, some set of encoding of integers and we will have letter n to range over **Num**. **Var** denotes a set of variables which is assumed to be infinite. Small letters such as $x, y, z, \ldots$ will denote variables and range over **Var**. To represent the labels, we have $l, l', l'' \ldots$ that will range over **L** (set of all the labels). Finally, **Aexp** represents the set of arithmetic expressions and **Cexp** the set of conditional expressions. The abstract syntax for arithmetic and conditional expressions is presented in the following definition:

Definition 1 (Abstract syntax for Expressions) *For* $x \in \mathbf{Var}$ *and* $n \in \mathbf{Num}$*:*

$$\begin{array}{lll} \mathbf{Aexp} \ni a & ::= n \mid x \mid a + a' \mid a \times a' \mid a - a' \\ \mathbf{Cexp} \ni c & ::= \neg c \mid c \vee c' \mid c \wedge c' \mid a = a' \mid a \leq a' \mid a \geq a' \end{array}$$

We will use a, a',... to range over **Aexp** and c, c',... to range over **Cexp**. For a (or c) an arithmetic (or conditional) expression, we write $\mathsf{varsE}(a)$ (or $\mathsf{varsE}(c)$) to denote the set of variables occurring in a (or c). In the next definition, we will introduce the abstract syntax for programs and blocks for $\mathcal{GL}$. We will use **P** to denote the set of programs of $\mathcal{GL}$ and we will use **B** to denote the set of blocks of $\mathcal{GL}$. In what follows, we will use letters $p, p_1, \ldots$ to range over programs (in **P**) and letter $b, b_1 \ldots$ to range over blocks (in **B**).

Definition 2 (Abstract Syntax for blocks and programs) *Let* $a \in \mathbf{Aexp}$, $c \in \mathbf{Cexp}$ *and* $l \in \mathbf{L}$:

$$\begin{array}{lll} \mathbf{P} \ni p & ::= b\,(l : b)^* \\ \mathbf{B} \ni b & ::= x := a\,;b \\ & \quad \mid \mathtt{return}\, a \\ & \quad \mid \mathtt{goto}\, l \\ & \quad \mid \mathtt{branch}\ c\ l\ l' \end{array}$$

Informally speaking, a program starts with one block that is not named by a label, representing the entry point of the program. A block is a sequence of assignments that ends with a `return`, `goto` or `branch` instruction.

An instruction $x := a$ represents an assignment of the arithmetic expression a to variable x and $x := a\,; b$ represents a sequence of instructions: first the assignment is performed and then the rest of the block b is executed.

A variable x is *read* in a block b if there is in b an instruction:

- $y := a$ and $x \in \mathsf{varsE}(a)$

- `return` a and $x \in \mathsf{varsE}(a)$
- `branch` $c\ \ l\ l'$ and $x \in \mathsf{varsE}(c)$

We say that a variable is *used* in a block b if is read or assigned in that block. The blocks, **B**, are a sequences of assignments that end with a jump instruction in which:

- `goto` l represents a *jump* from the current block to the block labeled l;
- `branch` $c\ \ l\ l'$ represents a conditional jump, which means it only jumps to the block labeled l if c is evaluated to `true`, otherwise it jumps to the block labeled l';
- `return` a instruction finishes execution and returns expression a as "result".

We say that a label is *defined* in a program p when it is the identifier/name for some block b. We write $(l : b) \in p$ to denote that in program p there is a block b labeled with l. We say that a label is *invoked by an instruction* in a program p, if there is a `goto` or `branch` instruction that uses the label as argument, *i.e.*, we say that l is *invoked* in a block b if b contains an instruction `branch` $c\ \ l\ l'$, `branch` $c\ \ l'\ l$ or `goto` l. We say that l is *invoked* in a program p, if it is invoked in a block of p.

We will be interested only in a subset of **P**, well formed programs that we define now.

Definition 3 (Well-formed program) *Let $p \in \mathbf{P}$. We say that p is* well-formed*, denoted* $\mathsf{wfProg}(p)$*, if:*

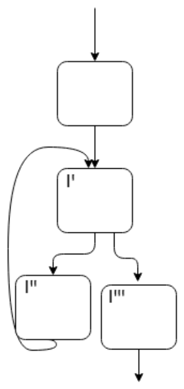
1. *each label is defined at most once;*
2. *each label invoked in an instruction of p, is defined in p.*

Note that, for a well-formed program p, if a label l is *defined* we use $(l : b) \in p$ to identify the block b associated to l.

A control flow graph (CFG) is a graphical representation, of all paths that might be traversed through a program during its execution. Let us give a small example of a well formed program.

Example 1 *Bellow we present a very small program p_1 in $\mathcal{GL}$ and sketched its control flow graph. Note that the program is well formed since every label is declared only once.*

$$\begin{array}{ll}
 & y := w; \\
 & x := 0; \\
 & \texttt{goto}\, l'; \\
l' : & \texttt{branch}\ y \leq 4\ l''\ l''' \\
l'' : & x := y; \\
 & \texttt{goto}\, l' \\
l''' : & z := x + y; \\
 & \texttt{return}\, z
\end{array}$$



2.1.2 *Semantics*

The semantics of $\mathcal{GL}$ programs will be given using a small-step operational semantics. The focus is on individual steps of the execution.
In this subsection, we will define the operational semantics of the goto language $\mathcal{GL}$ in terms of a *transition relation* on *configurations*.
Configurations will capture the fundamental information about execution of a program at a given instant. Namely, the instruction that is being executed and the values for the variables of the program. The transition relation will then specify how the execution of each instruction changes configurations.

The meaning of conditional and arithmetic expressions depend on the values of variables that may occur in the expressions. The semantics is given in terms of states. In other words, the interpretation of expressions depends on a state which is a function that maps each variable of the program into an integer. We will write

$$\Sigma = \mathbf{Var} \to \mathbb{Z}$$

for the set of states. We let $s, s_1, s_2...$ range over Σ. As expected, the value of a variable x in some state s is represented by $s(x)$.

In order to interpret expressions we also need to interpret constants in $\mathbf{Num}$. To keep notation light we simply use the same notation for constants and for its interpretation, an integer.

Definition 4 (Semantics of expressions) *The meaning of arithmetic expressions is a function:*

$$\mathcal{A} : \mathbf{Aexp} \to (\Sigma \to \mathbb{Z})$$

defined recursively as follows:

$[\![a]\!] : \Sigma \to \mathbb{Z}$ is recursively defined as:

$$\begin{array}{rcl}
\mathcal{A}[\![n]\!](s) & = & \mathcal{N}[\![n]\!] = n \\
\mathcal{A}[\![x]\!](s) & = & s(x) \\
\mathcal{A}[\![a + a']\!](s) & = & [\![a]\!](s) + [\![a']\!](s) \\
\mathcal{A}[\![a \times a']\!](s) & = & [\![a]\!](s) \times [\![a']\!](s) \\
\mathcal{A}[\![a - a']\!](s) & = & [\![a]\!](s) - [\![a']\!](s)
\end{array}$$

The values for conditional expressions are truth values, so in a similar way we define $\mathcal{C}$:

$$\mathcal{C} : \mathbf{Cexp} \to (\Sigma \to \{\mathtt{T}, \mathtt{F}\})$$

defined recursively as follows:

$$\begin{array}{rcl}
\mathcal{C}[\![a < a']\!](s) & = & \begin{cases} \mathtt{T} & \mathit{if}\ \mathcal{A}[\![a]\!](s) \leq \mathcal{A}[\![a']\!](s) \\ \mathtt{F} & \mathit{otherwise} \end{cases} \\
\mathcal{C}[\![a = a']\!](s) & = & \begin{cases} \mathtt{T} & \mathit{if}\ \mathcal{A}[\![a]\!](s) = \mathcal{A}[\![a']\!](s) \\ \mathtt{F} & \mathit{otherwise} \end{cases} \\
\mathcal{C}[\![\neg c]\!](s) & = & \begin{cases} \mathtt{T} & \mathit{if}\ \mathcal{C}[\![c]\!](s) = \mathtt{F} \\ \mathtt{F} & \mathit{otherwise} \end{cases} \\
\mathcal{C}[\![c \vee c']\!](s) & = & \begin{cases} \mathtt{T} & \mathit{if}\ \mathcal{C}[\![c]\!](s) = \mathtt{T}\ \mathit{or}\ \mathcal{C}[\![c']\!](s) = \mathtt{T} \\ \mathtt{F} & \mathit{otherwise} \end{cases} \\
\mathcal{C}[\![c \wedge c']\!](s) & = & \begin{cases} \mathtt{T} & \mathit{if}\ \mathcal{C}[\![c]\!](s) = \mathtt{T}\ \mathit{and}\ \mathcal{C}[\![c']\!](s) = \mathtt{T} \\ \mathtt{F} & \mathit{otherwise} \end{cases}
\end{array}$$

In general, we write only $[\![c]\!](s)$ and $[\![a]\!](s)$, instead of $\mathcal{C}[\![c]\!](s)$ and $\mathcal{A}[\![a]\!](s)$ when there is no danger of confusion.

The operational semantics of $\mathcal{GL}$, as previously mentioned, is given through a relationship between two configurations. A *configuration* is defined as a pair $\langle b, s\rangle$ for $b \in \mathbf{B}$ and $s \in \Sigma$. Sometimes, we will use γ to represent a configuration. A *terminal configuration* has the form $\langle \mathtt{return}\, a, s\rangle$ for some $a \in \mathbf{Aexp}$ and $s \in \Sigma$.

Next, we will introduce the transition relation in $\mathcal{GL}$ is parameterized by a program p. Such a program does not change during execution but it needs to be consulted to perform jump instructions.

Definition 5 (Transition relation for $\mathcal{GL}$) *Let p be a program. The transition relation induced by p is a binary relation on configurations and is denoted by $\rightsquigarrow_p$ or simply $\rightsquigarrow$ when it is clear what is the program p in mind.*

The pairs in the relation will have the form of $\langle b, s\rangle \rightsquigarrow_p \langle b', s'\rangle$, and the relation is defined by the *following axioms:*

$$\textsf{atrib} \qquad \frac{}{\langle x := a\,; b, s\rangle \rightsquigarrow_p \langle b, s[x \mapsto [\![a]\!](s)]\rangle}$$

$$\textsf{goto} \qquad \frac{(l : b) \in p}{\langle \texttt{goto}\ \ l, s\rangle \rightsquigarrow_p \langle b, s\rangle}$$

$$\textsf{branch}_{\textsf{T}} \qquad \frac{(l : b) \in p \qquad [\![c]\!](s) = \texttt{T}}{\langle \texttt{branch}\ c\ \ l\ l', s\rangle \rightsquigarrow_p \langle b, s\rangle}$$

$$\textsf{branch}_{\textsf{F}} \qquad \frac{(l' : b) \in p \qquad [\![c]\!](s) = \texttt{F}}{\langle \texttt{branch}\ c\ \ l\ l', s\rangle \rightsquigarrow_p \langle b, s\rangle}$$

We represent by $\rightsquigarrow_p^*$ the reflexive and transitive closure of $\rightsquigarrow_p$, that allows to put together a sequence of zero or more transitions.

A configuration $\langle b, s\rangle$ is said to be *blocked* if it is not terminal and there is no configuration $\langle b', s'\rangle$ such that $\langle b, s\rangle \rightsquigarrow_p \langle b', s'\rangle$. Note that the cases where the configurations can be blocked are the cases where labels are not defined.
A *sequence of transitions* generated by a configuration γ_0 starting in a state s is said to be:

- finite, if $\gamma_0 \rightsquigarrow_p^* \gamma_n$ and where γ_n is either a terminal configuration or a blocked configuration;
- infinite, if no γ_n exists.

We write $\gamma_0 \rightsquigarrow_p^k \gamma_k$ to indicate that there are k steps in the sequence of transitions from γ_0 to γ_k.

Definition 6 (Execution of program p in a state s in $\mathcal{GL}$ *w.r.t.* $\rightsquigarrow$) *Let p be a program and b the starting block of p. A* execution of the program p in the state s *is a sequence of transitions (using the transition relation $\rightsquigarrow$) generated by $\langle b, s\rangle$.*

Note that, if p is not well formed, it is possible to have a non-deterministic execution of p for some state s since a label can be declared more than once. Note also that, if p is not well formed, is possible to have a finite execution stopping in a blocked configuration whenever there is an instruction that invoke a label that is not defined.

Definition 7 (Result of an execution of program p in a state s in $\mathcal{GL}$) *Let p be a program and s a state. When the execution of p in state s is finite and the terminal configuration is $\langle \texttt{return}\, a\,,\ s_1\rangle$, we say that $[\![a]\!](s_1)$ is the result of the execution.*

Lemma 1 *Let p be a well formed program. The transition relation $\rightsquigarrow_p$ is deterministic; that is for any $b \in \mathbf{B}\,, s \in \Sigma$ if $\langle b, s\rangle \rightsquigarrow_p \langle b', s'\rangle$ and $\langle b, s\rangle \rightsquigarrow_p \langle b'', s''\rangle$ then $b = b''$ and $s = s''$.*

Proof. If follows from item 1 of Definition 3 and with induction on the derivation. □

Proposition 1 *Let p be a well-formed program.*

(i) The execution of program p in state s is deterministic*: given a state $s \in \Sigma$, exists exactly one sequence of transitions generated by $\langle b, s\rangle$ where b is the starting block of program p;*

(ii) The execution of program p in state is finite: if it ends, it ends in a terminal configuration.

Proof. It follows directly from Lemma 1. Also, it is a direct consequence of item 2 of Definition 3. □

Example 2 *The execution of the program p_1 of Example 1 with a initial state s_0 such that $s_0(w) = 5$ is:*
$\langle y := 5; x := 0; \texttt{goto}\, l', s_0\rangle \rightsquigarrow_{p_1} \langle x := 0; \texttt{goto}\, l', s_1\rangle \rightsquigarrow_{p_1} \langle \texttt{goto}\, l', s_2\rangle \rightsquigarrow_{p_1}$
$\langle \texttt{branch}\ y \leq 4\ l''\ l''', s_2\rangle \rightsquigarrow_{p_1} \langle z := x + y;\ \texttt{return}\, z, s_2\rangle \rightsquigarrow_{p_1} \langle \texttt{return}\, z, s_3\rangle$
where: $s_1 = s_0[y \mapsto 5]$, $s_2 = s_1[x \mapsto 0]$ *and* $s_3 = s_2[z \mapsto 5]$
In this example, the result of the execution is 5.

Example 3 *The execution of the program p_1 of Example 1) with a initial state s_0 such that $s_0(w) = 3$ is:*
$\langle y := 3; x := 0; \texttt{goto}\, l', s_0\rangle \rightsquigarrow_{p_1} \langle x := 0; \texttt{goto}\, l', s_1\rangle \rightsquigarrow_{p_1} \langle \texttt{goto}\, l', s_2\rangle \rightsquigarrow_{p_1}$
$\langle \texttt{branch}\ y \leq 4\ l''\ l''', s_2\rangle \rightsquigarrow_{p_1} \langle x := y;\ \texttt{goto}\, l', s_2\rangle \rightsquigarrow_{p_1} \langle \texttt{branch}\ y \leq 4\ l''\ l''', s_2\rangle \rightsquigarrow_{p_1} \langle x := y;\ \texttt{goto}\, l', s_2\rangle \rightsquigarrow_{p_1} (...)$
where: $s_1 = s_0[y \mapsto 3]$ *and* $s_2 = s_1[x \mapsto 0]$.
Therefore, the program loops.

2.2 SA LANGUAGE

We will now present a variant of the programs in **P** that are in SSA form. For the sake of simplicity, we will call those single assignment (SA) programs and represent them by $\mathbf{P^{SA}}$. The basic idea in the SA programs is to create versions of variables so that a version of a variable does not get assigned more than once in a SA block.

2.2.1 *Syntax*

In this subsection, we present a variant of the $\mathcal{GL}$ where the program is in single assignment form.

Syntactically, relatively to the $\mathcal{GL}$, there are essentially three differences:

1. Variables now come with an index over $\mathbb{N}$. We let a $\mathbf{Var}^{\mathbf{SA}} = \mathbf{Var} \times \mathbb{N}$ be the set of SA variables and we will write x_i to denote $(x, i) \in \mathbf{Var}^{\mathbf{SA}}$;

2. Similarly, labels come with an index over $\mathbb{N}$. We let a $\mathbf{L}^{\mathbf{SA}} = \mathbf{L} \times \mathbb{N}$ be the set of SA labels and we will write l_i to denote $(l, i) \in \mathbf{L}^{\mathbf{SA}}$;

3. There is a new ingredient that, as usually in the literature, we call *ϕ-functions*, which syntactically we write as $\phi(\vec{x_j})$ where $\vec{x_j}$ is a vector of SA variables, and semantically will be responsible to control what is the correct version of a variable to use at a given point of program execution.

   More specifically, the *ϕ-functions* let us decide which version of the variable should we use by knowing from which block came the edge. The k'th argument of $\phi(\vec{x_j})$ will be used when control reach the block from `goto` l_k . In the rest of the document, instead of using the word *edge* we will normally use the word *door*, so `goto` l_k can be read as: execute the block l knowing we arrive at it through door k.

   The synchronization of variables will happen at the beginning of the blocks. An SA block will start with a list of assignments of *ϕ-functions*, which we generally represent by ϕ and let Φ denote the set of lists of arguments of ϕ-functions. We denote by $\mathsf{dom}\,(\phi)$ the set of variables assigned in ϕ. For instance, $\mathsf{dom}\,(x_1 := \phi(x_5, x_3), y_2 := \phi(y_1, y_4)) = \{x_1, y_2\}$.

The specification of SA Programs is captured by the abstract syntax:

Definition 8 (Abstract syntax for SA programs)

$$
\begin{array}{rl}
\Phi \ni \phi & ::= (x_i := \phi(\vec{x_j}))^* \\
\mathbf{P}^{\mathbf{SA}} \ni p' & ::= b(l : \phi b)^* \\
\mathbf{B}^{\mathbf{SA}} \ni b' & ::= x_i := a \ ; b \\
 & \mid \texttt{return}\, a \\
 & \mid \texttt{goto}\, l_i \\
 & \mid \texttt{branch}\ c\ \ l_i\ \ l'_j
\end{array}
$$

We will use $p', p'', ...$ to represent SA programs (in $\mathbf{P}^{\mathbf{SA}}$) and $b', b''..$ to represent SA blocks (in $\mathbf{B}^{\mathbf{SA}}$). Recall that $p, p_1, ...$ are used for programs in $\mathbf{P}$.

Example 4 *Example of program p'_1 - SA version of program p_1 presented in Example 1:*

```
        $y_1 := w;$
        $x_1 := 0;$
        goto $l'_1$
$l'$ :  branch $y_1 \leq 4$ $l''_1$ $l'''_1$
$l''$ : $x_2 := y_1;$
        goto $l'_2$ ;
$l'''$ : $x_3 := \phi(x_2, x_1);$
        $z_1 := x_3 + y_1;$
        return $z_1$
```

Definition 9 (Well-formed SA program) *Let $p' \in \mathbf{P^{SA}}$. We say that p' is* well-formed, *denoted* $\mathsf{wfSAProg(p')}$, *if:*

1. *each label is defined exactly once in p';*
2. *each label l_i invoked in a* `goto` *or* `branch` *statement of p' is unique in p';*
3. *if label l_i is invoked in a* `goto` *or* `branch` *statement in p', l is declared in p';*
4. *if label l_i is invoked in a* `goto` *or* `branch` *statement in p', so is l_j(and $0 < j < i$);*
5. *if $(l : \phi b') \in p'$, the arity of ϕ is equal to the number of edges that lead to block b';*
6. *every variable is assigned at most once.*

These conditions imply that each label is numbered sequentially. Consider the CFG of the program (see example 1). Beside the first block, every block can have income and outcome edges. With these conditions, we know that each edge represents a jump. If a block ends with a `goto`, then it will be one edge out of that block, and if it ends with a `branch`, there will be two edges out of the block.

2.2.2 *Operational semantics in SA programs*

The semantics for SA programs will be given in terms of a transition relation in a configuration and its focus in on individual steps. The interpretation of expressions in SA depends on the state and the state, as for the base programs, maps each variable of the program (in SA) in $\mathbb{Z}$.

$$\Sigma^{\mathbf{SA}} : \mathbf{Var}^{\mathbf{SA}} \rightarrow \mathbb{Z}$$

We let $s', s'', ...$ range over $\Sigma^{\mathbf{SA}}$. Expressions can also be arithmetic or conditional but now all the variables come with an index. A *configuration* is a pair $\langle b', s' \rangle$ for $b' \in \mathbf{B^{SA}}$ and $s' \in \Sigma^{\mathbf{SA}}$.

A *terminal configuration* has the same form of a terminal configuration in $\mathcal{GL}$: $\langle \texttt{return}\, a\ , s' \rangle$ for $s \in \Sigma^{\mathbf{SA}}$ and $a\ \in \mathbf{Aexp}$. For the constants in $\mathbf{Num}$, we will use the same notation: an integer.

Next, we will introduce the transition relation in SA Programs parameterized by a SA program p'. This relation is analogous to the transition relation in $\mathcal{GL}$ for a program p. The difference is in the way of denoting the variables, which now have an index.

Definition 10 (Operational semantics to SA programs) *Let p' be a well formed program in SA. The transition relation induced by p' is binary and is represented by $\rightsquigarrow_{p'}$. The pairs have the form by $\langle b', s' \rangle \rightsquigarrow_{p'} \langle b'', s'' \rangle$. This transition relation is defined by the following axioms:*

$$\textsf{atrib} \qquad \frac{}{\langle x_i := a\,; b, s\rangle \rightsquigarrow_{p'} \langle b, s[x_i \mapsto [\![a]\!](s)]\rangle}$$

$$\textsf{goto} \qquad \frac{(l : \phi\, b') \in p'}{\langle \texttt{goto}\ \ l_i, s\rangle \rightsquigarrow_{p'} \langle b, \textsf{upd}\,(\phi, i, s))\rangle}$$

$$\textsf{branch}_{\textsf{T}} \qquad \frac{(l : \phi\, b') \in p' \qquad [\![c]\!](s) = \texttt{true}}{\langle \texttt{branch}\ c\ \ l_i\ l'_j, s\rangle \rightsquigarrow_{p'} \langle b, \textsf{upd}\,(\phi, i, s))\rangle}$$

$$\textsf{branch}_{\textsf{F}} \qquad \frac{(l : \phi\, b') \in p' \qquad [\![c]\!](s) = \texttt{false}}{\langle \texttt{branch}\ c\ \ l_i\ l'_j, s\rangle \rightsquigarrow_{p'} \langle b, \textsf{upd}\,(\phi, j, s))\rangle}$$

where upd *is the* update *function defined as follows:*

$$\begin{aligned} &\textsf{upd}\ : \Phi \times \mathbb{N} \times \Sigma^{\mathbf{SA}} \to \Sigma^{\mathbf{SA}} \\ &(\phi, n, s) \longmapsto s[x \mapsto [\![\vec{x}[n]]\!](s) | x := \phi(\vec{x}) \in \phi] \end{aligned}$$

The upd function select the right version of the variable in the beginning of each block. Note that, because $p' \in \mathbf{P}^{\mathbf{SA}}$ is well-formed, the upd function is well-defined (item 5 of Definition 9).

Lemma 2 *Let p' be a well formed program. The transition relation $\rightsquigarrow_{p'}$ is deterministic; that is for any $b' \in \mathbf{B}^{\mathbf{SA}}, s' \in \Sigma^{\mathbf{SA}}$ if $\langle b', s' \rangle \rightsquigarrow_p \langle b'', s'' \rangle$ and $\langle b', s' \rangle \rightsquigarrow_p \langle b''', s''' \rangle$ then $b'' = b'''$ and $s'' = s'''$.*

Proof. If follows directly from Item $1, 2$ and 3 of Definition 9. □

Definition 11 (Execution of program p' in a state s' in SA *w.r.t.* $\rightsquigarrow$**)** *Let p' be a SA program and b' the starting block of p'. A* execution of the program p' in the state s' *is a sequence of transitions using the transition relation $\rightsquigarrow$ generated by $\langle b', s' \rangle$.*

Note that, if p' is not well formed, it is possible to have a variable assigned more than once, and it is possible to have variables that were not in the ϕ domain. Moreover, it is possible to have a jump instruction in p' that leads to no block, since it is possible to invoke a label that is not defined. Similarly to programs in $\mathcal{GL}$, if p' is not well formed it is possible to have a non-deterministic execution of p' for some state s' since a label can be declared more than once.

Definition 12 (Result of an execution of program p in a state s in SA) *Let p' be a program in SA and s' a state. When the execution of p' in state s' is finite and the terminal configuration is $\langle \texttt{return}\, a' , s'_1 \rangle$, we say that $[\![a']\!](s'_1)$ is the result of the execution.*

Proposition 2 *Let p' be a well-formed SA program.*

(i) The execution of program p' in state s' is deterministic*: given a state $s' \in \Sigma^{\mathbf{SA}}$, exists exactly one sequence of transitions generated by $\langle b', s' \rangle$ were b' is the starting point of p';*

(ii) If the execution of program p' in state s' is finite, then it ends in a terminal configuration.

Proof. It follows directly from Lemma 2. □

Example 5 *The execution of p'_1 (in Example 4) with the initial state s_0 such that $s_0(w) = 5$ is:*
$\langle y_1 := 5; x_1 := 0; \texttt{goto}\, l'_1 , s_0 \rangle \rightsquigarrow_{p'_1} \langle x_1 := 0; \texttt{goto}\, l'_1 , s_1 \rangle \rightsquigarrow_{p'_1} \langle \texttt{goto}\, l'_1 , s_2 \rangle \rightsquigarrow_{p'_1}$
$\langle \texttt{branch}\ y_1 \leq 4\ l''_1\ l'''_1 ,\ s_2 \rangle \rightsquigarrow_{p'_1} \langle x_3 := \phi(x_2, x_1); z_1 := x_3 + y_1;\ \texttt{return}\, z_1 , s_3 \rangle \rightsquigarrow_{p'_1} \langle z_1 := x_3 + y_1;\ \texttt{return}\, z_1 , s_3 \rangle \rightsquigarrow_{p'_1} \langle \texttt{return}\, z_1 , s_4 \rangle$, *where* $s_1 = s_0[y \mapsto 5]$, $s_2 = s_1[x \mapsto 0]$ *and* $s_3 = s_2[z \mapsto 5]$ *and* $s_4 = s_3[x_3 \mapsto s_3(x_2)]$

3

A TRANSLATION INTO SA FORM

In this chapter we present an algorithm to transform $\mathcal{GL}$ programs into SA format. The algorithm has been implemented in Haskell (the code is in Appendix A). We will prove the correctness of the translation in the next chapter.
Throughout this chapter we assume that the programs in **P** are always well formed.

3.1 TRANSLATING INTO SA FORM

A program in **P** has an entrance block (with no label associated) and a sequence of labeled blocks. In order to uniform the translation algorithm, we will introduce a distinguished label, denoted by $\bullet$, that we will associate to the entrance block. Of course the special label $\bullet$ can not be invoked in the program.

The programs we translate belong to the class $\mathbf{P}_\bullet$ defined by:

$$\mathbf{P}_\bullet \ni p ::= (\bullet : b)(l : b)^*$$

The SA programs produced by the translation will belong to the class $\mathbf{P}_\bullet^{\mathbf{SA}}$ defined by:

$$\mathbf{P}_\bullet^{\mathbf{SA}} \ni p' ::= (\bullet : b')(l : \phi b')^*$$

The conversion of a program from **P** to $\mathbf{P}_\bullet$ and from $\mathbf{P}_\bullet^{\mathbf{SA}}$ to $\mathbf{P}^{\mathbf{SA}}$ is trivial (just add/remove the label $\bullet$) and deserve no further comments. We let $\mathbf{L}_\bullet$ represent the set of labels enriched with the distinguished label $\bullet$, *i.e.*, $\mathbf{L}_\bullet = \mathbf{L} \cup \{\bullet\}$. Programs in $\mathbf{P}_\bullet$ can be seen as elements of $(\mathbf{L}_\bullet \times \mathbf{B})^*$.

The translation from $\mathbf{P}_\bullet$ into $\mathbf{P}_\bullet^{\mathbf{SA}}$ is performed by function $\mathcal{T}$ and is done in two steps. In the first step we create an intermediate structure composed by a *quasi* SA program (SA program without ϕ-functions) - $\mathbf{P}_\bullet^{\mathbf{I}} = (\mathbf{L}_\bullet \times \mathbf{B}^{\mathbf{SA}})^*$ - and a context that registers information of the translation process. The second step uses this intermediate structure to add the ϕ-functions to the *quasi* SA program, producing a program in $\mathbf{P}_\bullet^{\mathbf{SA}}$ .The translation process is illustrated in Figure 6. We name the first step $\mathcal{R}$ (after *renaming*) and the second step $\mathcal{S}$ (after *synchronization*).

The $\mathcal{R}$ function does three different tasks:

$$\begin{tikzcd}
\mathbf{P} \arrow[r] & \mathbf{P}_{\bullet} \arrow[r, "\mathcal{R}"] \arrow[rr, bend right, "\mathcal{T}"'] & \mathbf{P}^{\mathbf{I}}_{\bullet} \times \mathbf{C} \arrow[r, "\mathcal{S}"] & \mathbf{P}^{\mathbf{SA}}_{\bullet} \arrow[r] & \mathbf{P}^{\mathbf{SA}}
\end{tikzcd}$$

Figure 6.: Translation into SA format

1. Converts the code of each block to SA form. This is done by tagging the variable with adequate indexes (versions);
2. Assigns a sequential number the incoming edge (or door) of each block.

The $\mathcal{S}$ function is devoted to the synchronization task. Its only purpose is to build at the beginning of each block the assignments of *ϕ-functions* to the variables.

To implement the first step of the translation, we need to know:

- The current version of each variable. For that we need a function that maps each variable identifier to a non-negative integer. For a program $p \in \mathbf{P}$ we let $\mathbf{VS}_p = \mathbf{Var}_p \rightarrow \mathbb{N}$ where$\mathbf{Var}_p$ denotes the set of variables of a program p. For the sake of simplicity, whenever it is clear what is the program p being translated, we drop the subscript p and write simply $\mathbf{VS}$. We let $\mathcal{V}$, $\mathcal{V}$ ', ... range over $\mathbf{VS}$. We call *version functions* to the elements of $\mathbf{VS}$.
  Given a function $\mathcal{V} = \mathbf{Var} \rightarrow \mathbb{N}$, we define the function $\widehat{\mathcal{V}} = \mathbf{Var} \rightarrow \mathbf{Var}^{\mathbf{SA}}$ as being such that $\widehat{\mathcal{V}}(x) = x_{\mathcal{V}(x)}$. $\widehat{\mathcal{V}}$ is *lifted* to $\mathbf{Aexp}$ and $\mathbf{Cexp}$ in the obvious way, renaming the variables according to $\mathcal{V}$. Given a state s and a version $\mathcal{V}$, $\widehat{\mathcal{V}}(s)$ is the function $\widehat{\mathcal{V}}(s) = \mathbf{Var}^{\mathbf{SA}} \rightarrow \mathbb{Z}$ that for all $x_i \in \mathbf{Var}^{\mathbf{SA}}$, $\widehat{\mathcal{V}}(x_i) = s(x)$. For any label l, $\widehat{\mathcal{V}}(l) = l$.

  Furthermore, given a version function $\mathcal{V} \in \mathbf{VS}$, we define an auxiliary function $\mathsf{inc} :: \mathbf{VS} \rightarrow \mathbf{VS}$ that increments the version of all the variables.

- The current number of edges arriving to a labeled block, *i.e.*, the number of already discovered doors of a labeled block, the version function associated to each door of a labeled block and the version of the variables to be synchronized. For a program $p \in \mathbf{P}$ we let $\mathbf{LS}_p = \mathbf{L}_p \rightarrow \mathbb{N} \times \mathbf{VS}_p^* \times \mathbf{VS}_p$ where $\mathbf{L}_p$ denotes the set of labels of a program p. For the sake of simplicity, whenever it is clear what is the program p being translated, we drop the subscript p and write simply $\mathbf{LS}$. We let $\mathcal{L}$, $\mathcal{L}$ ', ... range over $\mathbf{LS}$. We call *label functions* to the elements in $\mathbf{LS}$.

We aggregate these informations in what we call a *context*. We let $\mathbf{C} = \mathbf{VS} \times \mathbf{LS}$ and let $\mathcal{C}$, $\mathcal{C}$ ', ... range over $\mathbf{C}$.

In the beginning of the translation, the context is initialized (by function initC) and then incrementally built in the translation process, as we go through the given program in $\mathbf{P}_{\bullet}$.

- The version function starts with all variables with version equal to -1. The versions are incremented at the beginning of each block, so the first version in use will be 0;

$$
\begin{array}{l}
\mathcal{R} : \mathbf{P}_{\bullet} \to \mathbf{P}^{\mathbf{I}}_{\bullet} \times \mathbf{C} \\
\mathcal{R}\, p = \mathsf{T_L}\, p\ (\mathsf{initC}\, p) \\
\\
\mathsf{initC} : \mathbf{P}_{\bullet} \to \mathbf{C} \\
\mathsf{initC}\, p = (\mathcal{V}_0, [l \mapsto (0, [], \mathcal{V}_0) \mid l \in \mathsf{labels}(p)] \\
\quad \text{where} \quad \mathcal{V}_0 \ = [x \mapsto -1 \mid x \in \mathsf{vars}(p)] \\
\\
\mathsf{T_L} : (\mathbf{L}_{\bullet} \times \mathbf{B})^* \to \mathbf{C} \to (\mathbf{L}_{\bullet} \times \mathbf{B}^{\mathbf{SA}})^* \times \mathbf{C} \\
\mathsf{T_L}\, ((l,b) : t)(\mathcal{V},\mathcal{L}) = ((l,b') : t', \mathcal{C}'') \\
\quad \text{where} \quad
\begin{array}{rl}
(n,d,\_) & = \mathcal{L}(l) \\
(b',\mathcal{C}') & = \mathsf{T_B}\,(b,l)(\mathsf{inc}\,\mathcal{V}, \mathcal{L}\,[l \mapsto (n,d,\mathsf{inc}\,\mathcal{V})]) \\
(t',\mathcal{C}'') & = \mathsf{T_L}\ t\ \mathcal{C}'
\end{array} \\
\mathsf{T_L}\ [\,]\,\mathcal{C} = ([\,],\mathcal{C}) \\
\\
\mathsf{T_B} : \mathbf{B} \times \mathbf{L} \to \mathbf{C} \to \mathbf{B}^{\mathbf{SA}} \times \mathbf{C} \\
\mathsf{T_B}\,(x := a; b, l)\ (\mathcal{V},\mathcal{L}) = (x_{\mathcal{V}(x)+1} := \widehat{\mathcal{V}}\,(a); b', \mathcal{C}') \\
\quad \text{where} \quad
\begin{array}{rl}
\mathcal{C} & = (\mathcal{V}\,[x \mapsto \mathcal{V}\,(x)+1], \mathcal{L}) \\
(b',\mathcal{C}') & = \mathsf{T_B}\,(b,l)\ \mathcal{C}
\end{array} \\
\mathsf{T_B}\,(\mathtt{goto}\, l', l)\ (\mathcal{V},\mathcal{L}) = (\mathtt{goto}\, l'_{n+1}, \mathcal{C}) \\
\quad \text{where} \quad
\begin{array}{rl}
(n,d,\mathcal{V}_0) & = \mathcal{L}\,(l') \\
\mathcal{C} & = (\mathcal{V}, \mathcal{L}\,[l' \mapsto (n+1, d \mathbin{+\!\!+} [\mathcal{V}\,], \mathcal{V}_0)])
\end{array} \\
\mathsf{T_B}\,(\mathtt{return}\, a, l)\ (\mathcal{V},\mathcal{L}) = (\mathtt{return}\, \widehat{\mathcal{V}}\,(a), (\mathcal{V},\mathcal{L})) \\
\mathsf{T_B}\,(\mathtt{branch}\ c\ l'\ l'', l)\ (\mathcal{V},\mathcal{L}) = (\mathtt{branch}\ \widehat{\mathcal{V}}\,(c)\ l'_{n+1}\ l''_{n'+1}, \mathcal{C}) \\
\quad \text{where} \quad
\begin{array}{rl}
(n,d,\mathcal{V}_0) & = \mathcal{L}\,(l') \\
\mathcal{L}' & = \mathcal{L}\,[l' \mapsto (n+1, d \mathbin{+\!\!+} [\mathcal{V}\,], \mathcal{V}_0)] \\
(n',d',\mathcal{V}'_0) & = \mathcal{L}'(l'') \\
\mathcal{L}'' & = \mathcal{L}'[l'' \mapsto (n'+1, d' \mathbin{+\!\!+} [\mathcal{V}\,], \mathcal{V}'_0)] \\
\mathcal{C} & = (\mathcal{V}, \mathcal{L}'')
\end{array}
\end{array}
$$

Figure 7.: Function $\mathcal{R}$ and its auxiliary functions

- The label function begins with the labels with the respective number of discovered doors equal to 0, an empty list of version functions and a version function which we assume to have -1 associated to each variable (although the concrete number associated to each variable is irrelevant at this point).

The initC function uses auxiliary functions that we omit for the sake of simplicity (one can see the details in Appendix A). For a program p, $\mathsf{vars}(p)$ collects every variable of the program and $\mathsf{labels}(p)$ collects every label of the program.

Function $\mathcal{R}$, presented in Figure 7, is defined using a mixture of mathematical notation and Haskell-like syntax. This function relies on two auxiliary functions: $\mathsf{T_L}$ and $\mathsf{T_B}$. We give a brief description of each of then:

$$
\begin{array}{l}
\mathsf{sync} :: \mathbf{L} \to \mathbf{C} \to \Phi \\
\mathsf{sync}\ l\ (\_, \mathcal{L}) = [\, x_{\mathcal{V}(x)} := \phi([x_{\mathcal{V}'(x)} \mid \mathcal{V}' \leftarrow d]) \mid x \in \mathsf{dom}\,(\mathcal{V})\,] \\
\quad \text{where} \quad (\_, d, \mathcal{V}) \quad = \mathcal{L}\,(l) \\
\\
\mathcal{S} :: \mathbf{P}^{\mathbf{I}}_{\bullet} \times \mathbf{C} \to \mathbf{P}^{\mathbf{SA}}_{\bullet} \\
\mathcal{S}\,([\,],\ \_) = [\,] \\
\mathcal{S}\,((\bullet, b) : t,\ \mathcal{C}) = (\bullet, [\,], b) : \mathcal{S}\,(t,\ \mathcal{C}) \\
\mathcal{S}\,((l, b) : t, \mathcal{C}) = (l, \mathsf{sync}\ l\, \mathcal{C},\ b) : \mathcal{S}\ (t,\ \mathcal{C})
\end{array}
$$

Figure 8.: Functions sync and $\mathcal{S}$

$$
\begin{array}{l}
\mathcal{T} :: \mathbf{P}_{\bullet} \to \mathbf{P}^{\mathbf{SA}}_{\bullet} \\
\mathcal{T}\, p = \mathcal{S}\,(\mathcal{R}\, p)
\end{array}
$$

Figure 9.: Function $\mathcal{T}$

- $\mathsf{T_L}$ iterates over the program, translating each block and *carrying* the corresponding auxiliary context. It begins by incrementing the version of each variable, in order to spare a version for synchronization. This task is performed by the auxiliary function inc. The synchronization version of the block being translated is registered in the label function, by associating it to the label of the block.

- $\mathsf{T_B}$ starts the translation of a labeled block. $\mathsf{T_B}$ receives the block to be translated, its label and the context. $\mathsf{T_B}$ is responsible for:

  (i) tagging the variables according to the version function and the SA format;

  (ii) tagging the labels of the `goto` and `branch` commands according to the label function.

  Moreover, it updates the context coherently.

After we get the intermediate program (in $\mathbf{P}^{\mathbf{I}}_{\bullet}$), it is necessary to synchronize the variables that can come from different doors. To do that, we must add the *ϕ-functions*. This operation is done by the function $\mathcal{S}$, which can be seen in Figure 8. For the sake of simplicity, we assume that the *ϕ-functions* sequence follows some predefined order established over the set of variables (any order will do).

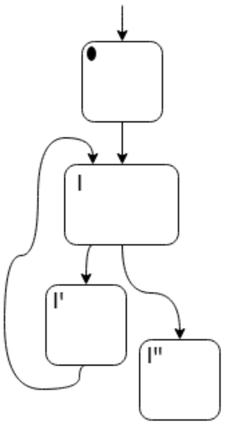
The final result of the translation into SA format is obtained by applying first $\mathcal{R}$ and secondly apply the function $\mathcal{S}$ as stated in Figure 6, *i.e.*, $\mathcal{T} = \mathcal{S} \circ \mathcal{R}$. We can see function $\mathcal{T}$ in Figure 9.

As it can be seen, this translation function synchronizes every variable of the program at the beginning of each block. This is obviously very inefficient and can be optimized. Using static analysis techniques it is possible to calculate the optimal placement of ϕ-functions that leads to a minimum set of ϕ-functions. We did not follow that path because our focus is to prove the correctness of the translation with respect to the operational semantics, and we thought it was better to work with a naive version to begin with. However, we think this translation can be adapted to an optimized version if previously one calculates the optimal placement of the ϕ functions. With this information one would initialize the label function and, in the synchronization phase, the $\mathcal{S}$ function would only write the ϕ-functions previously calculated. We will say more on this in Chapter 8.

Let us now illustrate the translation done by function $\mathcal{T}$ with a small program. The following $\mathcal{GL}$ program calculates the factorial of 5.

Example 6

$$
\begin{array}{ll}
\bullet : & x := 5; \\
 & f := 1; \\
 & c := 1; \\
 & \texttt{goto}\, l \\
l : & \texttt{branch}\ c \leq x\ l'\ l'' \\
l' : & f := f * c; \\
 & c := c + 1; \\
 & \texttt{goto}\, l \\
l'' : & \texttt{return}\, f
\end{array}
$$



We present the intermediate program and final context, after applying function $\mathcal{R}$.

$$
\begin{array}{ll}
\bullet : & x_1 := 5; \\
 & f_1 := 1; \\
 & c_1 := 1; \\
 & \texttt{goto}\, l_1 \\
l : & \texttt{branch}\ (c_2 \leq x_2)\ l'_1\ l''_1 \\
l' : & f_4 := f_3 * c_3; \\
 & c_4 := c_3 + 1; \\
 & \texttt{goto}\, l_2 \\
l'' : & \texttt{return}\, f_5
\end{array}
$$

Final Context:
Version Function: $[("c",5),("f",5),("x",4)]$
Label Function:

$\bullet$	$(0,[],[("c",0),("f",0),("x",0)]$
$"l"$	$(2,[[("c",1),("f",1),("x",1)],[("c",4),("f",4),("x",3)]],[("c",2),("f",2),("x",2)])$
$"l'"$	$(1,[[("c",2),("f",2),("x",2)]],[("c",3),("f",3),("x",3)])$
$"l''"$	$(1,[[("c",2),("f",2),("x",2)]],[("c",5),("f",5),("x",4)])$

The SA version of the program after applying function $\mathcal{S}$, is the following:

$$\begin{array}{ll}
\bullet: & x_1 := 5; \\
& f_1 := 1; \\
& c_1 := 1; \\
& \texttt{goto}\, l_1 \\
l: & c_2 := \phi(c_1, c_4); \\
& f_2 := \phi(f_1, f_4); \\
& x_2 := \phi(x_1, x_3); \\
& \texttt{branch}\ c_2 \leq x_2\ l'_1\ l''_1 \\
l': & c_3 := \phi(c_2); \\
& f_3 := \phi(f_2); \\
& x_3 := \phi(x_2); \\
& f_4 := f_3 * c_3; \\
& c_4 := c_3 + 1; \\
& \texttt{goto}\, l_2 \\
l'': & c_5 := \phi(c_2); \\
& f_5 := \phi(f_2); \\
& x_4 := \phi(x_2); \\
& \texttt{return}\, f_5
\end{array}$$

We now show that T preserves the well-formedness of programs.

Proposition 3 *Let $p \in \mathbf{P}_\bullet$. If p is well formed then $\mathcal{T}$ (p) is also well formed.*

Proof. Assume p is a well-formed program in $\mathbf{P}_\bullet$. We will now argue about the conditions defined in Definition 9:

1. Each label is defined exactly once in $\mathcal{T}$ (p), since $\mathcal{T}$ does not change any name of label and each label was defined exactly one in p;

2. The numbering of the doors of each label is done sequentially, so each label l_i invoked in a `goto` or `branch` statement is unique in $\mathcal{T}$ (p);

3. As p is well formed, each label invoked in p is declared so, since $\mathcal{T}$ does not change any label name, $\mathcal{T}$ (p) does not have undeclared labels;

4. As stated before, the numbering of the doors of each label is done sequentially, so if a label l_i is invoked in a `goto` or `branch` statement in $\mathcal{T}$ (p), so it is every l_j with $0 < j < i$;

5. Inspecting the function sync displayed in Figure 8 we can see that the arity of ϕ-functions is equal to the number of edges recorded in the label function $\mathcal{L}$, because the sync function is invoked with the context produced by $\mathcal{R}$ after processing all the program. $\mathcal{R}$ constructs coherently the context by starting with a context where the number of doors already found for each label is zero and updating this number each time a label is invoked and recording the version function associated to the recorded label;

6. By analyzing function $\mathsf{T_B}$ displayed in Figure 7 one can see that whenever there is an assignment, the variable assigned receives the next index available for that variable and the version function is updated accordingly. This way, each variable SA will be assigned at most once in the block;

□

In the next chapter, we will show that function $\mathcal{T}$ is correct, that is, that the translated program preserves the operational semantics of the original program.

4 CORRECTNESS OF THE TRANSLATION INTO SA FORM

In this chapter we prove that the translation of programs in $\mathcal{GL}$ to SA form, defined in the previous chapter, is correct relatively to the defined semantics, that is: the execution of a source program in $\mathcal{GL}$ terminates with a certain result if and only if the execution of its translation into SA terminates with the same result. This will be established in Corollary 3, as a consequence of soundness and completeness results from the translation $\mathcal{T}$ (Theorem 2 and Theorem 1)

To prove the correctness of the translation function $\mathcal{T}$, we separate each of the transition relations $\rightsquigarrow_p$ defined in the previous chapter, into two transition relations that will take into account whether computation continues inside the block or jumps into another block.

In this chapter we will need the auxiliary function that follows. For $p \in \mathbf{P}_\bullet$, b and $b_1 \in \mathbf{B}$, $x \in \mathsf{vars}(p)$ and $a \in \mathbf{Aexp}$, the notation $b = \overrightarrow{(x := a)}; b_1$ will mean that block b comprises a sequence of assignments $\overrightarrow{(x := a)}$, followed by block b_1. Also, for $\mathcal{V}$ a version function, $\mathcal{V}\,[\overrightarrow{(x := a)}]$ will mean a new version function $\mathcal{V}_1$ such that:

$$\begin{aligned} \mathcal{V}_1([\,]) &= \mathcal{V} \\ \mathcal{V}_1(x := a; \overrightarrow{(y := a_1)}) &= \mathcal{V}'[x \mapsto \mathcal{V}'(x) + 1] \\ \textit{where} & \\ \mathcal{V}' &= \mathcal{V}_1[\overrightarrow{y := a_1}] \end{aligned}$$

This corresponds to the evolution of a version function in the process of translating a sequence of $\mathcal{GL}$-assignments in SA form.

Throughout this chapter we will assume that any $p \in \mathbf{P}_\bullet$ and $p' \in \mathbf{P}_\bullet^{\mathbf{SA}}$ is well-formed.

4.1 SPLITTING THE TRANSITION RELATION OF THE SOURCE LANGUAGE $\mathcal{GL}$

4.1.1 *Computation inside block*

The new transition relation in $\mathcal{GL}$ inside blocks is given through a binary relation on configurations. The configurations are now triples $\langle b, s, \mathcal{V} \rangle$ where b is a block, s is a state and $\mathcal{V}$ is a function that keeps the version of each variable. This will be important in the proofs latter in this chapter,

since it allows the connection between execution of a program and execution of its translation into SA form. A *terminal configuration inside a block* has the form $\langle d, s, \mathcal{V} \rangle$ for $d = \texttt{return}\, a$, $d = \texttt{goto}\, l$ or $d = \texttt{branch}\ c\ l\ l'$, *i.e.*, a terminal configuration is a configuration that represents jump or return instructions inside a block.
As this relation represents computation steps inside a block, the only rule that defines the relation has to do with assignments, as presented in the next definition.

Definition 13 (Transition relation for $\mathcal{GL}$ inside blocks) *The transition relation inside blocks in $\mathcal{GL}$ is denoted by $\rightarrow$ and is the binary relation on configurations given by the following rule:*

$$\langle x := a\,; b, s, \mathcal{V} \rangle \ \rightarrow \langle b, s[x \mapsto [\![a]\!](s)], \mathcal{V}\,[x \mapsto \mathcal{V}\,(x) + 1] \rangle$$

We represent by $\rightarrow^*$ the reflexive and transitive closure of $\rightarrow$, that allows to put together a sequence of zero or more transitions. By $\rightarrow^n$ we represent n transitions in the relation $\rightarrow$.

Lemma 3 *Let $p \in \mathbf{P}_\bullet$. Let b_1, b_2 be two blocks. Let $s_1,\ s_2 \in \Sigma$. Let $\mathcal{V}_1, \mathcal{V}_2 \in \mathbf{VS}$. We have, for any $n \in \mathbb{N}_0$:*

$$\langle b_1, s_1, \mathcal{V}_1 \rangle \ \rightarrow^n \langle b_2, s_2, \mathcal{V}_2 \rangle \ \Longrightarrow \ \langle b_1, s_1 \rangle \rightsquigarrow_p^n \langle b_2, s_2 \rangle$$

Proof. Easy induction on n having in mind Definition 13 and Definition 5 (case atrib). □

4.1.2 *Computation across blocks*

We define now the second transition relation to deal with the *jump instructions*. Here, the configurations are $\langle c, s, \mathcal{V} \rangle$ for $c := l$ or $c = \texttt{return}\, a$. Next, we present the definition of this relation in $\mathcal{GL}$. Before this, we will need an auxiliary function, vrs(l,p), that will produce the correct version of each variable to be used in the block labeled l in program p, which we define next.

Definition 14 (Synchronized versions) *Let $p \in \mathbf{P}_\bullet$ and $(l : b) \in p$.*

$$\mathsf{vrs}(l, p) = [x \mapsto i \mid x_i \in \mathsf{dom}\,(\phi) \ and \ (l : \phi b') \in \mathcal{T}\,(p)]$$

Definition 15 (Transition relation across blocks in $\mathcal{GL}$) *Let $p \in \mathbf{P}_\bullet$ be a program in $\mathcal{GL}$. The transition relation across blocks induced by p is a binary relation on configurations and is denoted* *by $-\!\!\circ\!\!\rightarrow_p$, or simply $-\!\!\circ\!\!\rightarrow$, when it is clear the program we have in mind. This transition relation is defined by the following rules:*

$$\texttt{return}\qquad \frac{(l : b)\ \in\ p \qquad \langle b, s, \mathcal{V}\,\rangle \rightarrow^{*} \langle \texttt{return}\, a\,, s', \mathcal{V}\,'\rangle}{\langle l, s, \mathcal{V}\,\rangle\ -\!\!\circ\!\!\rightarrow_p \langle \texttt{return}\, a\,, s', \mathcal{V}\,'\rangle}$$

$$\texttt{goto}\qquad \frac{(l : b)\ \in\ p \qquad \langle b, s, \mathcal{V}\,\rangle \rightarrow^{*} \langle \texttt{goto}\, l'\,, s', \mathcal{V}\,'\rangle}{\langle l, s, \mathcal{V}\,\rangle\ -\!\!\circ\!\!\rightarrow_p \langle l', s', \mathsf{vrs}(l', p)\rangle}$$

$$\texttt{branch}_{\texttt{T}}\qquad \frac{(l : b)\ \in\ p \qquad \langle b, s, \mathcal{V}\,\rangle \rightarrow^{*} \langle \texttt{branch}\ c\ \ l'\ l''\,, s', \mathcal{V}\,'\rangle \qquad [\![c\,]\!](s') = \texttt{T}}{\langle l, s, \mathcal{V}\,\rangle\ -\!\!\circ\!\!\rightarrow_p \langle l', s', \mathsf{vrs}(l', p)\rangle}$$

$$\texttt{branch}_{\texttt{F}}\qquad \frac{(l : b)\ \in\ p \qquad \langle b, s, \mathcal{V}\,\rangle \rightarrow^{*} \langle \texttt{branch}\ c\ \ l'\ l''\,, s', \mathcal{V}\,'\rangle \qquad [\![c\,]\!](s') = \texttt{F}}{\langle l, s, \mathcal{V}\,\rangle\ -\!\!\circ\!\!\rightarrow_p \langle l'', s', \mathsf{vrs}(l'', p)\rangle}$$

In a sequence of transitions across blocks, an intermediate configuration has the form $\langle l, s, \mathcal{V}\,\rangle$ and a *terminal configuration across blocks* has the form $\langle \texttt{return}\, a\ , s', \mathcal{V}\,'\rangle$. We represent by $-\!\!\circ\!\!\rightarrow_p^{*}$ the reflexive and transitive closure of $-\!\!\circ\!\!\rightarrow_p$, that allows to put together a sequence of zero or more transitions. By $-\!\!\circ\!\!\rightarrow_p^{n}$ we represent n transitions on $-\!\!\circ\!\!\rightarrow_p$.

Now we establish some basic results relating the transition relations $\rightsquigarrow$ and $-\!\!\circ\!\!\rightarrow$ for $\mathcal{GL}$ programs.

Lemma 4 *Let $p \in \mathbf{P}_\bullet$ and $(l : b_1) \in p$. Let $s_1, s_2 \in \Sigma$. We have one of the following, for some n:*

1. $\langle l, s_1, \mathcal{V}_1\rangle\ -\!\!\circ\!\!\rightarrow \langle \texttt{return}\, a\ , s_2, \mathcal{V}_2\rangle \implies \langle b_1, s_1\rangle \rightsquigarrow^{n} \langle \texttt{return}\, a\ , s_2\rangle$

2. $\langle l, s_1, \mathcal{V}_1\rangle\ -\!\!\circ\!\!\rightarrow \langle l', s_2, \mathcal{V}_2\rangle$ *implies one of the following rules:*

   a) $\langle b_1, s_1\rangle \rightsquigarrow^{n} \langle \texttt{goto}\, l'\ , s_2\rangle$
   or

   b) $\langle b_1, s_1\rangle \rightsquigarrow^{n} \langle \texttt{branch}\ c\ \ l'\ l''\ , s_2\rangle$ *and* $[\![c\,]\!](s_2) = \texttt{T}$
   or

   c) $\langle b_1, s_1\rangle \rightsquigarrow^{n} \langle \texttt{branch}\ c\ \ l''\ l'\ , s_2\rangle$ *and* $[\![c\,]\!](s_2) = \texttt{F}$

Proof. Follows easily from Definition 15 and Lemma 3. □

Lemma 5 *Let $p \in \mathbf{P}_\bullet$ and $(l : b_1), (l' : b_2) \in p$. Let $s_1, s_2 \in \Sigma$. Let $n = \#\overrightarrow{(x := e\,)}$. We have that:*

1. *If* $b_1 = \overrightarrow{(x := e\,)}; \texttt{return}\, a$ *and* $\langle b_1, s_1\rangle \rightsquigarrow^{n} \langle \texttt{return}\, a\ , s_2\rangle$ *then,* $\forall \mathcal{V}_1\ \exists \mathcal{V}_2$*:*

   1.1. $\langle b_1, s_1, \mathcal{V}_1\rangle\ \rightarrow^{n} \langle \texttt{return}\, a\ , s_2, \mathcal{V}_2\rangle$

   1.2. $\langle l, s_1, \mathcal{V}_1\rangle\ -\!\!\circ\!\!\rightarrow \langle \texttt{return}\, a\ , s_2, \mathcal{V}_2\rangle$

2. *If* $b_1 = \overrightarrow{(x := e\,)}; \texttt{goto}\, l'$ *and* $\langle b_1, s_1\rangle \rightsquigarrow^{n} \langle \texttt{goto}\, l'\ , s_2\rangle$ *then,* $\forall \mathcal{V}_1\ \exists \mathcal{V}_2$*:*

2.1. $\langle b_1, s_1, \mathcal{V}_1\rangle \;\rightarrow^n\; \langle \mathtt{goto}\, l', s_2, \mathcal{V}_2\rangle$

2.2. $\langle l, s_1, \mathcal{V}_1\rangle \;\multimap\!\!\rightarrow\; \langle l', s_2, \mathsf{vrs}(l', p)\rangle$

3. *If* $b_1 = \overrightarrow{(x := e)}; \mathtt{branch}\; c\;\; l'\;\; l''$*, and* $\langle b_1, s_1\rangle \rightsquigarrow^n \langle \mathtt{branch}\; c\;\; l'\;\; l'', s_2\rangle$ *then,* $\forall\, \mathcal{V}_1\; \exists\, \mathcal{V}_2$*:*

   3.1. $\langle b_1, s_1, \mathcal{V}_1\rangle \;\rightarrow^n\; \langle \mathtt{branch}\; c\;\; l'\;\; l'', s_2, \mathcal{V}_2\rangle$

   3.2. Depending on the meaning of c*, we have:*

   3.2.1. If $[\![c]\!](s_2) = \mathtt{T}$*,* $\langle l, s_1, \mathcal{V}_1\rangle \;\multimap\!\!\rightarrow\; \langle l', s_2, \mathsf{vrs}(l', p)\rangle$

   3.2.2. If $[\![c]\!](s_2) = \mathtt{F}$*,* $\langle l, s_1, \mathcal{V}_1\rangle \;\multimap\!\!\rightarrow\; \langle l'', s_2, \mathsf{vrs}(l'', p)\rangle$

Proof. The parts 1 are easy induction on n, and parts 2 follow then from the respective parts 1 and Definition 15. □

Definition 16 (Execution of program p in a state s in $\mathcal{GL}$ *w.r.t.* $\multimap\!\!\rightarrow$) *Let p be a program in* $\mathbf{P}_\bullet$*. Let* $s \in \Sigma$*. The* execution of the program p in the state s *w.r.t.* $\multimap\!\!\rightarrow$ *is the sequence of transitions (using the transition relation* $\multimap\!\!\rightarrow$*) generated by* $\langle \bullet, s, \mathcal{V}_1\rangle$*, where* $\mathcal{V}_1$ *corresponds to the initial version function for translation* $\mathcal{T}$*, i.e.,* $\mathcal{V}_1 = \mathsf{inc}\,(\pi_1(\mathsf{initC}\, p))$*. When the execution is finite and has terminal configuration* $\langle \mathtt{return}\, a\;, s_1, \mathcal{V}_2\rangle$*, we say that the result of the execution is* $[\![a]\!](s_1)$

Throughout this chapter, we assume that when we refer to the execution of a $\mathcal{GL}$-program *w.r.t.* a state and we do not specify the relation, we refer to an execution *w.r.t.* $\rightsquigarrow$.

Let us prove that indeed the two semantics for a $\mathcal{GL}$-program are equivalent.

Proposition 4 *Let* $p \in \mathbf{P}_\bullet$ *and* $s_0 \in \Sigma$*.*

1. *If the execution of p in state s_0 w.r.t.* $\rightsquigarrow$ *is finite and has* $\langle \mathtt{return}\, a\;, s\rangle$ *as terminal configuration, then the execution of p in state s_0 w.r.t.* $\multimap\!\!\rightarrow$ *is finite and has terminal configuration* $\langle \mathtt{return}\, a\;, s, \mathcal{V}\rangle$ *for some* $\mathcal{V}$*.*

2. *If the execution of p in state s_0 w.r.t.* $\multimap\!\!\rightarrow$ *is finite and has* $\langle \mathtt{return}\, a\;, s, \mathcal{V}\rangle$ *as terminal configuration, then the execution of p in state s_0 w.r.t.* $\rightsquigarrow$ *is finite and has terminal configuration* $\langle \mathtt{return}\, a\;, s\rangle$*.*

Proof. Let b be the initial block of p so we have $(\bullet : b) \in p$.

1. If the execution of p in state s_0 with respect to the semantics of relation $\rightsquigarrow$ is finite and has $\langle \mathtt{return}\, a\;, s\rangle$ as terminal configuration, exists a finite sequence of jump instructions j_i such that

$$\langle b, s_0\rangle \rightsquigarrow^* \langle j_1, s_1\rangle \rightsquigarrow \langle b_{j_1}, s_1\rangle \rightsquigarrow^* \langle j_2, s_2\rangle \rightsquigarrow \langle b_{j_2}, s_2\rangle \rightsquigarrow \cdots \rightsquigarrow^* \langle \mathtt{return}\, a\;, s\rangle$$

where j_i are the `goto` or `branch` instructions in the execution of program p in state s_0 and each b_{j_i} is the block determined by the jump j_i and the state s_i. So, by Lemma 5, for some $\mathcal{V}$,

$$\langle \bullet, s_0, \pi_1(\mathsf{initC}\ p)\rangle \multimap \langle l_1, s_1, \mathsf{vrs}(l_1, p)\rangle \multimap \langle l_2, s_2, \mathsf{vrs}(l_2, p)\rangle \multimap \cdots \multimap^* \langle \mathtt{return}\, a\ , s, \mathcal{V}\rangle$$

where the l_i are the labels determined by the corresponding jump instruction j_i and state s_i. So, there is a finite execution of p in state s_0 *w.r.t.* the relation $\multimap$.

2. Conversely, if the execution of p in state s_0 *w.r.t.* $\multimap$ is finite and its terminal configuration is $\langle \mathtt{return}\, a\ , s, \mathcal{V}\rangle$, then, using Lemma 4, we can conclude that the execution of p in state s_0 *w.r.t.* $\rightsquigarrow$ has terminal configuration $\langle \mathtt{return}\, a\ , s\rangle$.

□

Corollary 1 *Let $p \in \mathbf{P}_\bullet$ and $s_0 \in \Sigma$. The execution of p in state s_0 w.r.t. to $\rightsquigarrow$ is finite* iff *the execution of p in state s_0 w.r.t. $\multimap$ is finite. Furthermore, when both executions are finite, the results are the same.*

Proof. Follows directly from Proposition 4. □

4.2 SPLITTING THE TRANSITION RELATION IN THE SA LANGUAGE

In the SA language, the relation $\rightsquigarrow$ we defined in Chapter 2 will also be split in the same manner as for $\mathcal{GL}$, giving rise to a relation $\rightarrow$ for computations inside blocks and a relation $\multimap$ for computations across blocks.

4.2.1 *Computation inside blocks*

The new transition relation inside blocks in SA is given through a binary relation on configurations. The configurations are now triples $\langle b', s', \mathcal{V}\rangle$ where b' is a block in SA, s' is a state SA and $\mathcal{V}$ is a function that keeps the version of each variable. The transition relation inside blocks is similar to the transition relation inside blocks for $\mathcal{GL}$-programs, which only has one rule related with assignments.

Definition 17 (Transition relation for SA programs inside blocks) *The transition relation inside blocks in SA programs is denoted by $\rightarrow$ and is the binary relation on configurations given by the following rule:*

$$\langle x_i := a'; b', s', \mathcal{V}\rangle \rightarrow \langle b', s'[x_i \mapsto [\![a']\!](s')], \mathcal{V}[x \mapsto i]\rangle$$

Lemma 6 *Let $p' \in \mathbf{P}^{\mathbf{SA}}_{\bullet}$. Let b'_1, b'_2 be two SA blocks and let $s'_1,\ s'_2 \in \Sigma^{\mathbf{SA}}$. Let $\mathcal{V}_1, \mathcal{V}_2 \in \mathbf{VS}$. For any $n \in \mathbb{N}_0$, we have:*

$$\langle b'_1, s'_1, \mathcal{V}_1\rangle \ \rightarrow^n \langle b'_2, s'_2, \mathcal{V}_2\rangle \implies \langle b'_1, s'_1\rangle \rightsquigarrow^n_p \langle b'_2, s'_2\rangle$$

Proof. Easy induction on n having in mind Definition 17 and Definition 10 (case atrib). □

As usual, $\rightarrow^*$ represents the reflexive and transitive closure of $\rightarrow$ and $\rightarrow^n$ represents n transitions between configurations of $\rightarrow$.

4.2.2 *Computation across blocks*

We define the second transition relation to deal with the *jump instructions* for SA programs. The configurations are triples $\langle c', s', \mathcal{V}\rangle$ where $c' = l$ or $c' = \texttt{return}\, a'$, s' is a SA state. We will need the function upd defined in Definition 10 and the following notion of synchronized versions applied to SA programs.

Definition 18 (Synchronized versions for SA programs) *Let $p' \in \mathbf{P}^{\mathbf{SA}}_{\bullet}$ and let l be a label of p'.*

$$\mathsf{vrs}(l, p') = [x \mapsto i \mid x_i \in \mathsf{dom}\,(\phi) \text{ and } (l : \phi b') \in p']$$

Definition 19 (Transition relation across blocks in SA) *Let $p' \in \mathbf{P}^{\mathbf{SA}}_{\bullet}$. The transition relation induced by p' is a binary relation on configurations and is denoted by $\multimap\!\!\rightarrow_{p'}$ or simply $\multimap\!\!\rightarrow$ when it is clear the program we have in mind. This transition relation is defined by the following rules:*

return
$$\frac{(l : \phi b') \in p' \qquad \langle b', s', \mathcal{V}\rangle \ \rightarrow^* \langle \texttt{return}\, a', s'', \mathcal{V}'\rangle}{\langle l, s', \mathcal{V}\rangle \ \multimap\!\!\rightarrow_{p'} \langle \texttt{return}\, a', s'', \mathcal{V}'\rangle}$$

goto
$$\frac{(l' : \phi'' b''),\ (l : \phi b') \in p' \qquad \langle b', s', \mathcal{V}\rangle \ \rightarrow^* \langle \texttt{goto}\, l'_i, s'', \mathcal{V}'\rangle}{\langle l, s', \mathcal{V}\rangle \ \multimap\!\!\rightarrow_{p'} \langle l', \mathsf{upd}\,(\phi'', i, s''), \mathsf{vrs}(l', p)\rangle}$$

$\mathsf{branch}_{\mathtt{T}}$
$$\frac{(l' : \phi'' b''),\ (l'' : \phi''' b'''),\ (l : \phi b') \in p' \ \langle b', s', \mathcal{V}\rangle \ \rightarrow^* \langle \texttt{branch}\ c'\ l'_i\ l''_j, s'', \mathcal{V}'\rangle \ [\![c']\!](s'') = \mathtt{T}}{\langle l, s', \mathcal{V}\rangle \ \multimap\!\!\rightarrow_{p'} \langle l', \mathsf{upd}\,(\phi'', i, s''), \mathsf{vrs}(l', p')\rangle}$$

$\mathsf{branch}_{\mathtt{F}}$
$$\frac{(l' : \phi'' b''),\ (l'' : \phi''' b'''),\ (l : \phi b') \in p' \ \langle b', s', \mathcal{V}\rangle \ \rightarrow^* \langle \texttt{branch}\ c'\ l'_i\ l''_j, s'', \mathcal{V}'\rangle \ [\![c']\!](s'') = \mathtt{F}}{\langle l, s', \mathcal{V}\rangle \ \multimap\!\!\rightarrow_{p'} \langle l'', \mathsf{upd}\,(\phi''', j, s''), \mathsf{vrs}(l'', p')\rangle}$$

It is important to recall that when the execution of a program consists of a jump into a block, an update of the state is required according to the correspondent ϕ-function.

Definition 20 (Execution of program p' in a state s' in SA *w.r.t.* $\multimap\!\!\rightarrow$**)** *Let $p \in \mathbf{P}_\bullet$ and $p' = \mathcal{T}(p)$. Let $s' \in \Sigma^{\mathbf{SA}}$. The* execution of the program p' in the state s' *w.r.t.* $\multimap\!\!\rightarrow$ *is the sequence of transitions (using the transition relation $\multimap\!\!\rightarrow$) generated by $\langle \bullet, s', \mathcal{V}_1 \rangle$, with $\mathcal{V}_1 = \mathsf{inc}\,(\pi_1(\mathsf{initC}\,p))$*

Firstly, we will see two lemmas relating the relations $\rightsquigarrow$ and $\multimap\!\!\rightarrow$ for SA programs.

Lemma 7 *Let $p' \in \mathbf{P}^{\mathbf{SA}}_\bullet$, $(l : \phi b'_1)$, $(l' : \phi' b')$ and $(l'' : \phi'' b'') \in p'$. Let $s'_1, s'_2 \in \Sigma^{\mathbf{SA}}$. We have that, for some $n \in \mathbb{N}_0$:*

1. $\langle l, s'_1, \mathcal{V}_1 \rangle \;\multimap\!\!\rightarrow\; \langle \mathtt{return}\, a', s'_2, \mathcal{V}_2 \rangle \implies \langle b'_1, s'_1 \rangle \rightsquigarrow^n \langle \mathtt{return}\, a', s'_2 \rangle$
2. $\langle l, s'_1, \mathcal{V}_1 \rangle \;\multimap\!\!\rightarrow\; \langle l', s'_2, \mathcal{V}_2 \rangle$ *implies one of the following:*
   a) $\langle b'_1, s'_1 \rangle \rightsquigarrow^n \langle \mathtt{goto}\, l'_i, s' \rangle$ *where* $s'_2 = \mathsf{upd}\,(\phi', i, s')$
   or
   b) $\langle b'_1, s'_1 \rangle \rightsquigarrow^n \langle \mathtt{branch}\; c'\; l'_i\; l''_j, s' \rangle$ *and* $[\![c']\!](s'_2) = \mathtt{T}$ *where* $s'_2 = \mathsf{upd}\,(\phi', i, s')$
   or
   c) $\langle b'_1, s'_1 \rangle \rightsquigarrow^n \langle \mathtt{branch}\; c'\; l''_i\; l'_j, s' \rangle$ *and* $[\![c']\!](s'_2) = \mathtt{F}$ *where* $s'_2 = \mathsf{upd}\,(\phi', j, s')$

Proof. Follows easily from Definition 19 and Lemma 6. □

Lemma 8 *Let $p' \in \mathbf{P}^{\mathbf{SA}}_\bullet$, $(l : \phi b'_1)$, $(l' : \phi' b'_2)$ and $(l'' : \phi'' b'_3) \in p'$. Let $s'_1, s'_2 \in \Sigma^{\mathbf{SA}}$. Let $n = \#\overrightarrow{(x := e')}$. We have that:*

1. *If* $b'_1 = \overrightarrow{(x := e')}; \mathtt{return}\, a'$ *and* $\langle b'_1, s'_1 \rangle \rightsquigarrow^n \langle \mathtt{return}\, a', s'_2 \rangle$ *then,* $\forall \mathcal{V}_1\, \exists \mathcal{V}_2$*:*
   1.1. $\langle b'_1, s'_1, \mathcal{V}_1 \rangle \;\rightarrow^n\; \langle \mathtt{return}\, a', s'_2, \mathcal{V}_2 \rangle$
   1.2. $\langle l, s'_1, \mathcal{V}_1 \rangle \;\multimap\!\!\rightarrow\; \langle \mathtt{return}\, a', s'_2, \mathcal{V}_2 \rangle$
2. *If* $b'_1 = \overrightarrow{(x := e)}; \mathtt{goto}\, l'_i$ *and* $\langle b'_1, s'_1 \rangle \rightsquigarrow^n \langle \mathtt{goto}\, l'_i, s'_2 \rangle$ *then,* $\forall \mathcal{V}_1\, \exists \mathcal{V}_2$*:*
   2.1. $\langle b'_1, s'_1, \mathcal{V}_1 \rangle \;\rightarrow^n\; \langle \mathtt{goto}\, l'_i, s'_2, \mathcal{V}_2 \rangle$
   2.2. $\langle l, s'_1, \mathcal{V}_1 \rangle \;\multimap\!\!\rightarrow\; \langle l', \mathsf{upd}\,(\phi', i, s'_2), \mathsf{vrs}(l', p') \rangle$
3. *If* $b'_1 = \overrightarrow{(x := e')}; \mathtt{branch}\; c'\; l'_i\; l''_j$, *and* $\langle b'_1, s'_1 \rangle \rightsquigarrow^n \langle \mathtt{branch}\; c'\; l'_i\; l''_j, s'_2 \rangle$ *then,* $\forall \mathcal{V}_1\, \exists \mathcal{V}_2$*:*
   3.1. $\langle b'_1, s'_1, \mathcal{V}_1 \rangle \;\rightarrow^n\; \langle \mathtt{branch}\; c'\; l'_i\; l''_j, s'_2, \mathcal{V}_2 \rangle$
   3.2. *Depending on the meaning of c, we have:*
      3.2.1. *If* $[\![c']\!](s'_2) = \mathtt{T}$, $\langle l, s'_1, \mathcal{V}_1 \rangle \;\multimap\!\!\rightarrow\; \langle l', \mathsf{upd}\,(\phi', i, s'_2), \mathsf{vrs}(l', p') \rangle$
      3.2.2. *If* $[\![c']\!](s'_2) = \mathtt{F}$, $\langle l, s'_1, \mathcal{V}_1 \rangle \;\multimap\!\!\rightarrow\; \langle l'', \mathsf{upd}\,(\phi'', j, s'_2), \mathsf{vrs}(l'', p') \rangle$

Proof. The parts 1 are easy inductions on n, and parts 2 follow then from the respective parts 1 and Definition 19. $\square$

Now, we can prove that both semantics for $\mathcal{GL}$ are equivalent.

Proposition 5 *Let $p \in \mathbf{P}_\bullet$ and $p' = \mathcal{T}(p)$. Let $s'_0 \in \Sigma^{\mathbf{SA}}$.*

1. *If the execution of p' in state s'_0 w.r.t. $\rightsquigarrow$ is finite and has $\langle \mathtt{return}\, a', s' \rangle$ as terminal configuration, then the execution of p' in state s'_0 w.r.t. $\multimap\!\!\rightarrow$ is finite and has a terminal configuration of the form $\langle \mathtt{return}\, a', s', \mathcal{V} \rangle$, for some $\mathcal{V}$.*

2. *If the execution of p' in state s'_0 w.r.t. $\multimap\!\!\rightarrow$ is finite and has terminal configuration $\langle \mathtt{return}\, a', s', \mathcal{V} \rangle$, then the execution of p' in state s'_0 w.r.t. $\rightsquigarrow$ is finite and has $\langle \mathtt{return}\, a', s' \rangle$ as terminal configuration.*

Proof. Let b' be the initial block of p'. So, we have $(\bullet : b') \in p'$,

1. If the execution of p' in state s'_0 *w.r.t.* $\rightsquigarrow_{p'}$ is finite and has $\langle \mathtt{return}\, a', s' \rangle$ as terminal configuration, exists a finite sequence of jump instructions j'_i such that

$$\langle b', s'_0 \rangle \rightsquigarrow^* \langle j'_1, s'_1 \rangle \rightsquigarrow \langle b'_{j'_1}, \mathsf{upd}\,(\phi_{j'_1} i_{j'_1}, s'_1) \rangle \rightsquigarrow^* \cdots \rightsquigarrow \langle \mathtt{return}\, a', s' \rangle$$

   where the j'_k are the `goto` or `branch` instructions in the execution, $b'_{j'_k}, \phi_{j'_k}, i_{j'_k}$ are, respectively, the block, the synchronization preamble and the door determined by each jump instruction j'_k and state s'_k. So, by Lemma 8,

$$\langle \bullet, s'_0, \pi_1(\mathsf{initC}\, p') \rangle \multimap\!\!\rightarrow \langle l_1, \mathsf{upd}\,(\phi_{j'_1}, i_{j'_1}, s'_1), \mathsf{vrs}(l_1, p') \rangle \multimap\!\!\rightarrow^* \cdots \multimap\!\!\rightarrow \langle \mathtt{return}\, a', s', \mathcal{V} \rangle$$

   where l_k are the labels determined by the corresponding jump instruction j'_k and state s'_k. So, there is a finite execution of p' in state s'_0 *w.r.t.* the relation $\multimap\!\!\rightarrow$.

2. If the execution of p' in state s' with respect to $\multimap\!\!\rightarrow$ is finite and its terminal configuration is $\langle \mathtt{return}\, a', s'_2, \mathcal{V}_2 \rangle$ then by a converse reasoning to 1. and Lemma 7, we can conclude that the execution of p' in state s' with respect to $\rightsquigarrow$ has terminal configuration $\langle \mathtt{return}\, a', s'_2 \rangle$.

$\square$

Corollary 2 *Let $p \in \mathbf{P}_\bullet$ and $p' = \mathcal{T}(p)$. Let $s'_0 \in \Sigma^{\mathbf{SA}}$. The execution of p' in state s'_0 w.r.t. to $\rightsquigarrow$ is finite* iff *the execution of p' in state s'_0 w.r.t. $\multimap\!\!\rightarrow$ is finite. Furthermore, when both executions are finite, the results are the same.*

Proof. Follows directly from Proposition 5. $\square$

4.3 PROPERTIES OF THE TRANSLATION INTO SA FORM

In this section we will establish the main properties of our translation $\mathcal{T}$ from $\mathcal{GL}$-programs into SA-programs. We will show soundness and completeness of translation $\mathcal{T}$ relatively to the semantic relations $\multimap\!\!\rightarrow$ of the previous two subsections, *i.e.*, we will show that a sequence of n steps of the translated program is mapped to a sequence of n steps in the source program (Theorem 1 and vice-versa (Theorem 2).

With the help of the soundness and completeness results for $\mathcal{T}$, and the results linking the original semantics for $\mathcal{GL}$ and SA, given by relations $\rightsquigarrow$, with the new semantics for $\mathcal{GL}$ and SA in previous subsection, given by relation $\multimap\!\!\rightarrow$, we will also prove the correctness of $\mathcal{T}$ (Theorem 1).

In this subsection we will need the following auxiliary definition and the following lemmas.

Definition 21 (Context in which l is translated) *Let $p \in \mathbf{P}_\bullet$ and $l \in \mathsf{labels}(p)$. The context determined by the translation of p till label l is given by:*
$\mathsf{ctx}(p,l)=\pi_2(\mathsf{T_L}\,(p_0, \mathsf{initC}\,p))$, *for the unique p_0 such that $p = (p_0,\, l : b\,, p_1)$, for some b and p_1.*

As a convention, we say that $\mathsf{ctx}(p,\bullet) = \pi_1(\mathsf{initC}\,p)$.

Lemma 9 *For e an expression of* **Aexp** *or* **Cexp**, *$s' \in \Sigma^{\mathbf{SA}}$, $s_1 \in \Sigma$ and $\mathcal{V} \in \mathbf{VS}$, we have that:* $[\![e]\!](s_1) = [\![\widehat{\mathcal{V}}(e)]\!](s' \oplus \widehat{\mathcal{V}}(s_1))$.

Proof. The proof is made using induction on the structure of expressions. $\mathsf{varsE}(e)$ is either an empty set, a single set or a set of many variables because e can be either a number (which the number of variables is 0), a single variable (that just have one variable) or a binary expression.

- Case $e = n$ and $n \in \mathbb{N}_0$, $\mathsf{varsE}(e)$ is an empty set, immediately we have:
  $[\![n]\!](s_1) = n$
  $[\![\widehat{\mathcal{V}}(n)]\!](s' \oplus \widehat{\mathcal{V}}(s_1)) = n$
- Case $e = x$, and $x \in \mathsf{vars}(p)$, $\mathsf{varsE}(e)$ is a single set. We have that:
  $[\![x]\!](s_1) = s_1(x)$
  $[\![\widehat{\mathcal{V}}(x)]\!](s' \oplus \widehat{\mathcal{V}}(s_1)) = [\![x_{\mathcal{V}(x)}]\!](s' \oplus \widehat{\mathcal{V}}(s_1)) = s_1(x)$. Note that $x_{\mathcal{V}(x)} \in \mathsf{dom}\,(\widehat{\mathcal{V}}(s_1))$ and so $\widehat{\mathcal{V}}(s_1)(x_{\mathcal{V}(x)}) = s_1(x)$. See Page 21.
- Case $e = e_1 \;\mathsf{op}\; e_2$, e_1 and e_2 expressions
  Let assume that this lemma is valid for e_1 and e_2. We want to prove that the lemma is also valid for $e_1 \;\mathsf{op}\; e_2$. We have that:
  - $[\![e_1]\!](s_1) = [\![\widehat{\mathcal{V}}(e_1)]\!](s' \oplus \widehat{\mathcal{V}}(s_1))$
  - $[\![e_2]\!](s_1) = [\![\widehat{\mathcal{V}}(e_2)]\!](s' \oplus \widehat{\mathcal{V}}(s_1))$

We want to prove that: $[\![\widehat{\mathcal{V}}(e_1 \mathsf{\ op\ } e_2)]\!](s' \oplus \widehat{\mathcal{V}}(s_1)) = [\![e_1]\!](s_1) \mathsf{\ op\ } [\![e_2]\!](s_1)$
So,
$[\![\widehat{\mathcal{V}}(e_1 \mathsf{\ op\ } e_2)]\!](s' \oplus \widehat{\mathcal{V}}(s_1)) = [\![\widehat{\mathcal{V}}(e_1) \mathsf{\ op\ } \widehat{\mathcal{V}}(e_2))]\!](s' \oplus \widehat{\mathcal{V}}(s_1)) =$
$[\![\widehat{\mathcal{V}}(e_1)]\!](s' \oplus \widehat{\mathcal{V}}(s_1)) \mathsf{\ op\ } [\![\widehat{\mathcal{V}}(e_2)]\!](s' \oplus \widehat{\mathcal{V}}(s_1)) = [\![e_1]\!](s_1) \mathsf{\ op\ } [\![e_2]\!](s_1)$

□

4.3.1 *Soundness*

We start with some auxiliary functions and two properties about them.

Definition 22 (Erase) *The erase function transforms the argument in SA to its counterpart in the source language. In other words, all indexes and ϕ-functions are removed. We have that:*

- $\mathsf{erase}(a' \mathsf{\ op\ } a'') = \mathsf{erase}(a') \mathsf{\ op\ } \mathsf{erase}(a'')$
- $\mathsf{erase}(\mathtt{return}\, a') = \mathtt{return}(\mathsf{erase}(a'))$
- $\mathsf{erase}(\mathtt{goto}\, l_i) = \mathtt{goto}\, l$
- $\mathsf{erase}(\mathtt{branch}\ c'\ l'_i\ l''_j) =$ $\mathtt{branch}\ (\mathsf{erase}(c'))\ l'\ l''$
- $\mathsf{erase}(x_i := a'; b') =$ $x := \mathsf{erase}(a'); \mathsf{erase}(b')$
- $\mathsf{erase}(\phi b') = \mathsf{erase}(b')$
- $\mathsf{erase}(x_i) = x$
- $\mathsf{erase}(\bullet) = \bullet$
- $\mathsf{erase}(l) = l$

Definition 23 (State restricted to a version function) *Let $s' \in \Sigma^{\mathbf{SA}}$ and $\mathcal{V} \in \mathbf{VS}$. The state s' restrict to the version function $\mathcal{V}$ is represented by $s'|_{\mathcal{V}}$ and is the function thats maps each variable to the correspondent value according to its version in $\mathcal{V}$, that is $s'|_{\mathcal{V}}(x) = s'(x_{\mathcal{V}(x)})$, for $x \in \mathbf{Var}$.*

Lemma 10 *Let $p \in \mathbf{P}_\bullet$ and $p' = \mathcal{T}(p)$. Let $(l : b) \in p$ and $(l : \phi b')$ its counterpart in p'. Let $(\mathcal{V}, \mathcal{L}) = \mathsf{ctx}(p, l)$ and let $\mathcal{V}'$ and $\mathcal{L}'$ be such that $\mathsf{T_B}(b, l)(\mathcal{V}, \mathcal{L}) = (b', (\mathcal{V}', \mathcal{L}'))$. We have that:* $\mathsf{erase}(b') = b$

Proof. The proof is by induction on the structure of the block b. □

Lemma 11 *Let $a \in \mathbf{Aexp}$. Let $s' \in \Sigma^{\mathbf{SA}}$ and $\mathcal{V} \in \mathbf{VS}$. We have that:*
$[\![a]\!](s'|_{\mathcal{V}}) = [\![\widehat{\mathcal{V}}(a)]\!](s')$

Proof. The proof is made using induction on the structure of expressions.

- Case $a = n$ and $n \in \mathbb{N}$,
  $[\![n]\!](s'|_{\mathcal{V}}) = n$
  $[\![\widehat{\mathcal{V}}(n)]\!](s') = n$

- Case $a = x$ and x is a variable,
  $[\![x]\!](s'|_{\mathcal{V}}) = s'|_{\mathcal{V}}(x) = s'(x_{\mathcal{V}(x)})$
  $[\![\widehat{\mathcal{V}}(x)]\!](s') = s'(x_{\mathcal{V}(x)})$

- Case $a = a_1\ op\ a_2$,
  Let's assume that the lemma is valid for a_1 and a_2. We have:
  1. $[\![a_1]\!](s'|_{\mathcal{V}}) = [\![\widehat{\mathcal{V}}(a_1)]\!](s')$
  2. $[\![a_2]\!](s'|_{\mathcal{V}}) = [\![\widehat{\mathcal{V}}(a_2)]\!](s')$

  We want to prove that: $[\![a_1\ op\ a_2]\!](s'|_{\mathcal{V}}) = [\![\widehat{\mathcal{V}}(a_1\ op\ a_2)]\!](s')$
  So,
  $[\![a_1\ op\ a_2]\!](s'|_{\mathcal{V}}) = s'|_{\mathcal{V}}(a_1\ op\ a_2) = s'|_{\mathcal{V}}(a_1)\ op\ s'|_{\mathcal{V}}(s_2) = [\![a_1]\!](\mathcal{V})\ op\ [\![a_2]\!](\mathcal{V}) =$
  $[\![\widehat{\mathcal{V}}(a_1)]\!](s')\ op\ [\![\widehat{\mathcal{V}}(a_2)]\!](s') = [\![\widehat{\mathcal{V}}(a_1\ op\ a_2)]\!](s')$

$\square$

We establish now the basic result to have soundness of the translation relatively to the computations inside blocks.

Proposition 6 *Let $p \in \mathbf{P}_{\bullet}$ and let p' in $\mathbf{P}_{\bullet}^{\mathbf{SA}}$ be such that $p' = \mathcal{T}(p)$. Let $(l : b)$ be a labeled block in p and $(l : \phi b')$ its translation in p'. Let $n \in \mathbb{N}_0$ and $\mathsf{ctx}(p,l)$=$(\mathcal{V},\mathcal{L})$. Let b_1 be a part of b such that $b = \overrightarrow{(y := e)}; b_1$ and b_1 its correspondent translation in p'. Let $\mathcal{V}_1$ be such that $\mathcal{V}_1 = \mathsf{inc}(\mathcal{V})[\overrightarrow{(y := e)}]$. Let $\mathcal{L}_1$ be a label function such that $\mathcal{L}_1 = \mathcal{L}[l \mapsto (\pi_1(\mathcal{L}(l)), \pi_2(\mathcal{L}(l)), \mathsf{inc}(\mathcal{V}))]$. Let f' be a instruction in SA `goto` l, or `return` a' or `branch` $c'\ l'\ l''$ and let $\mathcal{V}', \mathcal{L}', s'_1, s'_2$ such that:*

$\mathsf{T_B}(b_1, l)(\mathcal{V}_1, \mathcal{L}_1) = (b'_1, (\mathcal{V}', \mathcal{L}'))$ *and* $\langle b'_1, s'_1, \mathcal{V}_1\rangle \rightarrow^n_{p'} \langle f', s'_2, \mathcal{V}_2\rangle$
Then: $\langle b_1, s'_1|_{\mathcal{V}_1}, \mathcal{V}_1\rangle \rightarrow^n_p \langle \mathsf{erase}(f'), s'_2|_{\mathcal{V}_2}, \mathcal{V}_2\rangle$

Proof. By induction on n ($\in \mathbb{N}_0$).
Base Case, for n=0: We immediately have $b'_1 = f', s'_1 = s'_2, \mathcal{V}_1 = \mathcal{V}_2$ and:

- $\mathsf{erase}(f') = f$ by Lemma 10;
- $s'_1|_{\mathcal{V}_1} = s'_2|_{\mathcal{V}_2}$ because $s'_1 = s'_2$ and $\mathcal{V}_1 = \mathcal{V}_2$

Inductive, for $n > 0$:
Suppose that $\langle b'_1, s'_1, \mathcal{V}_1\rangle \rightarrow^1 \langle b'_0, s'_0, \mathcal{V}_0\rangle \rightarrow^{n-1} \langle f', s'_2, \mathcal{V}_2\rangle$
Because of $\langle b'_1, s'_1, \mathcal{V}_1\rangle \rightarrow^1 \langle b'_0, s'_0, \mathcal{V}_0\rangle$ and Definition17 we have that:

(a) $b'_1 = (x_i := a'); b'_0$

(b) $s'_0 = s'_1[x_i \mapsto [\![a']\!](s'_1)]$

(c) $\mathcal{V}_0 = \mathcal{V}_1[x \mapsto i]$

We want to prove that: $\langle \mathsf{erase}\,(b'_1)\,,\, s'_1|_{\mathcal{V}_1},\, \mathcal{V}_1\rangle \rightarrow^1 \langle \mathsf{erase}\,(b'_0)\,,\, s'_0|_{\mathcal{V}_0},\, \mathcal{V}_0\rangle$
For which we need (by Definition 13):

1. $\mathcal{V}_0 = \mathcal{V}_1[x \mapsto \mathcal{V}_1(x) + 1]$
2. $s'_0|_{\mathcal{V}_0} = s'_1|_{\mathcal{V}_1}[x \mapsto [\![\mathsf{erase}\,(a')\,]\!](s'_1|_{\mathcal{V}_1})]$

1. We know that $\mathcal{V}_0 = \mathcal{V}_1[x \mapsto i]$ by **(c)** above, so we need $i = \mathcal{V}_1(x) + 1$. By Lemma 10, we have that: $b_1 = (x := \mathsf{erase}\,(a')\,); \mathsf{erase}\,(b'_0)$. Furthermore, form definition of $\mathsf{T_B}$, we have that

- $i = \mathcal{V}(x) + 1$
- $a' = \widehat{\mathcal{V}}_1(\mathsf{erase}\,(a')\,)$ (*)

2. From **(b)**, **(c)** and Definition 23 it is enough to prove that

$$s'_1[x_i \mapsto [\![a']\!](s'_1)]|_{\mathcal{V}_1[x \mapsto \mathcal{V}_1(x)+1]} = s'_1|_{\mathcal{V}_1}[x \mapsto [\![\mathsf{erase}\,(a')\,]\!](s'_1|_{\mathcal{V}_1})]$$

Let $y \in \mathsf{vars}(p)$,

- Case $y \neq x$,
  - $s'_1[x_i \mapsto [\![a']\!](s'_1)]|_{\mathcal{V}_1[x \mapsto \mathcal{V}_1(x)+1]}(y) = s'_1(y_{\mathcal{V}_1(y)})$
  - $s'_1|_{\mathcal{V}_1}[x \mapsto [\![\mathsf{erase}\,(a')\,]\!](s'_1|_{\mathcal{V}_1})](y) = s'_1(y_{\mathcal{V}_1(y)})$
- Case $y = x$,
  - $s'_1[x_i \mapsto [\![a']\!](s'_1)]|_{\mathcal{V}_1[x \mapsto \mathcal{V}_1(x)+1]}(y) = [\![a']\!](s'_1)$
  - $s'_1|_{\mathcal{V}_1}[x \mapsto [\![\mathsf{erase}\,(a')\,]\!](s'_1|_{\mathcal{V}_1})](y) = [\![\mathsf{erase}\,(a')\,]\!](s'_1|_{\mathcal{V}_1})$

By (*), $[\![a']\!](s'_1) = [\![\widehat{\mathcal{V}}_1(\mathsf{erase}\,(a')\,)]\!](s'_1)$ which is equal to $[\![\mathsf{erase}\,(a')\,]\!](s'_1|_{\mathcal{V}_1})$ by Lemma 11.

□

The previous result allows us to obtain soundness *w.r.t.* one step computations across blocks.

Proposition 7 *Let* $p \in \mathbf{P}_\bullet$ *and let* p' *in* $\mathbf{P}^{\mathbf{SA}}_\bullet$ *be such that* $p' = \mathcal{T}(p)$*. Let* $l \in \mathsf{labels}(p)$*,* $\mathcal{V}_1 \in \mathbf{VS}$ *and let* $\mathcal{L}_1 = \mathcal{L}[l \mapsto (\pi_1(\mathcal{L}(l)), \pi_2(\mathcal{L}(l)), \mathsf{inc}\,(\mathcal{V}))]$ *such that* $\mathsf{ctx}(p, l) = (\mathcal{V}_1, \mathcal{L}_1)$*. Let* $\mathcal{V}_0 = \mathsf{inc}\,(\mathcal{V}_1)$*. Let* f' *be a instruction in SA* $\mathtt{goto}\, l$*, or* $\mathtt{return}\, a'$ *or* $\mathtt{branch}\; c'\; l'\; l''$ *and let* $s'_0, s'_2, \mathcal{V}_2$ *such that:*

$$\langle l, s'_0, \mathcal{V}_0\rangle \multimap\!\!\rightarrow_{p'} \langle f', s'_2, \mathcal{V}_2\rangle \implies \langle l, s'_0|_{\mathcal{V}_0}, \mathcal{V}_0\rangle \multimap\!\!\rightarrow_{p'} \langle \mathsf{erase}\,(f')\,, s'_2|_{\mathcal{V}_2}, \mathcal{V}_2\rangle$$

Proof. Let b be a block labeled by l in p and b' its translation to SA in p'.

- For $f' = \texttt{return}\, a'$, We have that:
  $\langle l, s'_0, \mathcal{V}_0 \rangle \multimap\!\!\to_{p'} \langle \texttt{return}\, a', s'_2, \mathcal{V}_2 \rangle$
  From Definition 15 follows:
  $\langle b', s'_0, \mathcal{V}_0 \rangle \to^n_{p'} \langle \texttt{return}\, a', s'_2, \mathcal{V}_2 \rangle$
  By Proposition 6 we have $\langle b, s'_0|_{\mathcal{V}_0}, \mathcal{V}_0 \rangle \to^n_p \langle \mathsf{erase}\,(\texttt{return}\, a'), s'_2|_{\mathcal{V}_2}, \mathcal{V}_2 \rangle$
  Again, using Definition 15 follows: $\langle l, s'_0|_{\mathcal{V}_0}, \mathcal{V}_0 \rangle \multimap\!\!\to_{p'} \langle \mathsf{erase}\,(\texttt{return}\, a'), s'_2|_{\mathcal{V}_2}, \mathcal{V}_2 \rangle$

- For $f' = l$, is analogous.

□

With the help of the previous proposition, we achieve now the soundness of the translation $\mathcal{T}$ for the relations that speak about computations across blocks.

Theorem 1 (Soundness) *Let $p \in \mathbf{P}_\bullet$ and let p' be $\mathcal{T}(p)$. Let $\mathcal{V}_1 = \mathsf{inc}\,(\pi_1(\mathsf{initC}\, p))$. For any $s'_1, s'_2, \mathcal{V}_2$ and f',*
$\langle \bullet, s'_1, \mathcal{V}_1 \rangle \multimap\!\!\to^n_{p'} \langle f', s'_2, \mathcal{V}_2 \rangle \implies \langle \bullet, s'_1|_{\mathcal{V}_1}, \mathcal{V}_1 \rangle \multimap\!\!\to^n_p \langle \mathsf{erase}\,(f'), s'_2|_{\mathcal{V}_2}, \mathcal{V}_2 \rangle$

Proof. By induction on $n \in \mathbb{N}_0$.

Case $n = 0$.
We have that: $\langle \bullet, s'_1, \mathcal{V}_1 \rangle \multimap\!\!\to^0_{p'} \langle f', s'_2, \mathcal{V}_2 \rangle$
And by definition of $\multimap\!\!\to$ we know that:

- $s'_1 = s'_2$
- $\mathcal{V}_2 = \mathcal{V}_1$
- $f' = \bullet$

We want to prove that: $\langle \bullet, s'_1 \mathcal{V}_1|, \mathcal{V}_1 \rangle \multimap\!\!\to^0_p \langle \mathsf{erase}\,(f'), s'_2|_{\mathcal{V}_2}, \mathcal{V}_2 \rangle$, which is immediate since:

- $\mathsf{erase}\,(f) = \mathsf{erase}\,(\bullet) = \bullet$;
- $s'_1|_{\mathcal{V}_1} = s'_2|_{\mathcal{V}_2}$ because $s'_2 = s'_1$ and $\mathcal{V}_2 = \mathcal{V}_1$.

Case $n \geq 1$
We have that: $\langle \bullet, s'_1, \mathcal{V}_1 \rangle \multimap\!\!\to^{n-1}_{p'} \langle l, s'_0, \mathcal{V}_0 \rangle) \multimap\!\!\to_{p'} \langle f', s'_2, \mathcal{V}_2 \rangle$
By IH we have $\langle \bullet, s'_1|_{\mathcal{V}_1}, \mathcal{V}_1 \rangle \multimap\!\!\to^{n-1}_p \langle l, s'_0 \mathcal{V}_0|, \mathcal{V}_0 \rangle$.
We now will prove that $\langle l, s'_0|_{\mathcal{V}_0}, \mathcal{V}_0 \rangle \multimap\!\!\to_p \langle \mathsf{erase}\,(f'), s'_2|_{\mathcal{V}_2}, \mathcal{V}_2 \rangle$. We also know that: $\langle l, s'_0, \mathcal{V}_0 \rangle \multimap\!\!\to_{p'} \langle f, s'_2, \mathcal{V}_2 \rangle$ and $f = \texttt{return}\, a'$ or $f = l'$. So by applying Proposition 7, we have that $\langle l, s'_0|_{\mathcal{V}_0}, \mathcal{V}_0 \rangle \multimap\!\!\to_p \langle f', s'_2|_{\mathcal{V}_2}, \mathcal{V}_2 \rangle$ with $f' = \mathsf{erase}\,(\texttt{return}\, a')$ or $f' = l'$. □

4.3.2 *Completeness*

In this subsection we will present some results culminating with the presentation of Theorem 2, which establishes the completeness of the translation $\mathcal{T}$ into SA form. We will prove in Proposition 8 that if there is a sequence of n steps from anywhere in a block till the end of that block, then there is an execution from the same point in the translated block till the end of that block.

Proposition 8 *Let $p \in \mathbf{P}_\bullet$ and let p' in $\mathbf{P}_\bullet^{\mathbf{SA}}$ be such that $p' = \mathcal{T}(p)$. Let $(l : b)$ be a labeled block in p and $(l : \phi b')$ its translation in p'. Let $n \in \mathbb{N}_0$ and* $\mathsf{ctx}(p,l)$=*$(\mathcal{V},\mathcal{L})$. Let b_1 be a part of b such that $b = \overrightarrow{(y := e)};b_1$. Let $\mathcal{V}_1$ be such that $\mathcal{V}_1 = \mathsf{inc}\,(\mathcal{V})[\overrightarrow{(y := e)}]$. Let $\mathcal{L}_1$ be a label function such that $\mathcal{L}_1 = \mathcal{L}\,[l \mapsto (\pi_1(\mathcal{L}\,(l)), \pi_2(\mathcal{L}\,(l)), \mathsf{inc}\,(\mathcal{V}))]$. Let f be a instruction* $\mathtt{goto}\,l$*, or* $\mathtt{return}\,a$ *or* $\mathtt{branch}\ c\ \ l'\ l''$ *and let $b'_1, \mathcal{V}', \mathcal{L}', s_1, s_2, \mathcal{V}_2$ such that:*

$$\mathsf{T_B}\,(b_1,l)(\mathcal{V}_1,\mathcal{L}_1) = (b'_1,(\mathcal{V}',\mathcal{L}'))\ \textit{and}\ \langle b_1,s_1,\mathcal{V}_1\rangle \ \rightarrow^n \langle f,s_2,\mathcal{V}_2\rangle$$

Then, $\mathcal{V}' = \mathcal{V}_2$ and:

- *If $f = \mathtt{return}\,a$,*
  1. $\mathcal{L}' = \mathcal{L}_1$
  2. $\forall s \in \Sigma^{\mathbf{SA}}, \exists s' \in \Sigma^{\mathbf{SA}} : \langle b'_1, s \oplus \widehat{\mathcal{V}}_1(s_1), \mathcal{V}_1\rangle \ \rightarrow^n \langle \mathtt{return}\,\widehat{\mathcal{V}}_2(a\,), s' \oplus \widehat{\mathcal{V}}_2(s_2), \mathcal{V}_2\rangle$
- *If $f = \mathtt{goto}\,l'$,*
  1. $\mathcal{L}' = \mathcal{L}_1[l' \mapsto (\pi_1\mathcal{L}_1(l') + 1, \pi_2\mathcal{L}_1(l') +\!\!+ [\mathcal{V}'], \pi_3\mathcal{L}_1(l'))]$
  2. $\forall s \in \Sigma^{\mathbf{SA}}, \exists s' \in \Sigma^{\mathbf{SA}} : \langle b'_1, s \oplus \widehat{\mathcal{V}}_1(s_1), \mathcal{V}_1\rangle \ \rightarrow^n \langle \mathtt{goto}\,l'_{\pi_1(\mathcal{L}'(l'))}, s' \oplus \widehat{\mathcal{V}}_2(s_2), \mathcal{V}_2\rangle$
- *If $f = \mathtt{branch}\ c\ \ l'\ l''$,*
  1. *$\mathcal{L}_2 = \mathcal{L}_1[l' \mapsto (n'+1, d' +\!\!+ [\mathcal{V}_1], \mathcal{V}'_0)]$ where $(n',d',\mathcal{V}'_0) = \mathcal{L}_1(l')$ and $\mathcal{L}' = \mathcal{L}_2[l'' \mapsto (n''+1, d'' +\!\!+ [\mathcal{V}_1], \mathcal{V}''_0)]$ where $(n'',d'',\mathcal{V}''_0) = \mathcal{L}_2(l'')$*
  2. $\forall s \in \Sigma^{\mathbf{SA}}, \exists s' \in \Sigma^{\mathbf{SA}}$:
     $\langle b'_1, s \oplus \widehat{\mathcal{V}}_1(s_1), \mathcal{V}_1\rangle \ \rightarrow^n \langle \mathtt{branch}\ \widehat{\mathcal{V}}_2(c\,)\ l'_{\pi_1(\mathcal{L}'(l'))}\ l''_{\pi_1(\mathcal{L}'(l''))}, s' \oplus \widehat{\mathcal{V}}_2(s_2), \mathcal{V}_2\rangle$

Proof. By induction on n $(\in \mathbb{N}_0)$.
Base Case, for n=0: We immediately have $b_1 = f, s_1 = s_2, \mathcal{V}_1 = \mathcal{V}_2$ and by inspection of $\mathsf{T_B}$ we see that $\mathcal{V}' = \mathcal{V}_1 = \mathcal{V}_2$.

- If $f = \mathtt{return}\,a$,
  1. Inspection of $\mathsf{T_B}$ gives immediately $\mathcal{L}' = \mathcal{L}_1$
  2. Also, we have:
     - $b_1 = \mathtt{return}\,a$

- $b'_1 = \texttt{return}\,\widehat{\mathcal{V}}_1(a)$

We want to prove that: $\forall s \exists s' : \langle b'_1, s \oplus \widehat{\mathcal{V}}_1(s_1), \mathcal{V}_1\rangle \rightarrow^0 \langle \texttt{return}\,\widehat{\mathcal{V}}_2(a), s' \oplus \widehat{\mathcal{V}}_2(s_2), \mathcal{V}_2\rangle$
So, it suffices to take $s' = s$, since $\mathcal{V}_1 = \mathcal{V}_2$ and $s_1 = s_2$.

- If $f = \texttt{goto}\,l'$,

1. Inspection of $\mathsf{T_B}$ and the fact that $\mathcal{V}' = \mathcal{V}_1$ gives
$\mathcal{L}' = \mathcal{L}_1[l' \mapsto (\pi_1\mathcal{L}_1(l') + 1, \pi_2\mathcal{L}_1(l') +\!\!+ [\mathcal{V}'], \pi_3\mathcal{L}_1(l'))]$

2. Also, we have:

- $b_1 = \texttt{goto}\,l'$
- $b'_1 = \texttt{goto}\,l'_{\pi_1(\mathcal{L}'(l'))+1}$

We want to prove that $\langle b'_1, s \oplus \widehat{\mathcal{V}}_1(s_1), \mathcal{V}_1\rangle \rightarrow^0 \langle \texttt{goto}\,l'_{\pi_1(\mathcal{L}'(l'))}, s' \oplus \widehat{\mathcal{V}}_2(s_2), \mathcal{V}_2\rangle$
So, it suffices to take $s' = s$ since $\mathcal{V}_1 = \mathcal{V}_2$ and $s_1 = s_2$.

- If $f = \texttt{branch}\ c\ \ l'\ l''$,

1. Let $\mathcal{L}_2 = \mathcal{L}_1[l' \mapsto (n' + 1, d' +\!\!+ [\mathcal{V}_1], \mathcal{V}'_0)]$ where $(n', d', \mathcal{V}'_0) = \mathcal{L}_1(l')$. By inspection of $\mathsf{T_B}$ and the fact that $\mathcal{V}' = \mathcal{V}_1$ we have:
$\mathcal{L}' = \mathcal{L}_2[l'' \mapsto (n'' + 1, d'' +\!\!+ [\mathcal{V}_1], \mathcal{V}''_0)]$ where $(n'', d'', \mathcal{V}''_0) = \mathcal{L}_2(l'')$

2. Also, we have:

- $b_1 = \texttt{branch}\ c\ \ l'\ l''$
- $b'_1 = \texttt{branch}\ \widehat{\mathcal{V}}_1(c)\ l'_{n'+1}\ l''_{n''+1}$

We want to prove that $\forall s \exists s'$ such that
$\langle b'_1, s \oplus \widehat{\mathcal{V}}_1(s_1), \mathcal{V}_1\rangle \rightarrow^0 \langle \texttt{branch}\ \widehat{\mathcal{V}}_2(c)\ l'_{\pi_1(\mathcal{L}'(l'))}\ l''_{\pi_1(\mathcal{L}'(l''))}, s' \oplus \widehat{\mathcal{V}}_2(s_2), \mathcal{V}_2\rangle$
It suffices to take $s' = s$ since $\mathcal{V}_1 = \mathcal{V}_2,\ s_1 = s_2$ and

$\texttt{branch}\ \widehat{\mathcal{V}}_1(c)\ l'_{n'+1}\ l''_{n''+1} = \texttt{branch}\ \widehat{\mathcal{V}}_2(c)\ l'_{\pi_1(\mathcal{L}'(l'))}\ l''_{\pi_1(\mathcal{L}'(l''))}$ because:

- $\mathcal{V}_1 = \mathcal{V}_2$
- $\pi_1(\mathcal{L}'(l')) = n' + 1$
- $\pi_1(\mathcal{L}'(l'')) = n'' + 1$

Inductive Case, for $n > 0$:
Suppose that $\langle b_1, s_1, \mathcal{V}_1\rangle \rightarrow^1 \langle b_0, s_0, \mathcal{V}_0\rangle \rightarrow^{n-1} \langle f, s_2, \mathcal{V}_2\rangle$
Because of $\langle b_1, s_1, \mathcal{V}_1\rangle \rightarrow^1 \langle b_0, s_0, \mathcal{V}_0\rangle$, we have that:

- $b_1 = (x := a_0); b_0$, for some x and a_0
- $s_0 = s_1[x \mapsto [\![a_0]\!](s_1)]$ (*)
- $\mathcal{V}_0 = \mathcal{V}_1[x \mapsto \mathcal{V}_1(x) + 1]$ (**)

- $\mathsf{T_B}\,(b_1, l)(\mathcal{V}_1, \mathcal{L}_1) = (x_{\mathcal{V}_1(x)+1} := \widehat{\mathcal{V}}_1(a_0); b'_0, (\mathcal{V}', \mathcal{L}'))$
  where $(b'_0, (\mathcal{V}', \mathcal{L}')) = \mathsf{T_B}\,(b_0, l)(\mathcal{V}_0, \mathcal{L}_1)$

We have:

- $b = (\overrightarrow{y := e}) +\!\!+\, x := a_0; b_0$ (recall $b = \overrightarrow{(y := e)}; b_1$ and $b_1 = x := a_0; b_0$) (1)
- $\mathsf{T_B}\,(b_0, l)(\mathcal{V}_0, \mathcal{L}_1) = (b'_0, (\mathcal{V}', \mathcal{L}'))$ (2)
- $\langle b_0, s_0, \mathcal{V}_0\rangle \;\rightarrow^{n-1} \langle f, s_2, \mathcal{V}_2\rangle$ (3)
- $\mathcal{V}_0 = \mathsf{inc}\,(\mathcal{V})[(\overrightarrow{y := e}) +\!\!+\, x := a_0]$ because $\mathcal{V}_1 = \mathsf{inc}\,(\mathcal{V})[(\overrightarrow{y := e})]$ and (**) (4)

Because of (1), (2), (3) and (4) we can use the IH to conclude that $\mathcal{V}' = \mathcal{V}_2$.

- If $f = \mathtt{return}\, a$,
  1. From the IH, we can conclude that $\mathcal{L}' = \mathcal{L}_1$
  2. From the IH, we can also conclude that:

     $\forall s \exists s'$: (5)

     $\langle b'_0, s \oplus \widehat{\mathcal{V}_1[x \mapsto \mathcal{V}_1(x)+1]}(s_1[x \mapsto [\![a_0]\!](s_1)]), \mathcal{V}_0\rangle \;\rightarrow^{n-1} \langle \mathtt{return}\, \widehat{\mathcal{V}}_2(a)\,, s' \oplus \widehat{\mathcal{V}}_2(s_2), \mathcal{V}_2\rangle$

     Recall the equalities on (*) and (**).

     For a fixed $s \in \Sigma^{\mathbf{SA}}$: (6)
     $\langle x_{\mathcal{V}_1(x)+1} := \widehat{\mathcal{V}}_1(a_0); b'_0, s \oplus \widehat{\mathcal{V}}_1(s_1), \mathcal{V}_1\rangle \rightarrow$
     $\langle b'_0, s \oplus \widehat{\mathcal{V}}_1(s_1)[x_{\mathcal{V}_1(x)+1} \mapsto [\![\widehat{\mathcal{V}}_1(a_0)]\!](s \oplus \widehat{\mathcal{V}}_1(s_1))], \mathcal{V}_1[x \mapsto \mathcal{V}_1(x)+1]\rangle$

     Fix $s \in \Sigma^{\mathbf{SA}}$. Because of (6) and (5), it suffices to prove that:

     $s \oplus \widehat{\mathcal{V}}_1(s_1)[x_{\mathcal{V}_1(x)+1} \mapsto [\![\widehat{\mathcal{V}}_1(a_0)]\!](s \oplus \widehat{\mathcal{V}}_1(s_1))] =$ (7)

     $s \oplus \widehat{\mathcal{V}_1[x \mapsto \mathcal{V}_1(x)+1]}(s_1[x \mapsto [\![a_0]\!](s_1)])$

     and $\mathcal{V}_1[x \mapsto \mathcal{V}_1(x)+1] = \mathcal{V}_0$.
     The latter is given by (**). Also, we can prove that:

     $\widehat{\mathcal{V}}_1(s_1)[x_{\mathcal{V}_1(x)+1} \mapsto [\![\widehat{\mathcal{V}}_1(a_0)]\!](s \oplus \widehat{\mathcal{V}}_1(s_1))] = \widehat{\mathcal{V}_1[x \mapsto \mathcal{V}_1(x)+1]}(s_1[x \mapsto [\![a_0]\!](s_1)])$

     It is easy to see that the right hand side and left hand side assign the same values to variables distinct of $x_{\mathcal{V}(x)+1}$. For $x_{\mathcal{V}_1(x)+1}$, the left hand side state and the right hand side state assign the values $[\![\widehat{\mathcal{V}}_1(a_0)]\!](s \oplus \widehat{\mathcal{V}}_1(s_1))$ and $[\![a_0]\!](s_1)$, which are equal by Lemma 9.

– $f = \texttt{goto}\, l$ or $f = \texttt{branch}\ c\ \ l'\ l''$
The argument is analogous to the previous one. For example, if $f = \texttt{goto}\, l$,

1. We can use the IH to conclude that
$\mathcal{L}' = \mathcal{L}_1[l' \mapsto (\pi_1\mathcal{L}_1(l') + 1, \pi_2\mathcal{L}_1(l') \mathbin{+\!\!+} [\mathcal{V}'], \pi_3\mathcal{L}_1(l'))]$

2. We can use the IH to also conclude that:

$\langle b'_0, s \oplus \widehat{\mathcal{V}_1[x \mapsto \mathcal{V}_1(x)+1]}(s_1[x \mapsto [\![a_0]\!](s_1)]), \mathcal{V}_0\rangle \ \rightarrow^{n-1} \langle \texttt{goto}\, l_{\pi_1(\mathcal{L}'(l'))}, s' \oplus \widehat{\mathcal{V}}_2(s_2), \mathcal{V}_2\rangle$

so it suffices to prove (7), which we have already proved in the previous case.

□

In Proposition 9 we will establish a property analogous to the one in the previous proposition, Proposition 8, but now dealing with the relation $\multimap\!\!\to$.

Proposition 9 *Let $p \in \mathbf{P}_\bullet$ and let p' in $\mathbf{P}^{\mathbf{SA}}_\bullet$ be such that $p' = \mathcal{T}(p)$. Let l be a label of p. Let $\mathcal{V}_1 \in \mathbf{VS}$ be such that* $\mathsf{ctx}(l,p) = (\mathcal{V}_1, \mathcal{L}_1)$.
Let $s_1, s_2, \mathcal{V}_2$ be such that $\langle l, s_1, \mathsf{inc}\,(\mathcal{V}_1))\rangle \multimap\!\!\to_p \langle \texttt{return}\, a\ , s_2, \mathcal{V}_2\rangle$ (*)
we have, $\forall s \exists s'$, $\langle l, s \oplus \widehat{\mathsf{inc}\,(\mathcal{V}_1)}(s_1), \mathsf{inc}\,(\mathcal{V}_1)\rangle \multimap\!\!\to_{p'} \langle \texttt{return}\, \widehat{\mathcal{V}}_2(a)\ , s' \oplus \widehat{\mathcal{V}}_2(s_2), \mathcal{V}_2\rangle$

Proof. Let b be the block labeled by l in p and $(l : \phi b')$ its translation in p'.
Let $\mathcal{L}'_1 = \mathcal{L}_1[l \mapsto (\pi_1\mathcal{L}_1(l), \pi_2\mathcal{L}_1(l), \mathsf{inc}\,(\mathcal{V}_1))]$ and let $\mathsf{T_B}\,(b,l)(\mathsf{inc}\,(\mathcal{V}_1), \mathcal{L}'_1) = (b', (\mathcal{V}_0, \mathcal{L}_0))$
From assumption (*) and Definition 15, we have that
$\langle b, s_1, \mathsf{inc}\,(\mathcal{V}_1)\rangle \ \rightarrow^* \langle \texttt{return}\, a\ , s_2, \mathcal{V}_2\rangle$.
So, by applying Proposition 8(part 2, case return), we have
$\forall s \exists s', \langle b', s \oplus \widehat{\mathsf{inc}\,(\mathcal{V}_1)}(s_1), \mathsf{inc}\,(\mathcal{V}_1)\rangle \ \rightarrow^n \langle \texttt{return}\, \widehat{\mathcal{V}}_2(a)\ , s' \oplus \widehat{\mathcal{V}}_2(s_2), \mathcal{V}_2\rangle$.
Using Definition 19 follows: $\langle l, s \oplus \widehat{\mathsf{inc}\,(\mathcal{V}_1)}(s_1), \mathsf{inc}\,(\mathcal{V}_1)\rangle \multimap\!\!\to_{p'} \langle \texttt{return}\, \widehat{\mathcal{V}}_2(a)\ , s' \oplus \widehat{\mathcal{V}}_2(s_2), \mathcal{V}_2\rangle$
□

We continue now with a lemma used in the proof of Proposition 10. Proposition 10 ensures that if a transition is made and execution moves to another block, then in the correspondent translated program, this execution is also possible.

Lemma 12 *Let $p \in \mathbf{P}_\bullet$ and $l, l' \in$* $\mathsf{labels}(p)$. *Let us have* $\mathsf{ctx}(p,l) = (\mathcal{V}, \mathcal{L})$.

$$\#(\pi_2(\mathcal{L}(l')) = \pi_1(\mathcal{L}(l'))$$

Proof. Follows from the Definition 21 and by analyzing inductively the function $\mathsf{T_L}$. □

Proposition 10 *Let $p \in \mathbf{P}_\bullet$ and let p' in $\mathbf{P}^{\mathbf{SA}}_\bullet$ be such that $p' = \mathcal{T}(p)$. Let $l, l' \in \mathsf{labels}(p)$. Let $(l : b)$ and $(l' : b')$ be labeled blocks in p and $(l : \phi b_0)$ and $(l' : \phi' b_1)$ be their translations in p'. Let $\mathsf{ctx}(p, l) = (\mathcal{V}_1, \mathcal{L}_1)$. Let s_1, s_2 and $\mathcal{V}_2$ be such that: $\langle l, s_1, \mathsf{inc}(\mathcal{V}_1)\rangle \multimap\!\!\rightarrow_p \langle l', s_2, \mathcal{V}_2\rangle$. Then, $\forall s \exists s'$, $\langle l, s \oplus \widehat{\mathsf{inc}(\mathcal{V}_1)}(s_1), \mathsf{inc}(\mathcal{V}_1)\rangle \multimap\!\!\rightarrow_{p'} \langle l', s' \oplus \widehat{\mathcal{V}}_2(s_2), \mathcal{V}_2\rangle$*

Proof. The proof will be by case analysis in the constitution of b, in particular in the last instruction of b. Let $\mathcal{L}_2 = \mathcal{L}_1[l \mapsto (\pi_1\mathcal{L}_1(l), \pi_2\mathcal{L}_1(l), \mathsf{inc}(\mathcal{V}_1))]$ and let $\mathsf{T_B}(b, l)(\mathsf{inc}(\mathcal{V}_1), \mathcal{L}_2) = (b_0, (\mathcal{V}', \mathcal{L}'))$, for some $\mathcal{V}'$ and $\mathcal{L}'$.

- Case b has the form $\overrightarrow{(y := a)}; \mathtt{goto}\, l'$.

  Let s be an arbitrary state. From $\langle l, s_1, \mathsf{inc}(\mathcal{V}_1)\rangle \multimap\!\!\rightarrow_p \langle l', s_2, \mathcal{V}_2\rangle$ we know that (by Definition 15), for some n and some $\mathcal{V}_3$, $\langle b, s_1, \mathsf{inc}(\mathcal{V}_1)\rangle \rightarrow^n_p \langle \mathtt{goto}\, l', s_2, \mathcal{V}_3\rangle$ and $\mathcal{V}_2 = \mathsf{vrs}(l', p)$. By Proposition 8, we have that $\mathcal{V}' = \mathcal{V}_3$.

  Also, by Proposition 8 (part 1, case goto) we have that:

  - $\mathcal{L}' = \mathcal{L}_2[l' \mapsto (\pi_1\mathcal{L}_2(l') + 1, \pi_2\mathcal{L}_2(l') +\!\!+ [\mathcal{V}_3], \pi_3\mathcal{L}_2(l'))])$

  From this, using Proposition 8 (part 2, case goto), we have that
  $\langle b_0, s \oplus \widehat{\mathsf{inc}(\mathcal{V}_1)}(s_1), \mathsf{inc}(\mathcal{V}_1)\rangle \rightarrow^* \langle \mathtt{goto}\, l'_{\pi_1(\mathcal{L}'(l'))}, s' \oplus \widehat{\mathcal{V}}_3(s_2), \mathcal{V}_3\rangle$, for some s''. Hence, by the *goto rule* from Definition 19,
  $\langle l, s \oplus \mathsf{inc}(\widehat{\mathcal{V}}_1)(s_1), \mathsf{inc}(\mathcal{V}_1)\rangle \multimap\!\!\rightarrow_{p'} \langle l', \mathsf{upd}(\phi', \pi_1(\mathcal{L}'(l')), s' \oplus \widehat{\mathcal{V}}_3(s_2)), \mathsf{vrs}(l', p')\rangle$. So, we just have to prove that $\mathsf{upd}(\phi', \pi_1(\mathcal{L}'(l')), s' \oplus \widehat{\mathcal{V}}_3(s_2)) = s' \oplus \widehat{\mathcal{V}}_2(s_2)$, for some s', and $\mathsf{vrs}(l', p') = \mathsf{vrs}(l', p) = \mathcal{V}$. The latter is immediate since $p' = \mathcal{T}(p)$. We will now show the former holds for $s' = s'' \oplus \mathcal{V}_3(s_2)$

  Let $s_* = \mathsf{upd}(\phi', \pi_1(\mathcal{L}'(l')), s' \oplus \widehat{\mathcal{V}}_3(s_2))$ and by Definition 10, we have that

  $s_* = s' \oplus \widehat{\mathcal{V}}_3(s_2)[x_i \mapsto [\![\overrightarrow{x}[\pi_1(\mathcal{L}'(l'))]]\!](s' \oplus \widehat{\mathcal{V}}_3(s_2)) \mid x_i := \phi(\overrightarrow{x}) \in \phi']$
  Let $y_j \in \mathbf{Var^{SA}}$

  - Case $y_j \notin \mathsf{dom}(\phi')$ and $\mathcal{V}_3(y) = j$.
    $s_*(y_j) = \widehat{\mathcal{V}}_3(s_2)(y_j) = s_2(y)$
    We want to show that $s_2(y) = (s' \oplus \widehat{\mathcal{V}}_2(s_2))(y_j)$
    As $y_2 \notin \mathsf{dom}(\phi')$ and $\mathcal{V}_2 = \mathsf{vrs}(l', p)$, $s_2(y) = s'(y_j)$
    How $s' = s'' \oplus \mathcal{V}_3(s_2), s'(y_j) = (s'' \oplus \mathcal{V}_3)(s_2)(y_j) = s_2(y_j)$

  - Case $y_j \notin \mathsf{dom}(\phi')$ and $\mathcal{V}_3(y) \neq j$
    $s_*(y_j) = s'(y_j)$.
    We want to show that $s'(y_j) = s' \oplus \widehat{\mathcal{V}}_2(s_2)(y_j)$
    This is true because $\mathcal{V}_2 = \mathsf{vrs}(l', p)$ and $y_j \notin \mathsf{dom}(\phi')$.

  - Case $y_j \in \mathsf{dom}(\phi')$, *i.e.*, $y_j := \phi(\overrightarrow{y}) \in \phi'$
    Let $n = \pi_1(\mathcal{L}'(l'))$,

$$\begin{aligned} s_*(y_j) &= [\![\overrightarrow{y}[n]]\!](s' \oplus \widehat{\mathcal{V}}_3(s_2)) \\ &= (s' \oplus \widehat{\mathcal{V}}_3(s_2))(y_{W(y)}), \textit{with } W = \pi_2(\mathcal{L}'(l'))[n] \text{(by definition of sync)} \\ &= (s' \oplus \widehat{\mathcal{V}}_3(s_2)(y_{\mathcal{V}_3(y)}) \ (W = \mathcal{V}_3, \textit{ which follows by Lemma 12}) \\ &= s_2(y) \ \textit{(From the definition of } \oplus \textit{ and } \hat{} \textit{ operators.)} \end{aligned}$$

Hence, we can conclude that $\mathsf{upd}\,(\phi', \pi_1(\mathcal{L}'(l')), s' \oplus \widehat{\mathcal{V}}_3(s_2)) = s' \oplus \mathcal{V}_2(s_2)$

Now, we want to show that $s_2(y) = s' \oplus \mathcal{V}_2(s_2)(y_j)$. This is true because $\mathcal{V}_2(y) = j$, since $\mathsf{vrs}(l, p) = \mathcal{V}_2$ by Definition 14 and $y_j \in \mathsf{dom}\,(\phi')$. For the $y_j \notin \mathsf{dom}\,(\phi')$, as $s' = s'' \oplus \mathcal{V}_3(s_2)$, this is also proved.

- Case b has the form $\overrightarrow{(y := a)}$; `branch` $c\ l'\ l''$
  This case is analogous to the previous one.

□

We now present two more lemmas (Lemma 13 and Lemma 14) and we bring forward a main theorem. This theorem will prove the completeness of our translation.

Lemma 13 *Let $p \in \mathbf{P}_\bullet$ and $l \in \mathsf{labels}(p)$. We have that:*

$$\mathsf{vrs}(l, p) = \mathsf{inc}\,(\pi_1(\mathsf{ctx}(p, l)))$$

Proof. The lemma follows from Definition 14 and Definition 21, and by inspecting the translation function $\mathcal{T}$, in what concerns the construction of the synchronization blocks ϕ . □

Lemma 14 *Let $p \in \mathbf{P}_\bullet$. Let $\mathcal{V}_1 = \mathsf{inc}\,(\pi_1(\mathsf{initC}\,p))$*
For any $n \geq 1, s_1, s_2, l$ and $\mathcal{V}_2$ we have that $\langle \bullet, s_1, \mathcal{V}_1 \rangle \multimap\!\!\to_p^n \langle l, s_2, \mathcal{V}_2 \rangle \implies \mathcal{V}_2 = \mathsf{inc}\,(\pi_1 \mathsf{ctx}(p, l))$.

Proof. Assume $\langle \bullet, s_1, \mathcal{V}_1 \rangle \multimap\!\!\to_p^{n-1} \langle l', s_3, \mathcal{V}_3 \rangle \multimap\!\!\to_p \langle l, s_2, \mathcal{V}_2 \rangle$, where in the case $n = 1$ we have that $l' = \bullet, s_1 = s_3$ and $\mathcal{V}_1 = \mathcal{V}_3$.
We want to prove that $\mathsf{inc}\,(\pi_1 \mathsf{ctx}(p, l)) = \mathcal{V}_2$
Inspecting Definition 15, we see that $\mathcal{V}_2 = \mathsf{vrs}(l, p)$. Hence, using Lemma 13, $\mathcal{V}_2 = \mathsf{inc}\,(\pi_1 \mathsf{ctx}(p, l))$.
□

Theorem 2 (Completeness) *Let $p \in \mathbf{P}_\bullet$ and let p' be $\mathcal{T}(p)$. Let $\mathcal{V}_1 = \mathsf{inc}\,(\pi_1(\mathsf{initC}\,p))$. For any $n, s_1, s_2, \mathcal{V}_2, f$,*
$\langle \bullet, s_1, \mathcal{V}_1 \rangle \multimap\!\!\to_p^n \langle f, s_2, \mathcal{V}_2 \rangle \implies \forall s \exists s',\ \langle \bullet, s \oplus \widehat{\mathcal{V}}_1(s_1), \mathcal{V}_1 \rangle \multimap\!\!\to_{p'}^n \langle \widehat{\mathcal{V}}_2(f), s' \oplus \widehat{\mathcal{V}}_2(s_2), \mathcal{V}_2 \rangle$

Proof. By induction on $n \in \mathbb{N}_0$.

Case $n = 0$.
We have that: $\langle \bullet, s_1, \mathsf{inc}\,(\mathcal{V}_1)\rangle \multimap\!\!\rightarrow_p^0 \langle f, s_2, \mathcal{V}_2\rangle$
So, we know that:

- $s_1 = s_2$
- $\mathcal{V}_2 = \mathsf{inc}\,(\mathcal{V}_1)$
- $f = \bullet$

We want to prove that: $\forall s \exists s', \langle \bullet, s \oplus \mathcal{V}_1(s_1), \mathcal{V}_1\rangle \multimap\!\!\rightarrow_{p'}^0 \langle \widehat{\mathcal{V}}_2(f), s' \oplus \widehat{\mathcal{V}}_2(s_2), \mathcal{V}_2\rangle$, which is immediate since $\widehat{\mathcal{V}}_2(f) = \widehat{\mathcal{V}}_2(\bullet) = \bullet$ and then we just have to take $s' = s$.

Case $n \geq 1$
We have that: $\langle \bullet, s_1, \mathcal{V}_1\rangle \multimap\!\!\rightarrow_p^{n-1} \langle l, s_0, \mathcal{V}_0\rangle \multimap\!\!\rightarrow_p \langle f, s_2, \mathcal{V}_2\rangle$. Let s be an arbitrary state.

By IH we have $\langle \bullet, s \oplus \widehat{\mathcal{V}}_1(s_1), \mathcal{V}_1\rangle \multimap\!\!\rightarrow_p^{n-1} \langle l, s'' \oplus \widehat{\mathcal{V}}_0)(s_0), \mathcal{V}_0\rangle$, for some s''. We now will prove that $\langle l, s'' \oplus \widehat{\mathcal{V}}_0(s_0), \mathcal{V}_0\rangle \multimap\!\!\rightarrow_{p'} \langle \widehat{\mathcal{V}}_2(f), s' \oplus \widehat{\mathcal{V}}_2(s_2), \mathcal{V}_2\rangle$ for some s'.

1. Case $f = \texttt{return}\,a$.
   By Lemma 14 we have that $\mathcal{V}_0 = \mathsf{inc}\,(\pi_1(\mathsf{ctx}(p, l)))$. We also know that:
   $\langle l, s_0, \mathcal{V}_0\rangle \multimap\!\!\rightarrow_p \langle \texttt{return}\,a\ , s_2, \mathcal{V}_2\rangle$
   So by applying Proposition 9, we have that for some s'
   $\langle l, s'' \oplus \widehat{\mathcal{V}}_0(s_0), \mathcal{V}_0\rangle \multimap\!\!\rightarrow_{p'} \langle \texttt{return}\,\widehat{\mathcal{V}}_2(a\,), s' \oplus \widehat{\mathcal{V}}_2(s_2), \mathcal{V}_2\rangle$.

2. Case $f = l'$,
   Again, by Lemma 14 we have that $\mathcal{V}_0 = \mathsf{inc}\,(\pi_1(\mathsf{ctx}(p, l)))$. We also know that:
   $\langle l, s_0, \mathcal{V}_0\rangle \multimap\!\!\rightarrow_p \langle l', s_2, \mathcal{V}_2\rangle$
   So by applying Proposition 10, we have that $\langle l, s' \oplus \widehat{\mathcal{V}}_0(s_0), \mathcal{V}_0\rangle \multimap\!\!\rightarrow_{p'} \langle l', s' \oplus \widehat{\mathcal{V}}_2(s_2), \mathcal{V}_2\rangle$, for some s'.

□

4.3.3 *Correctness*

Finally, we can establish the correctness of the translation $\mathcal{T}$ into SA form. This is achieved by combining the soundness and completeness results of the previous subsections, but also using the equivalence of the alternative semantics we introduced both for SA and $\mathcal{GL}$.

Theorem 3 (Correctness of the translation into SA form) *Let $p \in \mathbf{P}_\bullet$ and let $p' = \mathcal{T}(p)$. Let $s \in \Sigma$ and let $s', s'' \in \Sigma^{\mathbf{SA}}$ be such that $s' = s'' \oplus \widehat{\mathsf{inc}\,(\pi_1(\mathsf{initC}\,p))}(s)$.*

The execution of p in state s is finite iff *the execution of p' in state s' if finite. Furthermore, if both executions are finite, the results of both executions are the same.*

Proof. The first statement follows from Proposition 4, Proposition 5, Theorem 2 and Theorem 1 we give only details of the proof of the second statement which requires similar ideas:

1. Assume the execution of p in state s $w.r.t. \rightsquigarrow$ is finite and its terminal configuration is $\langle \mathtt{return}\, a\ , s_2 \rangle$ and so the result of the execution is $[\![a]\!](s_2)$. By Proposition 4, the execution of p in state s $w.r.t. \multimap\!\!\!\rightarrow$ is finite and gives a terminal configuration $\langle \mathtt{return}\, a\ , s_2, \mathcal{V}_2 \rangle$ for some $\mathcal{V}_2$. Therefore, by Theorem 2, the execution of p' in state s' $w.r.t. \multimap\!\!\!\rightarrow$ produces a terminal configuration $\langle \mathtt{return}\, \widehat{\mathcal{V}}_2(a\ ), s''' \oplus \widehat{\mathcal{V}}_2(s_2), \mathcal{V}_2 \rangle$, for some s'''. Hence, by Proposition 5, the execution of p' in state s' $w.r.t. \rightsquigarrow$ produces terminal configuration $\langle \mathtt{return}\, \widehat{\mathcal{V}}_2(a\ ), s''' \oplus \widehat{\mathcal{V}}_2(s_2) \rangle$ and its return value is $[\![\widehat{\mathcal{V}}_2(a\ )]\!](s''' \oplus \widehat{\mathcal{V}}_2(s_2))$, which is equal to $[\![a]\!](s_2)$ by Lemma 9, as we wanted to show.

2. The converse has similar details but using Theorem 1 instead of Theorem 2.

$\square$

5 FINAL REMARKS

This chapter concludes the thesis. We begin by analyzing some possible improvements to our SA translation function, $\mathcal{T}$. Then we discuss the relation between the programs obtained by the translation function and functional programs. Finally we present some conclusions and future work.

5.1 SA TRANSLATION

As we saw, the conversion of programs with jumps is done with the help of a notational trick, the ϕ-functions. They are used to somehow merge possible different values of a variable at the joint points of the CFG. For instance, in some node with two incoming edges (doors), the expression $\phi(x_1, x_2)$ has the value of x_1 if the node is reached on the first in-edge, and of x_2 if the node is reached on the second in-edge.

In Chapter 3 we have implemented a naive SA translation with obvious possible improvements. We choose to work with a naive version to begin with, because our focus was to prove its correction with respect to the structural operational semantics, and we anticipate that it would be a very intricate proof.

In our naive translation we choose synchronize every variable of the program at the beginning of each block, even if the block has a sole in-edge. So, for a program with n blocks and m variables, our translation produces $n \times m$ new assignments.

5.1.1 *Optimal placement of ϕ-functions*

There are obvious optimizations to our translation. For instance, we would not need to place ϕ-functions in nodes with a sole in-edge.

A tempting but erroneous optimization is to synchronize only variables used in the block. This is wrong because when two different definitions of a variable reach a node, even if this variable is not used in that block, it can be used in the subsequent nodes and so one needs to know which is the value of the variable, and for that we need to synchronize it with a ϕ-function.

```
• :   n := 10;
      i := 0;
      s := 0;
      goto l
l :   branch i ≤ n l′ l″
l′ :  s := s + i;
      i := i + 1;
      goto l
l″ :  return s
```

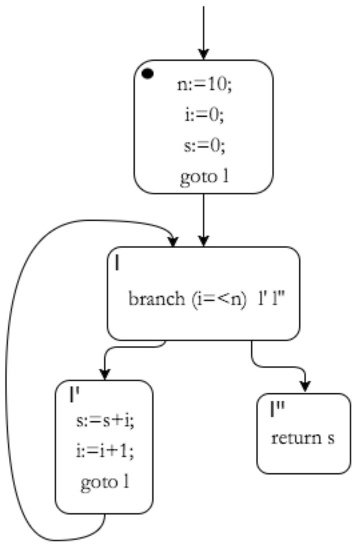


Figure 10.: Program *sum* and its CFG

The only place one really needs to place a ϕ-function in SSA form is in the beginning of the blocks where different definitions of a variable reach the block by different in-edges.

Consider example in Figure 10. In this case, only one definition of n reaches block l so there is no need of a ϕ-function for n in that block. Also, only one version of s reaches block l''.

To calculate the minimum set of ϕ-function needed in a correct translation into SSA form, one uses the notion of *dominance* and *dominance frontiers*.

If a node a contains a definition of some variable, say x, then any node in the dominance region of a that uses x needs a ϕ-function because any node in the dominance region of a is reachable by two different definitions of x.

We say that a node a *dominates* a node b , if every path from the entry node to b most go through a. By definition every node dominates itself. The dominance frontier of a is the region of the CFG dominated by a. In particular, if there is a node c dominated by b and b is dominated by a, c will be in the dominance frontier of a. The importance of dominance region to SSA relates to the fact that, as said in Appel (1998a):

> "Whenever a node a contains a definition of some variable x, then any node in the dominance frontier of a needs a ϕ-function."

There are efficient algorithms for calculating the dominator tree and dominance frontiers that can be found, for instance in Appel (1998b), Muchnick (1997b) and Wolfe (1996).

$$\begin{tikzcd}\mathbf{P}\times\mathbf{OP} \arrow[r] & \mathbf{P}_\bullet\times\mathbf{OP} \arrow[r, "\mathcal{R}"] \arrow[rr, bend right, "\mathcal{T}"'] & \mathbf{P}^{\mathbf{I}}_\bullet\times\mathbf{C} \arrow[r, "\mathcal{S}"] & \mathbf{P}^{\mathbf{SA}}_\bullet \arrow[r] & \mathbf{P}^{\mathbf{SA}}\end{tikzcd}$$

Figure 11.: Translation into SA format with $\mathbf{OP}$

5.1.2 *Adapting the translation function $\mathcal{T}$*

Our naive translation function $\mathcal{T}$ can be adapted to work with the optimal placement of the ϕ-functions, assuming this information is previously calculated and passed to $\mathcal{T}$. Let us illustrate how to do this.

We assume that, along with the original program, we receive the set of variables that need to be synchronized in the beginning of each block.

Let $\mathbf{OP}_p = \mathbf{L}_p \to \mathcal{P}(\mathbf{Var}_p)$ be the set of functions that, for each label gives the set of variables to be synchronized in the corresponding labeled block of the program p. We will let $\mathcal{O}, \mathcal{O}'$... range over $\mathbf{OP}_p$. We will drop the subscript p for the sake of simplicity.

For the translation process we need to associate the set of variables to be synchronized in the beginning of each block to the name of the block. For that, the second component of the context needs to be slightly modified.

For a program p we let

$$\mathbf{Ls}_p = \mathbf{L}_p \to \mathbb{N} \times \mathbf{Vs}_p^* \times \mathcal{P}(\mathbf{Var}_p^{\mathbf{SA}\ *})$$

As usual, for the sake of simplicity, we will drop the subscript p. So, $\mathcal{L} \in \mathbf{Ls}$ is a function that, for each label gives:

- The number of already discovered in-edges (doors) of the labeled block;
- The version function associated to each in-edge;
- The set of the variables that need to be synchronized.

The set of contexts must now be defined as follows: $\mathbf{C} = \mathbf{VS} \times \mathbf{Ls}$

The translation process is described in Figure 11. We keep the names of the functions, despite they are slightly different. The main differences are:

- The initialization of the context, initC, associates the empty set of synchronization variables;
- In the beginning of the translation of each block (done by $\mathsf{T_L}$), the variables to be synchronized in that block are placed in the context (associated to the respective label) with the adequate version;
- The synchronization function (done by sync) will then assign these variables with appropriate ϕ-functions.

$$
\begin{array}{l}
\mathcal{R} : \mathbf{P_\bullet} \times \mathbf{OP} \to \mathbf{P^I_\bullet} \times \mathbf{C} \\
\mathcal{R}\,(p,\mathcal{O}) = \mathsf{T_L}\; p\; \mathcal{O}\; (\mathsf{initC}\; p) \\
\\
\mathsf{initC} : \mathbf{P_\bullet} \to \mathbf{C} \\
\mathsf{initC}\; p = (\mathcal{V}_0, [l \mapsto (0,[],\emptyset) \mid l \in \mathsf{labels}(p)] \\
\quad \text{where} \quad \mathcal{V}_0 \; = [x \mapsto -1 \mid x \in \mathsf{vars}(p)] \\
\\
\mathsf{T_L} : (\mathbf{L_\bullet} \times \mathbf{B})^* \to \mathbf{OP} \to \mathbf{C} \to (\mathbf{L_\bullet} \times \mathbf{B^{SA}})^* \times \mathbf{C} \\
\mathsf{T_L}\,((l,b) : t)\; \mathcal{O}\; (\mathcal{V},\mathcal{L}) = ((l,b') : t', \mathcal{C}'') \\
\quad \text{where} \quad
\begin{array}{rl}
(n,d,\_) & = \mathcal{L}\,(l) \\
\mathcal{V}_1 & = \mathcal{V} \oplus [x \mapsto \mathcal{V}\,(x) + 1 \mid x \in \mathcal{O}\,(l)] \\
(b',\mathcal{C}') & = \mathsf{T_B}\,(b,l)(\mathcal{V}_1, \mathcal{L}\,[l \mapsto (n,d,\widehat{\mathsf{inc}\,\mathcal{V}}\,(\mathcal{O}\,(l)))]) \\
(t',\mathcal{C}'') & = \mathsf{T_L}\; t\; \mathcal{C}'
\end{array} \\
\mathsf{T_L}\; [\,]\, \mathcal{C} = ([\,],\mathcal{C}) \\
\\
\mathsf{T_B} : \mathbf{B} \times \mathbf{L} \to \mathbf{C} \to \mathbf{B^{SA}} \times \mathbf{C} \\
\mathsf{T_B}\,(x := a; b, l)\; (\mathcal{V},\mathcal{L}) = (x_{\mathcal{V}(x)+1} := \widehat{\mathcal{V}}\,(a); b', \mathcal{C}') \\
\quad \text{where} \quad
\begin{array}{rl}
\mathcal{C} & = (\mathcal{V}\,[x \mapsto \mathcal{V}\,(x) + 1], \mathcal{L}) \\
(b',\mathcal{C}') & = \mathsf{T_B}\,(b,l)\; \mathcal{C}
\end{array} \\
\mathsf{T_B}\,(\mathtt{goto}\, l', l)\; (\mathcal{V},\mathcal{L}) = (\mathtt{goto}\, l'_{n+1}, \mathcal{C}) \\
\quad \text{where} \quad
\begin{array}{rl}
(n,d,\mathcal{V}_0) & = \mathcal{L}\,(l') \\
\mathcal{C} & = (\mathcal{V}, \mathcal{L}\,[l' \mapsto (n+1, d \mathbin{+\!\!+} [\mathcal{V}], \mathcal{V}_0)])
\end{array} \\
\mathsf{T_B}\,(\mathtt{return}\, a, l)\; (\mathcal{V},\mathcal{L}) = (\mathtt{return}\, \widehat{\mathcal{V}}\,(a), (\mathcal{V},\mathcal{L})) \\
\mathsf{T_B}\,(\mathtt{branch}\; c\; l'\; l'', l)\; (\mathcal{V},\mathcal{L}) = (\mathtt{branch}\; \widehat{\mathcal{V}}\,(c)\; l'_{n+1}\; l''_{n'+1}, \mathcal{C}) \\
\quad \text{where} \quad
\begin{array}{rl}
(n,d,\mathcal{V}_0) & = \mathcal{L}\,(l') \\
\mathcal{L}' & = \mathcal{L}\,[l' \mapsto (n+1, d \mathbin{+\!\!+} [\mathcal{V}], \mathcal{V}_0)] \\
(n',d',\mathcal{V}'_0) & = \mathcal{L}'(l'') \\
\mathcal{L}'' & = \mathcal{L}'[l'' \mapsto (n'+1, d' \mathbin{+\!\!+} [\mathcal{V}], \mathcal{V}'_0)] \\
\mathcal{C} & = (\mathcal{V}, \mathcal{L}'')
\end{array}
\end{array}
$$

Figure 12.: Function $\mathcal{R}$ and its auxiliary functions

$$
\begin{array}{l}
\mathsf{sync} :: \mathbf{L} \to \mathbf{C} \to \Phi \\
\mathsf{sync}\; l\; (\_,\mathcal{L}) = [\, x_i := \phi([x_{\mathcal{V}'(x)} \mid \mathcal{V}' \leftarrow d]) \mid x_i \in \mathcal{W}\,] \\
\quad \text{where} \quad (\_,d,\mathcal{W}) \; = \mathcal{L}\,(l) \\
\\
\mathcal{S} :: \mathbf{P^I_\bullet} \times \mathbf{C} \to \mathbf{P^{SA}_\bullet} \\
\mathcal{S}\,([\,], \_) = [\,] \\
\mathcal{S}\,((\bullet,b) : t, \mathcal{C}) = (\bullet, [\,], b) : \mathcal{S}\,(t, \mathcal{C}) \\
\mathcal{S}\,((l,b) : t, \mathcal{C}) = (l, \mathsf{sync}\; l\, \mathcal{C}, b) : \mathcal{S}\,(t, \mathcal{C})
\end{array}
$$

Figure 13.: Function $\mathcal{S}$

$$\mathcal{T} :: \mathbf{P}_\bullet \times \mathbf{OP} \rightarrow \mathbf{P}_\bullet^{\mathbf{SA}}$$
$$\mathcal{T}(p, \mathcal{O}) = \mathcal{S}(\mathcal{R}(p, \mathcal{O}))$$

Figure 14.: Function $\mathcal{T}$

The details of these new translation function are shown in Figure 12, Figure 13 and Figure 14.

We have made no efforts concerning the correctness of this new translation. We suspect that, if we build upon the fact that the information about the optimal placement of ϕ-functions is correct, the proof of the correctness could be done with some adaptations of the proof of correctness of our naive translation.

The proof of correctness of the complete translation, including the computation of the optimal placement of ϕ-functions, is a completely different task that would require much more effort and that is out of scope of this thesis.

In Figure 15 we can find the program that sums the first n natural numbers, the result of translating it with the naive translation function presented in Chapter 3, and the result of the translation with this adapted version assuming that as input, along with the original program, one receives the following sets of variables to be synchronized in each block: $\{l \mapsto \{i, s\}, l' \mapsto \{\,\}, l'' \mapsto \{\,\}\}$.

We have also prototyped in Haskell this new version of the translation function. (See Appendix B).

5.2 RELATION WITH FUNCTIONAL PROGRAMMING

As observed in the early 1990s by Kelsey in (Kelsey, 1995), the imperative and functional paradigms, although apparently distant, can be related through a correspondence that can be establish between imperative programs in SSA format and functional programs.

The main property that allows to establish the connection between SSA imperative programs and functional programs is the referential transparency. In SSA form each variable has only one definition (is assigned only once) as it also happens in the functional setting.

Let us illustrate the relation with the example presented in Figure 15. We can view the program that results from our naive translation to SSA form as a set of functions, mutually recursive, where each function (except the first one) takes as arguments versions of all the programs variables (n, i, and s in this example)

Basically, each block origins a function and each edge of the CFG is a function call. The left-hand side of the ϕ-assignments in the synchronization preamble of each block is the *formal parameter* of the corresponding function; and the right-hand side argument of the ϕ-assignment is the *actual parameter* of some call to the corresponding function. This direct correspondence works because we are dealing with the result of our naive translation where every variable of the program is synchronized in the beginning of each block, as we can see in the program above:

Original program	**Naive SA translation**	**Adapted SA translation**
$\bullet : n := 10;$ $i := 0;$ $s := 0;$ $\texttt{goto}\, l$	$\bullet : n_1 := 10;$ $i_1 := 0;$ $s_1 := 0;$ $\texttt{goto}\, l_1$	$\bullet : n_1 := 10$ $i_1 := 0;$ $s_1 := 0;$ $\texttt{goto}\, l_1$
$l : \texttt{branch}\ (i \leq n)\ l'\ l''$	$l : i_2 := \phi(i_1, i_4);$ $n_2 := \phi(n_1, n_3);$ $s_2 := \phi(s_1, s_4);$ $\texttt{branch}\ (i_2 \leq n_2)\ l'_1\ l''_1$	$l : i_2 := \phi(i_1, i_3);$ $s_2 := \phi(s_1, s_3);$ $\texttt{branch}\ (i_2 \leq n_1)\ l'_1\ l''_1$
$l' : s := s + 1;$ $i := i + 1;$ $\texttt{goto}\, l$	$l' : i_3 := \phi(i_2);$ $n_3 := \phi(n_2);$ $s_3 := \phi(s_2);$ $s_4 := s_3 + i_3;$ $i_4 := i_3 + 1;$ $\texttt{goto}\, l_2$	$l' : s_3 := s_2 + i_2;$ $i_3 := i_2 + 1;$ $\texttt{goto}\, l_2$
$l'' : \texttt{return}\, s$	$l'' : i_5 := \phi(i_2);$ $n_4 := \phi(n_2);$ $s_5 := \phi(s_2);$ $\texttt{return}\, s_5$	$l'' : \texttt{return}\, s_2$

Figure 15.: The program *sum* and its translations with the two versions of $\mathcal{T}$

$$\begin{aligned}
f() &= \mathsf{let}\ n_1 = 10, i_1 = 0, s_1 = 0\ \mathsf{in}\ f_l(i_1, n_1, s_1)\\
f_l(i_2, n_2, s_2) &= \mathsf{if}\ (i_2 \le n_2)\ \mathsf{then}\ f_{l'}(i_2, n_2, s_2)\ \mathsf{else}\ f_{l''}(i_2, n_2, s_2)\\
f_{l'}(i_3, n_3, s_3) &= \mathsf{let}\ s_4 = s_3 + i_3, i_4 = i_3 + 1\ \mathsf{in}\ f_l(i_4, n_3, s_4)\\
f_{l''}(i_5, n_4, s_5) &= s_5
\end{aligned}$$

In this way, the state of the imperative program is received as the arguments of each function. However, if we apply blindly this recipe to the result of the new translations with optimal placement of ϕ-functions, we obtain the following badly formed functional program because some variables are out of the scope of the function that is trying to access:

$$\begin{aligned}
f() &= \mathsf{let}\ n_1 = 10, i_1 = 0, s_1 = 0\ \mathsf{in}\ f_l(i_1, s_1)\\
f_l(i_2, s_2) &= \mathsf{if}\ (i_2 \le n_1)\ \mathsf{then}\ f_{l'}(\,)\ \mathsf{else}\ f_{l''}(\,)\\
f_{l'}(\,) &= \mathsf{let}\ s_3 = s_2 + i_2, i_3 = i_2 + 1\ \mathsf{in}\ f_l(i_3, s_3)\\
f_{l''}(\,) &= s_5
\end{aligned}$$

As one can see, function f_l refers to a variable n_1 that is not in its scope and the same problem occurs in function $f_{l'}$ (*w.r.t.* variables s_2 and i_2) and in function $f_{l''}$ (*w.r.t.* variable s_5). These problems can be easily fixed if one uses the notion of nested scope and declare the functions locally as follows:

```
f() = let n_1 = 10, i_1 = 0, s_1 = 0
        in let f_l(i_2, s_2) = if (i_2 ≤ n_1) then let f_l'() = let s_3 = s_2 + i_2, i_3 = i_2 + 1
                                                                  in f_l(i_3, s_3)
                                                   in f_l'()
                                              else let f_l''() = s_2
                                                   in f_l''()
           in f_l(i_1, s_1)
```

The nested declaration of functions allow them to use non-local variables from the functions in which they are locally declared.

So, the same general ideas about converting SSA imperative programs into functional programs work if the entrance block corresponds to the top function and the other blocks correspond to local functions declared as needed.

5.3 CONCLUSIONS AND FUTURE WORK

The aim of this thesis was to study the central ideas of SSA programs in a context of a simple imperative language including jump instructions. Our main goal was to define a translation from programs of the base imperative language into the SSA format and to prove the correctness of the translation process. We think this goal was successfully attained.

We chose to work in a setting of a very simple imperative language in order to focus our attention in the essential and learn from it for future work. Even so, the task of proving correctness of SSA translation was very demanding both on technical aspects and size. We have done this at a detailed level but still there were some aspects treated informally. If on one hand it would have been extremely helpful to have a proof assistant tool to deal with the complexity of size of our proof, on the other hand the effort of formalizing our proof should require quite some time and this would be out of scope of this work. Note, however, that the effort of addressing formal verification of SSA intermediate languages has been considered in Barthe et al. (2012). This is a far more ambitious work, where in particular translation from the source language RTL(Register Transfer Language) into SSA is addressed. The language RTL is used in the formally verified compiler CompCert Leroy (2009) as a first step of compilation of a realistic subset of the C programming language (prior to the compilation of RTL into SSA). This overall framework is highly complex, and this makes very hard to understand in this context the essential ideas for the correct translation into SSA of a source language with jumps.

The proof of correctness of the translation process has put several challenges regarding the definition of the semantics both of source and SSA language and their interrelationship. After some preliminary analysis it was clear for us that we would have to deal with small-step operational semantics and that we needed to capture that the translation preserves semantics of the program in the following sense: the CFG structure of the source code must be preserved and the state transformation inside the blocks must be maintained to adequate versions of the variables. We highlight two key ideas that have been essential to work around this issue: the isolation of the computation inside the blocks and computation between blocks of code; and the introduction of a transition relation that does the bookkeeping of the versions in use. So, we defined an alternative operational semantics, both for the base imperative language and for the SSA language, which split the computation, according to whether it involves a jump or not, and keeps track of information about the variables in order to identify the adequate version of each variable, when relating execution of a base program and of its SSA translation.

Regarding directions for future work, a task for the immediate future would be the correctness proof of the new version of the translation function presented in Section 5.1. We think that this could be done by adapting the proof we have developed in this thesis. Another line of work could be in the context of the relation between SSA imperative programs and CPS functional programs.

In CPS each procedure takes an extra argument, its *continuation*, representing what should be done with the result the function is calculating. Moreover, every argument in a function call must be either a variable or the continuation. Adapting the example of Section 5.2 to CPS we would have the following (by convention the continuation function is represented as a parameter named k)

$$
\begin{array}{l}
f(k) = \mathsf{let}\ n_1 = 10, i_1 = 0, s_1 = 0 \\
\quad \mathsf{in}\ \mathsf{let}\, f_l(i_2, s_2, k) = \mathsf{if}\ (i_2\ \leq\ n_1)\ \mathsf{then}\, \mathsf{let}\, f_{l'}(k) = \mathsf{let}\, s_3 = s_2 + i_2, i_3 = i_2 + 1 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{in}\, f_l(i_3, s_3, k) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{in}\, f_{l'}(k) \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else}\, \mathsf{let}\, f_{l''}(k) = k\ s_2 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{in}\, f_{l''}(k) \\
\qquad \mathsf{in}\, f_l(i_1, s_1, k)
\end{array}
$$

A possible direction for future work is to study the central ideas about the CPS functional programs and their relations with the SSA programs. Mutually inverse translation between CPS and SSA are studied in Ferreira (2014) in a context where the programs only allow the SSA command assignment and conditional. They build between them a syntactic bijection (at code level) and a semantic bijection (at program execution level). We could profit from these ideas and apply them to a significantly richer context where jump commands and recursion functions are present.

BIBLIOGRAPHY

Andrew W. Appel. SSA is functional programming. *SIGPLAN notices*, 33(4):17–20, 1998a.

Andrew W. Appel. *Modern Compiler Implementation in {C,Java,ML}*. Cambridge University Press, 1998b.

Andrew W Appel. *Compiling with continuations*. Cambridge University Press, 2007.

Gilles Barthe, Delphine Demange, and David Pichardie. A formally verified SSA-based middle-end. pages 47–66, 2012.

Alonzo Church. A set of postulates for the foundation of logic. *Annals of mathematics*, pages 346–366, 1932.

Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

Tiago Alexandre da Costa Ferreira. Um estudo sobre a correspondência entre programação funcional com continuações e programação imperativa single assignment. Master's thesis, Universidade do Minho, Braga, 2014.

Richard A Kelsey. A correspondence between continuation passing style and static single assignment form. In *ACM SIGPLAN Notices*, volume 30, pages 13–22. ACM, 1995.

Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009. URL `http://gallium.inria.fr/~xleroy/publi/compcert-backend.pdf`.

Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997a.

Steven S. Muchnick. *Advanced compiler design implementation*. Morgan Kaufmann, 1997b.

Richard Stallman. Using and porting the gnu compiler collection. In *MIT Artificial Intelligence Laboratory*. Citeseer, 2001.

Michael Joseph Wolfe. *High performance compilers for parallel computing*. Addison-Wesley, 1996.

A

APPENDIX

EXAMPLES

```
exemple1 :: PROG
exemplo = [("bullet", (SEQ "i" (CONST 1) (SEQ "j" (CONST 1) (SEQ "k" (CONST 0) (
    GOTO "b"))))), ("b", BRANCH (LE (VAR "k") (CONST 100)) ("c") ("d")), ("c", (
    SEQ "j" (VAR "i") (SEQ "k" (SOMA (VAR "k") (CONST 1) ) (GOTO "e")))), ("d", (
    SEQ "j" (VAR "k") (SEQ "k" (SOMA (VAR "k") (CONST 2) ) (GOTO "e")))), ("e",
    RET (VAR "k"))]

fact :: PROG
fact = [("bullet", (SEQ "x" (CONST 5) (SEQ "f" (CONST 1) (SEQ "c" (CONST 1) (GOTO
     "l"))))), ("l", BRANCH (LE (VAR "c") (VAR "x")) ("l’") ("l’’")), ("l’", (SEQ
    "f" (MULT (VAR "f") (VAR "c")) (SEQ "c" (SOMA (VAR "c") (CONST 1) ) (GOTO "l")
    ))), ("l’’", RET (VAR "f"))]

suma :: PROG
suma = [("bullet", (SEQ "n" (CONST 10) (SEQ "i" (CONST 0) (SEQ "s" (CONST 0) (
    GOTO "l"))))), ("l", BRANCH (LE (VAR "i") (VAR "n")) ("l’") ("l’’")), ("l’", (
    SEQ "s" (SOMA (VAR "s") (VAR "i")) (SEQ "i" (SOMA (VAR "i") (CONST 1) ) (GOTO
    "l")))), ("l’’", RET (VAR "s"))]
```

TYPES AND DATA DECLARATIONS

```
type V = String {- V = variables -}
type L = String {- L = Labels -}
type N = Int {-N = natural numbers -}
type C = Int {-C = constants -}

type VS = [(V,N)] -- version function
type LS = [(L,(N,[VS],VS))]
```

```haskell
type Ctx = (VS,LS) -- context

type Vsa = (V,N) --  SA variables
type Lsa = (L,N) -- SA labels

type PROG = [(L,B)]
type PROGI = [(L,Bsa)]
type PROGsa = [(L,PHI,Bsa)]
type PHI = [(Vsa,[EXPsa])]

data EXP = VAR V | CONST C | SOMA EXP EXP | MULT EXP EXP | EQU EXP EXP | GE EXP
    EXP | LE EXP EXP
instance Show EXP where
    show (VAR v) = v
    show (CONST c) = show c
    show (SOMA exp exp1) = (show exp) ++ "+" ++ (show exp1)
    show (MULT exp exp1) = (show exp) ++ "*" ++ (show exp1)
    show (EQU exp exp1) = (show exp) ++ "==" ++ (show exp1)
    show (GE exp exp1) = (show exp) ++ "=>" ++ (show exp1)
    show (LE exp exp1) = (show exp) ++ "=<" ++ (show exp1)


data EXPsa = VARsa Vsa | CONSTsa C | SOMAsa EXPsa EXPsa | MULTsa EXPsa EXPsa |
    EQUsa EXPsa EXPsa | GEsa EXPsa EXPsa | LEsa EXPsa EXPsa
instance Show EXPsa where
    show (VARsa v) = "(" ++ fst v ++ "," ++ (show (snd v)) ++ ")"
    show (CONSTsa c) = show c
    show (SOMAsa exp exp1) = (show exp) ++ "+" ++ (show exp1)
    show (MULTsa exp exp1) = (show exp) ++ "*" ++ (show exp1)
    show (EQUsa exp exp1) = (show exp) ++ "==" ++ (show exp1)
    show (GEsa exp exp1) = (show exp) ++ "=>" ++ (show exp1)
    show (LEsa exp exp1) = (show exp) ++ "=<" ++ (show exp1)

data B = SEQ V EXP B | RET EXP | GOTO L | BRANCH EXP L L
instance Show B where
    show (SEQ v exp b) = "     " ++ v ++ ":=" ++ (show exp) ++ "; \n" ++ (show b)
    show (RET exp) = "     return " ++ (show exp) ++ "\n"
    show (GOTO l) = "     goto " ++ l  ++ "\n"
    show (BRANCH exp l1 l2) = "     branch " ++ (show exp) ++ "  " ++ l1 ++ "  "
        ++ l2 ++ "\n"


data Bsa = SEQsa Vsa EXPsa Bsa | RETsa EXPsa | GOTOsa Lsa | BRANCHsa EXPsa Lsa
    Lsa
instance Show Bsa where
    show (SEQsa v exp b) = "     " ++ (show v) ++ ":=" ++ (show exp) ++ "; \n" ++
         (show b)
    show (RETsa exp) = "      return " ++ (show exp)  ++ "\n"
    show (GOTOsa l) = "      goto " ++ (show l) ++ "\n"
    show (BRANCHsa exp l1 l2) = "      branch " ++ (show exp) ++ (show l1) ++ (
        show l2) ++ "\n"
```

AUXILIARY FUNCTIONS

```
union :: (Ord a) => [a] -> [a] -> [a]
union [] [] =[]
union [] (x:xs) =(x:xs)
union (x:xs) [] =(x:xs)
union (x:xs)(y:ys)
                | x<y = x:union xs (y:ys)
                | x==y = x: union xs ys
                | otherwise = y: union (x:xs) ys

vars :: PROG -> [V]
vars [] = []
vars ((l,b):t) = union (varsB b) (vars t)

varsB :: B -> [V]
varsB (SEQ v exp b1)= union (union [v] (varsB b1)) (varsE exp)
varsB (RET exp) = varsE(exp)
varsB (GOTO l) = []
varsB (BRANCH exp l1 l2) = varsE(exp)


varsE :: EXP -> [V]
varsE (CONST c) = []
varsE (VAR v) = [v]
varsE (SOMA exp exp1) = union (varsE exp) (varsE exp1)
varsE (MULT exp exp1) = union (varsE exp) (varsE exp1)
varsE (EQU exp exp1) = union (varsE exp) (varsE exp1)
varsE (GE exp exp1) = union (varsE exp) (varsE exp1)
varsE (LE exp exp1) = union (varsE exp) (varsE exp1)

labels :: PROG -> [L]
labels  ((l,b):t) = union [l] (labels t)
labels [] = []

vsInit :: PROG -> VS
vsInit p = [(x,-1) | x<-(vars p)]
```

```
lsInit :: PROG -> LS
lsInit p = [(l,(0,[],vs0)) | l<-(labels p)]
    where vs0 = vsInit p

initC :: PROG -> Ctx
initC p = (vs0,ls0)
    where
        vs0 = vsInit p
        ls0 = lsInit p

inc :: VS -> VS
inc [] = []
inc vs = [(x,n+1) | (x,n)<-vs]

hatE :: VS -> EXP -> EXPsa
hatE vs (CONST c) = CONSTsa c
hatE vs (VAR x) = VARsa (hatV vs x)
hatE vs (SOMA exp exp1) = let
                        y = hatE vs exp
                        m = hatE vs exp1
                        in SOMAsa y m
hatE vs (MULT exp exp1) = let
                        y = hatE vs exp
                        m = hatE vs exp1
                        in MULTsa y m
hatE vs (EQU exp exp1) = let
                        y = hatE vs exp
                        m = hatE vs exp1
                        in EQUsa y m
hatE vs (GE exp exp1) = let
                        y = hatE vs exp
                        m = hatE vs exp1
                        in GEsa y m
hatE vs (LE exp exp1) = let
                        y = hatE vs exp
                        m = hatE vs exp1
                        in LEsa y m

hatV :: VS -> V -> Vsa
hatV [] x = (x,-1)
hatV ((y,n):t) x = if x==y
                    then (x,n)
                    else hatV t x

consultaV :: VS -> V -> Int
consultaV ((h,x):t) v = if h==v
                            then x
                            else consultaV t v

consultaL :: LS -> L -> (Int,[VS],VS)
consultaL ((h,x):t) v = if h==v
                            then x
                            else consultaL t v
consultaL [] _ = (0,[],[])

nextV :: VS -> V -> VS
nextV ((y,n):t) x = if x==y
                    then (y,n+1):t
                    else (y,n):nextV t x

updateL :: LS -> L -> (Int,[VS],VS) -> LS
updateL ((l,(n1,d,w)):t) l1 (n,lvs,vs) = if l1==l
                                        then (l,(n,lvs,vs)):t
                                        else (l,(n1,d,w)):updateL t l1 (n,lvs,vs)
updateL [] _ _ = []
```

MAIN FUNCTIONS: $\mathcal{R}$ (RENAME), T_L , T_B , sync , $\mathcal{S}$ (SL), $\mathcal{T}$ (FINAL)

```
rename :: PROG -> (PROGI, Ctx)
rename p = tl p (initC p)


tl :: PROG -> Ctx -> (PROGI, Ctx)
tl ((l,b):t) (vs,ls) = (((l,b'):t'), c'')
                        where
                            (n,d,_) = consultaL ls l
                            lv = updateL ls l (n,d,inc vs)
                            (b',c') = tb (b,l, inc vs) (inc vs,lv)
                            (t',c'') = tl t c'
tl [] c = ([], c)

tb :: (B,L,VS) -> Ctx -> (Bsa,Ctx)
tb ((SEQ v exp b1), l, vs0) (vs,ls) = (SEQsa (v,(consultaV vs v)+1) (hatE vs exp)
     b', c')
                            where
                                c = (nextV vs v, ls)
                                (b',c') = tb (b1,l,vs0) c
tb ((GOTO a), l, vs0) (vs,ls) = (GOTOsa (a, n+1), c')
                            where
                                (n,d,w) = consultaL ls a
                                ls’ = updateL ls a (n+1, d++[vs],w)
                                c’ = (vs, ls’)
tb ((RET exp), l, vs0) c@(vs,ls) = (RETsa (hatE vs exp), c)
tb ((BRANCH exp a b), l, vs0) (vs,ls) = (BRANCHsa (hatE vs exp) (a, n+1) (b, n
    ’+1), c)
                            where
                                (n,d,w) = consultaL ls a
                                l’ = updateL ls a (n+1, d++[vs], w)
                                (n’,d’,w’) = consultaL l’ b
                                l’’ = updateL l’ b (n’+1, d++[vs], w’)
                                c = (vs,l’’)

sync :: L -> Ctx -> PHI
sync l (vs,ls) = [ ((x,consultaV w x) , [VARsa (x, consultaV vs0 x) | vs0 <- d] )
     | x <- dom w ]
                where
                    (_,d,w) = consultaL ls l

sl :: (PROGI,Ctx) -> [(L,PHI,Bsa)]
sl ([],c)  = []
sl (("bullet",b):t, c) = ("bullet",[],b): sl(t,c)
sl (((l,b):t),c) = (l, sync l c, b): sl(t,c)

final :: PROG -> PROGsa
final p = sl (rename p)
```

EXAMPLE

Input:

```
final suma
(traduction suma)
```

Output:

```
"bullet":
      ("n",1):=10;
      ("i",1):=0;
```

```
     ("s",1):=0;
     goto ("l",1)
"l":
     ("i",2):= phi((i,1),(i,4))
     ("n",2):= phi((n,1),(n,3))
     ("s",2):= phi((s,1),(s,4))
     branch (i,2)=<(n,2)("l’",1)("l’’",1)
"l’":
     ("i",3):= phi((i,2))
     ("n",3):= phi((n,2))
     ("s",3):= phi((s,2))
     ("s",4):=(s,3)+(i,3);
     ("i",4):=(i,3)+1;
     goto ("l",2)
"l’’":
     ("i",5):= phi((i,2))
     ("n",4):= phi((n,2))
     ("s",5):= phi((s,2))
     return (s,5)
```

B

APPENDIX

EXAMPLES

```
suma :: PROG
suma = [("bullet", (SEQ "n" (CONST 10) (SEQ "i" (CONST 0) (SEQ "s"
   (CONST 0) (GOTO "l"))))), ("l", BRANCH (LE (VAR "i") (VAR "n"))
   ("l′") ("l′′")), ("l′", (SEQ "s" (SOMA (VAR "s") (VAR "i")) (SEQ
    "i" (SOMA (VAR "i") (CONST 1) ) (GOTO "l")))), ("l′′", RET (VAR
    "s"))]

op :: OP
op = [("bullet",[]),("l",["i","s"]),("l′",[]),("l′′",[])]
```

TYPES AND DATA DECLARATIONS

```
type V = String {- V = variables -}
type L = String {- L = Labels -}
type N = Int {-N = natural numbers -}
type C = Int {-C = constants -}

type VS = [(V,N)] -- version function
type LS = [(L,(N,[VS],VS))]
type Ctx = (VS,LS) -- context

type Vsa = (V,N) --  SA variables
type Lsa = (L,N) -- SA labels
type PROG = [(L,B)]
type PROGI = [(L,Bsa)]
type PROGsa = [(L,PHI,Bsa)]
type PHI = [(Vsa,[EXPsa])]
type OP = [(L,[V])]

data EXP = VAR V | CONST C | SOMA EXP EXP | MULT EXP EXP | EQU EXP
   EXP | GE EXP EXP | LE EXP EXP
instance Show EXP where
    show (VAR v) = v
    show (CONST c) = show c
    show (SOMA exp exp1) = (show exp) ++ "+" ++ (show exp1)
    show (MULT exp exp1) = (show exp) ++ "*" ++ (show exp1)
    show (EQU exp exp1) = (show exp) ++ "==" ++ (show exp1)
    show (GE exp exp1) = (show exp) ++ "=>" ++ (show exp1)
    show (LE exp exp1) = (show exp) ++ "=<" ++ (show exp1)



data EXPsa = VARsa Vsa | CONSTsa C | SOMAsa EXPsa EXPsa | MULTsa
   EXPsa EXPsa | EQUsa EXPsa EXPsa | GEsa EXPsa EXPsa | LEsa EXPsa
   EXPsa
instance Show EXPsa where
    show (VARsa v) = "(" ++ fst v ++ "," ++ (show (snd v)) ++ ")"
    show (CONSTsa c) = show c
    show (SOMAsa exp exp1) = (show exp) ++ "+" ++ (show exp1)
    show (MULTsa exp exp1) = (show exp) ++ "*" ++ (show exp1)
    show (EQUsa exp exp1) = (show exp) ++ "==" ++ (show exp1)
    show (GEsa exp exp1) = (show exp) ++ "=>" ++ (show exp1)
    show (LEsa exp exp1) = (show exp) ++ "=<" ++ (show exp1)

data B = SEQ V EXP B | RET EXP | GOTO L | BRANCH EXP L L
instance Show B where
    show (SEQ v exp b) = "     " ++ v ++ ":=" ++ (show exp) ++ "; \
       n" ++ (show b)
    show (RET exp) = "     return " ++ (show exp) ++ "\n"
    show (GOTO l) = "     goto " ++ l  ++ "\n"
    show (BRANCH exp l1 l2) = "     branch " ++ (show exp) ++ "  "
       ++ l1 ++ "  " ++ l2 ++ "\n"


data Bsa = SEQsa Vsa EXPsa Bsa | RETsa EXPsa | GOTOsa Lsa |
   BRANCHsa EXPsa Lsa Lsa
instance Show Bsa where
    show (SEQsa v exp b) = "      " ++ (show v) ++ ":=" ++ (show exp
       ) ++ "; \n" ++ (show b)
    show (RETsa exp) = "     return " ++ (show exp)  ++ "\n"
    show (GOTOsa l) = "     goto " ++ (show l) ++ "\n"
    show (BRANCHsa exp l1 l2) = "     branch " ++ (show exp) ++ (
       show l1) ++ (show l2) ++ "\n"
```

AUXILIARY FUNCTIONS

```
union :: (Ord a) => [a] -> [a] -> [a]
union [] [] =[]
union [] (x:xs) =(x:xs)
union (x:xs) [] =(x:xs)
union (x:xs)(y:ys)
                | x<y = x:union xs (y:ys)
                | x==y = x: union xs ys
                | otherwise = y: union (x:xs) ys

vars :: PROG -> [V]
vars [] = []
vars ((l,b):t) = union (varsB b) (vars t)

varsB :: B -> [V]
varsB (SEQ v exp b1)= union (union [v] (varsB b1)) (varsE exp)
varsB (RET exp) = varsE(exp)
varsB (GOTO l) = []
varsB (BRANCH exp l1 l2) = varsE(exp)
```

```
varsE :: EXP -> [V]
varsE (CONST c) = []
varsE (VAR v) = [v]
varsE (SOMA exp exp1) = union (varsE exp) (varsE exp1)
varsE (MULT exp exp1) = union (varsE exp) (varsE exp1)
varsE (EQU exp exp1) = union (varsE exp) (varsE exp1)
varsE (GE exp exp1) = union (varsE exp) (varsE exp1)
varsE (LE exp exp1) = union (varsE exp) (varsE exp1)

labels :: PROG -> [L]
labels  ((l,b):t) = union [l] (labels t)
labels [] = []

vsInit :: PROG -> VS
vsInit p = [(x,-1) | x<-(vars p)]

lsInit :: PROG -> LS
lsInit p = [(l,(0,[],vs0)) | l<-(labels p)]
    where vs0 = vsInit p

initC :: PROG -> Ctx
initC p = (vs0,ls0)
    where
        vs0 = vsInit p
        ls0 = lsInit p

inc :: VS -> VS
inc [] = []
inc vs = [(x,n+1) | (x,n)<-vs]

hatE :: VS -> EXP -> EXPsa
hatE vs (CONST c) = CONSTsa c
hatE vs (VAR x) = VARsa (hatV vs x)
hatE vs (SOMA exp exp1) = let
                        y = hatE vs exp
                        m = hatE vs exp1
                        in SOMAsa y m
```

```
hatE vs (MULT exp exp1) = let
                        y = hatE vs exp
                        m = hatE vs exp1
                        in MULTsa y m
hatE vs (EQU exp exp1) = let
                        y = hatE vs exp
                        m = hatE vs exp1
                        in EQUsa y m
hatE vs (GE exp exp1) = let
                        y = hatE vs exp
                        m = hatE vs exp1
                        in GEsa y m
hatE vs (LE exp exp1) = let
                        y = hatE vs exp
                        m = hatE vs exp1
                        in LEsa y m

hatV :: VS -> V -> Vsa
hatV [] x = (x,-1)
hatV ((y,n):t) x = if x==y
                    then (x,n)
                    else hatV t x

consultaV :: VS -> V -> Int
consultaV ((h,x):t) v = if h==v
                            then x
                            else consultaV t v

consultaL :: LS -> L -> (Int,[VS],VS)
consultaL ((h,x):t) v = if h==v
                            then x
                            else consultaL t v
consultaL [] _ = (0,[],[])

nextV :: VS -> V -> VS
nextV ((y,n):t) x = if x==y
                    then (y,n+1):t
                    else (y,n):nextV t x
```

```
updateL :: LS -> L -> (Int,[VS],VS) -> LS
updateL ((l,(n1,d,w)):t) l1 (n,lvs,vs) = if l1==l
                                         then (l,(n,lvs,vs)):t
                                         else (l,(n1,d,w)):updateL t
                                             l1 (n,lvs,vs)
updateL [] _ _ = []

sobre :: VS -> VS -> VS
sobre ((x,y):t) ((x1,y1):t1) = if (x==x1)
                                 then (x,y1):(sobre t t1)
                                 else (x,y):(sobre t t1)
sobre vs [] = vs
sobre [] vs = []

consultaOP :: OP -> L -> [V]
consultaOP ((l,v):t) l1 = if (l==l1)
                         then v
                         else consultaOP t l1

incVersion :: VS -> [V] -> VS
incVersion vs@((x,n):t) (x1:t1) = if (x==x1)
                             then (x,n+1):incVersion vs t1
                             else incVersion vs t1
incVersion vs [] = []
incVersion [] v = []
```

MAIN FUNCTIONS: $\mathcal{R}$ (RENAME), T_{L} , T_{B} , sync , $\mathcal{S}$ (SL), $\mathcal{T}$ (FINAL)

```
rename :: PROG -> OP -> (PROGI, Ctx)
rename p o = tl p o (initC p)

tl :: PROG -> OP -> Ctx -> (PROGI, Ctx)
tl ((l,b):t) o (vs,ls) = (((l,b'):t'), c'')
                         where
```

```
                            (n,d,_) = consultaL ls l
                            vs1 = sobre vs (incVersion vs (
                               consultaOP o l))
                            lv = updateL ls l (n,d,(map (hatV (inc
                               vs)) (consultaOP o l)))
                            (b',c') = tb (b,l) (vs1,lv)
                            (t',c'') = tl t o c'
tl [] o c = ([], c)

tb :: (B,L,VS) -> Ctx -> (Bsa,Ctx)
tb ((SEQ v exp b1), l, vs0) (vs,ls) = (SEQsa (v,(consultaV vs v)+1)
    (hatE vs exp) b', c')
                            where
                                c = (nextV vs v, ls)
                                (b',c') = tb (b1,l,vs0) c
tb ((GOTO a), l, vs0) (vs,ls) = (GOTOsa (a, n+1), c')
                            where
                                (n,d,w) = consultaL ls a
                                ls' = updateL ls a (n+1, d++[vs],w)
                                c' = (vs, ls')
tb ((RET exp), l, vs0) c@(vs,ls) = (RETsa (hatE vs exp), c)
tb ((BRANCH exp a b), l, vs0) (vs,ls) = (BRANCHsa (hatE vs exp) (a,
    n+1) (b, n'+1), c)
                            where
                                (n,d,w) = consultaL ls a
                                l' = updateL ls a (n+1, d++[vs], w)
                                (n',d',w') = consultaL l' b
                                l'' = updateL l' b (n'+1, d++[vs],
                                   w')
                                c = (vs,l'')


sync :: L -> Ctx -> PHI
sync l (vs,ls) = [ ((x,consultaV w x) , [VARsa (x, consultaV vs0 x)
    | vs0 <- d] ) | x <- dom w ]
                where
                    (_,d,w) = consultaL ls l

sl :: (PROGI,Ctx) -> [(L,PHI,Bsa)]
sl ([],c)  = []
sl (("bullet",b):t, c) = ("bullet",[],b): sl(t,c)
sl (((l,b):t),c) = (l, sync l c, b): sl(t,c)

final :: (PROG,OP) -> PROGsa
final (p,o) = sl (rename p o)
```

EXAMPLE

Input:

```
final suma op
(traduction suma op)
```

Output:

```
"bullet":
     ("n",1):=10;
     ("i",1):=0;
     ("s",1):=0;
     goto ("l",1)
"l":
     ("i",2):= phi((i,1),(i,3))
     ("s",2):= phi((s,1),(s,2))
     branch (i,2)=<(n,1)("l'",1)("l''",1)
"l'":
     ("s",2):=(s,1)+(i,2);
     ("i",3):=(i,2)+1;
     goto ("l",2)
"l''":
     return (s,2)
```