Radboud University Nijmegen

Faculty of Social Science

Structural operational semantics of imperative programming languages in Coq

Thesis BSc Artificial Intelligence

Author:
Elitsa Bahovska
s1001758

Supervisors:

dr. Engelbert Hubbers
Digital Security

dr. Freek Wiedijk
Software Science

July 10, 2020

Abstract

This thesis proposes a design for a framework to aid the study of the course Semantics and Correctness from the Faculty of Science of Radboud University. More specifically, it covers the subject of Structural Operational - small step - Semantics, explaining what it is, and how is it applied, and shows the implementation *implementation* of a framework in **Coq** for a simple programming language - **While**. Then, some examples from the course are made, using this framework, and properties of Structural Operational Semantics are proven. That includes formalising the rules of Structural Operational Semantics, showing how those are used to prove some general examples, strong progress, determinism, essential exercises, semantic equivalence and equivalence with Natural Semantics.

Contents

Chapter 1

Introduction

“Formal semantics is concerned with rigorously specifying the meaning, or behaviour, of programs, pieces of hardware etc”, as generalised in “Semantics with Applications, A Formal Introduction” by Nielson and Nielson[4]. We need it to avoid ambiguities and unclear descriptions in documentation (of programming languages for example) and as a basis of implementation, analysis and verification. With fewer words, we need formal semantics to make proofs of correctness.

There are three major types of semantics - operational, denotational and axiomatic semantics. Radboud University Faculty of Science’s course Semantics and Correctness discusses, or at least mentions, all three of them and the relations between them. The main one studied, that provides basic knowledge to discuss the other two is operational semantics. It can be split in two types - Natural Semantics, also known as big-step, and Structural Operational Semantics also know as small-step semantics. This thesis focuses on the latter while the one written by Loes Kruger [3] in parallel is centred around the former. Some parts, that are not sub-topic dependent and are relevant to both theses are common in the two texts and made in collaboration.

1.1 Problem outline

Some students, who follow the course Semantics and Correctness find the proofs with operational semantics difficult and that affects their understanding of the course. Some proofs in Semantics and Correctness use several levels of induction, which, along with some left out details from the accompanying literature on Semantics and Correctness [4] makes mastering the material a difficult task.[2].

A comparison can be made with another course from RU Faculty of Science - Logic and Applications - that most students taking Semantics and Correctness have already participated in. Along all the study materials there, it includes an introduction to the functional language and proof assistant **Coq**, which seems to aid the learning process significantly.

A privately conveyed survey amongst a group of students shows however, that the work in **Coq** alone was not enough to ease their studying process - it had to be combined with detailed explanation of both the theory and how the examples in **Coq** are working.

1.2 Related work

Relevant work and similar formalisation in **Coq** already exists in “Software Foundations” by Pierce [6] and by Schirmer [8] and Nipkow and Klein [5] in Isabelle etc. While very helpful, those books are not tailored specifically for the needs of Semantics and Correctness: be it different notations, examples, programming languages, formalisation decisions, or that some of those works do not aim to be textbooks and are directed to more experienced readers.

1.3 Goals

Based on the above conditions, there is a place in Semantics and Correctness for additional assistance to the learning process. This thesis presents one option that covers the Structural Operational Semantics part of the course.

This thesis's goal is to build a framework for applying Structural Operational Semantics rules in **Coq**. That framework, along with a comprehensive guide to both using it and applications of Structural Operational Semantics would ultimately act as a guide to Structural Operational Semantics for the Semantics and Correctness student.

The explanations here aim to accompany the theory of Structural Operational Semantics, a step by step depiction of its formalisation in **Coq**, as well as all the examples. Effort is made to find examples for tricky and/or essential cases of application of Structural Operational Semantics (like using the rule $[\text{comp}^1_{\text{sos}}]$), proofs via induction on the length of the derivation sequence, as well as to address all the basic properties of Structural Operational Semantics and its equivalence with Natural Semantics. In the last chapter of this thesis, the response of students, who tried to use this aid can be found.

The thesis is going to try and answer the following research question: **Is it possible to formalise Structural Operational Semantics for While in Coq in a way that can help study it in the context of the course Semantics and Correctness?**

1.4 Approach

The approach to solving this problem is to make it easier to prove that executing a statement results in what we expected, using Structural Operational Semantics. That entails creating proofs in Structural Operational Semantics and illustrating the steps made to complete them. That will be achieved by creating and using a framework in **Coq** accompanied by a comprehensive guide to it. This would take the following steps:

Theory
The first step of our comprehensive explanations is to cover all the relevant theory in an understandable manner:

SOS Introduction to our type of semantics in theory.

While Explaining what this simple language entails and how are we going to use it.

Coq Explaining how we use the proof assistant.

Expressions
Before we start formalising the statements in **While**, we need to be able to evaluate expressions.

Complete execution
In SOS a statement can either be, or not be completely executed - illustrated e.g. in the difference between $[\text{comp}^1_{\text{sos}}]$ and $[\text{comp}^2_{\text{sos}}]$. We need to handle this problem - by making a special case of skip, special datatype or another method.

Basic setup
The basic rules of Structural Operational Semantics need to be defined : $[\text{ass}_{\text{sos}}]$, $[\text{skip}_{\text{sos}}]$, $[\text{if}^{\text{tt}}_{\text{sos}}]$, $[\text{if}^{\text{ff}}_{\text{sos}}]$, $[\text{comp}^1_{\text{sos}}]$, $[\text{comp}^2_{\text{sos}}]$, $[\text{while}_{\text{sos}}]$.

Properties
Once we have the rules, we use them to prove a number of Structural Operational Semantics properties. Those can be either in one step - like determinism and strong progress; or in many steps, that include induction on the length of the derivation sequence.

Equivalence with Natural Semantics
Many of the applications of Operational Semantics, such as semantic equivalence, proving soundness and completeness etc. are considerably more difficult to prove using Structural Operational Semantics than Natural Semantics. Hence, those are proven in the thesis focused on Natural Semantics by Loes Kruger [3], and here we will prove that Structural Operational Semantics is equivalent to Natural Semantics.

Experiment
Finally, when the formalisation and the comprehensive explanations are done, their effect on student's learning process will be depicted.

1.5 Contribution and work in Coq

This thesis depicts a formalisation of Structural Operational Semantics custom tailored for Semantics and Correctness, that differentiates from other work by following closely the notations and study flow of the course and uses a language, that is already familiar to most of the students that follow it. Together with Loes Kruger's [3] formalisation of Natural Semantics, there is coverage on most of the Semantics and Correctness topics.

The finished formalisations in **Coq** are available on GitHub to read and use with citation. They can be found on this link: `https://github.com/lkruger27/NS_SOS_Formalization`.

Chapter 2

Background

Before we get to the formalisation of Structural Operational Semantics, we will discuss some of the basic concepts that we will base the formalisation on: the language **While**, **Coq** and Structural Operational Semantics.

As **While** and **Coq** are common for considering semantics, they are also discussed in the Bachelor Thesis of Loes Kruger [3], written at the same time as this one. Section 2.2 **While** and section 2.3 **Coq** are adapted from there.

2.1 Basic notes

Here we will mention the most basic concepts most readers are probably familiar with.

2.1.1 Syntax vs Semantics

When we talk of syntax, we are considering what is written in text and notation. For example, the ordered collection of symbols `Add(x,y)` could be a language's syntax for addition.

When we talk of the semantics of something, we consider its meaning. For `Add(x,y)` that would be the reasoning, that when we see that syntax, we expect the computation behind it to be the addition of `x` and `y`. We could depict it as `(x + y) = z`, but that would be another syntax. What matters, when we are talking about semantics in this example is the whether the value of `z` is indeed the sum of of `x` and `y`. More general, the semantics is an operation to do what it is supposed to do.

That is why Semantics and Correctness is so important - we can write code as much as we want, but we need to be able to prove that it indeed does what is expected.

2.2 While

While is a simple imperative programming language, that only has a minimal number of structures, embodying a tiny core fragment of conventional languages. It is quite similar to **Imp**, described in “Software Foundations” [6]. Because of its simplicity, it is a good example language for Semantics and Correctness, to write provable programs on it and focus on the proofs. Its syntax can be fully defined by a context-free grammar with five non-terminals - here called meta variables - and their constructors, that can be seen in Figure 2.1, and will be described in detail in the following sections.

n will range over numerals **Num**
x will range over variables **Var**
a will range over arithmetic expressions **Aexp**
b will range over boolean expressions **Bexp**
S will range over statements **Stm**

Figure 2.1: The syntactic categories and their meta-variables

The constructs specify how to build numerals, variables, arithmetic expressions, boolean expression and statements. The structure of the constructs can be found in Figure 2.2.

$$\begin{array}{rcl}
n & ::= & n \;\texttt{::=}\; \texttt{0} \mid \texttt{1} \mid n\ \texttt{0} \mid n\ \texttt{1} \\
x & ::= & \text{strings of letters and digits starting with a letter} \\
a & ::= & n \;\texttt{|}\; x \;\texttt{|}\; a_1 \;\texttt{+}\; a_2 \;\texttt{|}\; a_1 \star a_2 \;\texttt{|}\; a_1 \;\texttt{-}\; a_2 \\
b & ::= & \texttt{true} \;\texttt{|}\; \texttt{false} \;\texttt{|}\; a_1 \;\texttt{=}\; a_2 \;\texttt{|}\; a_1 \leq a_2 \;\texttt{|}\; \neg b \;\texttt{|}\; b_1 \wedge b_2 \\
S & ::= & x \;\texttt{:=}\; a \;\texttt{|}\; \texttt{skip} \;\texttt{|}\; S_1\texttt{;}S_2 \;\texttt{|}\; \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \;\texttt{|}\; \texttt{while}\ b\ \texttt{do}\ S
\end{array}$$

Figure 2.2: The structure of the constructs

2.2.1 Num

The syntax definition for **Num** is

$$n ::= \texttt{0} \mid \texttt{1} \mid n\ \texttt{0} \mid n\ \texttt{1}$$

The function $\mathcal{N} : \textbf{Num} \rightarrow \textbf{Z}$ is used to express the value of a numeral n as an integer. The semantics of **Num** are defined in Figure 2.3. This is a nice example of what we explained in subsection 2.1.1 Syntax vs Semantics. Syntactically, a member of the type **Num** is a string of 1-s and 0-s. However, in latter examples, we will use decimal digits to refer to **Num**-s. We can do that, by implicitly converting the binary number, that the definition gives, to what is semantically just a number. That allows us to depict the semantics of a numeral bu using the comfortable syntax of decimals. In other words, because we know that binary and decimal numbers mean the same thing, we can interchange the syntax between them.

$$\begin{aligned}
\mathcal{N}[\![\mathtt{0}]\!]s &= 0\\
\mathcal{N}[\![\mathtt{1}]\!]s &= 1\\
\mathcal{N}[\![n\mathtt{0}]\!]s &= 2\cdot\mathcal{N}[\![n]\!]\\
\mathcal{N}[\![n\mathtt{1}]\!]s &= 2\cdot\mathcal{N}[\![n]\!]+1
\end{aligned}$$

Figure 2.3: The semantics of numerals

2.2.2 Var

A variable is a string that specifies a place to be filled by a value. This value can change at different steps of execution and hence be different in different states of the program. So, here is the moment to define **State**-s.

2.2.3 State

It is no coincidence that the notion **State** is used not only here, but also in the definition of an automaton. A state of a machine - or of the execution of a program - is a set of values for all the (relevant / defined) variables. As we are not considering non-determinism here, that would mean that each variable has exactly one value in a give state. On top of that, a state is completely defined by the values for all the variables. Hence the value of the variable is not necessary equal in every step of the program.

We can define a state as a function $s : \mathbf{Var} \rightarrow \mathbf{Z}$ which maps a variable to its value at the current moment.

2.2.4 Aexp

The syntax definition for **Aexp** is

$$a \ ::= \ n \ \mid \ x \ \mid \ a_1 \ \mathtt{+} \ a_2 \ \mid \ a_1 \ \star \ a_2 \ \mid \ a_1 \ \mathtt{-} \ a_2$$

To define a semantic function that expresses the value of a numeral or variable as an integer, we need states, as the evaluation of an expression depends on the values of the variables in it. So, how is **Aexp** used?

Here is the moment to give a definition for the function $\mathcal{A}$. It evaluates an arithmetical expression **Aexp** to its value as a number **Num**. Note, that this evaluation can vary over **State**-s as an **Aexp** can contain variables, whose values may change. The formally defined $\mathcal{A} : \mathbf{Aexp} \rightarrow (\mathbf{State} \rightarrow \mathbf{Z})$ is the function used to express the value of an arithmetic expression as an integer. The semantics of **Aexp** is defined in Figure 2.4.

$$
\begin{array}{rcl}
\mathcal{A}[\![n]\!]s & = & \mathcal{N}[\![n]\!] \\
\mathcal{A}[\![x]\!]s & = & s\ x \\
\mathcal{A}[\![a_1 \texttt{ + } a_2]\!]s & = & \mathcal{A}[\![a_1]\!]s + \mathcal{A}[\![a_2]\!]s \\
\mathcal{A}[\![a_1 \star a_2]\!]s & = & \mathcal{A}[\![a_1]\!]s \cdot \mathcal{A}[\![a_2]\!]s \\
\mathcal{A}[\![a_1 \texttt{ - } a_2]\!]s & = & \mathcal{A}[\![a_1]\!]s - \mathcal{A}[\![a_2]\!]s
\end{array}
$$

Figure 2.4: The semantics of arithmetic expressions

2.2.5 Bexp

The syntax definition for **Bexp** is

$$b \ ::= \texttt{true} \ | \ \texttt{false} \ | \ a_1 \texttt{ = } a_2 \ | \ a_1 \le a_2 \ | \ \neg b \ | \ b_1 \land b_2$$

The function $\mathcal{B}$: **Bexp** → (**State** → **T**) expresses the value of a boolean expression as **T**, where $\mathbf{T} = \{\mathbf{tt}, \mathbf{ff}\}$. The semantics of **Bexp** is defined in Figure 2.5.

$$
\begin{array}{rcl}
\mathcal{B}[\![\texttt{true}]\!]s & = & \mathbf{tt} \\
\mathcal{B}[\![\texttt{false}]\!]s & = & \mathbf{ff} \\
\mathcal{B}[\![a_1 \texttt{ = } a_2]\!]s & = & \begin{cases} \mathbf{tt} & \text{if } \mathcal{A}[\![a_1]\!]s = \mathcal{A}[\![a_2]\!]s \\ \mathbf{ff} & \text{if } \mathcal{A}[\![a_1]\!]s \neq \mathcal{A}[\![a_2]\!]s \end{cases} \\
\mathcal{B}[\![a_1 \le a_2]\!]s & = & \begin{cases} \mathbf{tt} & \text{if } \mathcal{A}[\![a_1]\!]s \le \mathcal{A}[\![a_2]\!]s \\ \mathbf{ff} & \text{if } \mathcal{A}[\![a_1]\!]s > \mathcal{A}[\![a_2]\!]s \end{cases} \\
\mathcal{B}[\![\neg b]\!]s & = & \begin{cases} \mathbf{tt} & \text{if } \mathcal{B}[\![b]\!]s = \mathbf{ff} \\ \mathbf{ff} & \text{if } \mathcal{B}[\![b]\!]s = \mathbf{tt} \end{cases} \\
\mathcal{B}[\![b_1 \land b_2]\!]s & = & \begin{cases} \mathbf{tt} & \text{if } \mathcal{B}[\![b_1]\!]s = \mathbf{tt} \text{ and } \mathcal{B}[\![b_2]\!]s = \mathbf{tt} \\ \mathbf{ff} & \text{if } \mathcal{B}[\![b_1]\!]s = \mathbf{ff} \text{ or } \mathcal{B}[\![b_2]\!]s = \mathbf{ff} \end{cases}
\end{array}
$$

Figure 2.5: The semantics of boolean expressions

2.2.6 Stm

A statement **Stm** can be considered as a command that tells the program what to do in imperative languages. Some more complicated programming languages might have more advanced structures, but **While** has only the essentials, defined as:

$$S \ ::= x \texttt{ := } a \ | \ \texttt{skip} \ | \ S_1\texttt{;}S_2 \ | \ \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \ | \ \texttt{while } b \texttt{ do } S$$

Those are assignment, skipping, composition, case distinction depending on a boolean expression and a loop that finished depending on a boolean expression respectively. However, their semantics is dependent on whether we are in the context of Structural Operational

Semantics or Natural Semantics. In this thesis, as we mentioned before, we are going to consider them in the light of Structural Operational Semantics, and this will be discussed in chapter 3 Structural Operational Semantics.

2.3 Coq

As described in its manual, "**Coq** is a formal proof management system. It provides a formal language to write mathematical definitions, executable algorithms and theorems together with an environment for semi-interactive development of machine-checked proofs. Typical applications include the certification of properties of programming language(...), the formalisation of mathematics (...), and teaching. " [1]

Here, we will describe the **Coq** tactics that will be used most often in the formalisation that follows.

2.3.1 Induction

As per "Software Foundations", Volume 1, Chapter "More Basic Tactics" [6], "`induction ... as...` (is) induction on values of inductively defined types". Let us clarify a bit on that.

Induction ca be applied on an arbitrary member of an inductive set.

In the case the inductive set is the one of natural numbers, induction, be it strong or not, is used to do proof by induction that as we know it. That would split the goal in two subgoals. The first one is for the natural number being zero - the base case. In the second subgoal, we have the hypothesis that the property we want to proof holds for an arbitrary number k (or, in the case of strong induction for all numbers smaller or equal than k), and the goal is to prove it of $k + 1$, or, in **Coq** notation - `S k` (the successor of `k`). That happens, because the inductive set of natural numbers is defined with two constructors: the zero-constructor, that claims that zero is a member of the natural numbers, and the step-constructor, that claims that if a number is a member of the natural numbers, then the successor of that number is also a natural number.

Alternatively, we can use induction on any other inductive set. That is, if we have a finite set of constructors, like the statements in **While**, and in the assumptions, we have an arbitrary member of that set, say `S` the command `induction S` will make as many subgoals as are in the set, where for each subgoal, the set member will be a specific one, until what we need to prove is proven for each member. Therefore `induction` is also used for case distinction on all the possible values of a member of the set. For more information you can also check As per "Software Foundations", Volume 1, Chapter "Proof by Induction" [6]

2.3.2 Inversion

In the following proofs we will use the tactic `inversion` very often. This tactic is applied on a hypothesis - e.g. `inversion H` and can be translated in natural language as "Because we know `H` is true, then it must be the case, that H0,...,HK are true", where `H0,...,HK` are properties, from which `H` follows. In other words, `inversion` adds the things that we can derive from `H` being true to the context.

A very simple example is the case where we have the hypothesis that `A /\ B` and apply `inversion` on it, that will add to the context, as separate hypotheses `A` and `B`.

An alternative example can be made in the context of Structural Operational Semantics. If in the context we have the hypothesis $\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s\rangle \Rightarrow \langle S_1, s\rangle$, introduced in section 3.2 Structural operational semantics for **While**, and apply `inversion` on it, it will be added to our context that the boolean expression b evalates to `true`, because it is the only way this transition would be possible.

2.3.3 `eapply`

In the formalisation that follows, there will be many instances of applying tactics where **Coq** will have to guess a variable. In some simple cases, it will manage, but in some more complicated ones - it will not be able to infer the variables and instead, we will need to use `apply ... with ...` and write it out ourselves. This will be annoying, both as that variable might be quite long to write, and as more often than not, the exact next thing we are going to - applying the next tactic - will tell us what the varable is. As Pierce explains it in "Software Foundations" [6], we can avoid all that by using `eapply` instead of `apply`. We will "tell **Coq**, essentially, 'Be patient: The missing part is going to be filled in later in the proof'. (...) In general, the `eapply H` tactic works just like `apply H` except that, instead of failing if unifying the goal with the conclusion of H does not determine how to instantiate all of the variables appearing in the premises of `H`, `eapply H` will replace these variables with existential variables (written `?nnn`), which functions as placeholders for expressions that will be determined (by further unification) later in the proof." [6].

2.4 Extra tactics

Here we will mention some of the other tactic we are going to use more often in the proofs in **Coq** that will follow.

- `intros`: move hypotheses/variables from goal to context.
- `reflexivity`: finish the proof (when the goal looks like e = e).
- `apply`: prove goal using a hypothesis, lemma, or constructor.
- `apply... in H`: apply a hypothesis, lemma, or constructor to a hypothesis in the context (forward reasoning).

- `apply... with...`: explicitly specify values for variables that cannot be determined by pattern matching.
- `simpl`: simplify computations in the goal.
- `subst`: substitute in the goal and context, according to equivalences present in the context.
- `rewrite`: use an equality hypothesis (or lemma) to rewrite the goal.
- `assert (e)`: introduce a ”local lemma” e.

These are all taken from “Software Foundations”, Volume 1, Chapter “More Basic Tactics” [6], which is a great read for beginner users of **Coq**.

Chapter 3

Structural Operational Semantics

3.1 Definition

Structural operational semantics - (SOS) is all about small, individual steps in the execution. As per the "Semantics with Applications, A Formal Introduction" by Nielson and Nielson [4], the transitions have the form:

$$\langle S, s \rangle \Rightarrow \gamma$$

where γ either is of the form $\langle S', s' \rangle$ or of the form s'. The transition expresses the first step of the execution of a statement S from state s. There are two possible outcomes:

- If γ is of the form $\langle S', s' \rangle$ then the execution of S from s is not completed and the remaining computation starts from the intermediate configuration $\langle S', s' \rangle$. In other words, there is another step (or steps) that needs to be executed before the statement is complete.
- If γ is of the form s' then the execution of S from s has terminated and the final state is s'. That is a terminal configuration.

3.2 Structural operational semantics for While

We cannot just define semantics without using some kind of syntax. In the previous chapter, we defined the syntax of **While**. Now it is time to define the rules for what do **While** statements mean, depicted in Figure 3.1. The rules for **While** are deterministic [4], as we will prove later.

$$[\text{ass}_{\text{sos}}] \qquad \langle x := a, s\rangle \Rightarrow s[x \mapsto \mathcal{A}[\![a]\!]s]$$

$$[\text{skip}_{\text{sos}}] \qquad \langle \texttt{skip}, s\rangle \Rightarrow s$$

$$[\text{comp}^{1}_{\text{sos}}] \qquad \frac{\langle S_1, s\rangle \Rightarrow \langle S_1', s'\rangle}{\langle S_1\texttt{;}\ S_2, s\rangle \Rightarrow \langle S_1'\texttt{;}\ S_2, s'\rangle}$$

$$[\text{comp}^{2}_{\text{sos}}] \qquad \frac{\langle S_1, s\rangle \Rightarrow s'}{\langle S_1\texttt{;}\ S_2, s\rangle \Rightarrow \langle S_2, s'\rangle}$$

$$[\text{if}^{\text{tt}}_{\text{sos}}] \qquad \langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s\rangle \Rightarrow \langle S_1, s\rangle \text{ if } \mathcal{B}[\![b]\!]s = \mathbf{tt}$$

$$[\text{if}^{\text{ff}}_{\text{sos}}] \qquad \langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s\rangle \Rightarrow \langle S_2, s\rangle \text{ if } \mathcal{B}[\![b]\!]s = \mathbf{ff}$$

$$[\text{while}_{\text{sos}}] \qquad \langle \texttt{while}\ b\ \texttt{do}\ S, s\rangle \Rightarrow \langle \texttt{if}\ b\ \texttt{then}\ (S\texttt{;}\ \texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else}\ \texttt{skip}, s\rangle$$

Figure 3.1: Structural operational semantics for **While**

We assume that the reader is familiar with the Structural Operational Semantics, as depicted in “Semantics with Applications, A Formal Introduction” by Nielson and Nielson [4]. We are not claiming that all theorems are mentioned and proved here, and will only recall those that are of special interest in the formalisation.

In this thesis, we will focus more on finite derivation sequences, as **Coq** is will give errors if it detects an infinite sequence.

3.2.1 Derivation Sequence

A derivation sequence starting from a configuration $\langle S, s\rangle$ is a sequence $\gamma_0, \gamma_1, \gamma_2...$, where $\gamma_0 = \langle S, s\rangle$ and $\gamma_i \Rightarrow \gamma_{i+1}$ for $i \geq 0$ That sequence can either be finite, finishing in γ_k, where γ_k is either a stuck (described in subsection 3.2.2 Stuck and Progress) or a terminal state (described in section 3.1 Definition) and $k \geq 0$, or infinite.

We write $\gamma \Rightarrow^i \gamma'$ to indicate there are i steps to get from γ to γ' and $\gamma \Rightarrow^* \gamma'$ to indicate that there are zero or more but finite number of steps, to get from γ to γ'. Note that γ' does not need to be a stuck or a terminal configuration, and hence, those notations do not necessarily describe a derivation sequence.

3.2.2 Stuck and Progress

A stuck configuration $\langle S, s\rangle$ is a non terminal one, where there is no γ, such that $\langle S, s\rangle \Rightarrow \gamma$. In other words, a stuck configuration is a pair $\langle S, s\rangle$, where, in the state s, we cannot execute the statement S and get to another state. Thus, there is no rule we can apply to make progress in the sequence.

"Software Foundations" [6] introduces the notion of strong progress. This is a property of the language that every configuration, is either terminal or if it is non-terminal - of the type $\langle S, s\rangle$ - then there will exist γ such that $\langle S, s\rangle \Rightarrow \gamma$.

In other words, each configuration is either terminal or can make progress, stepping to another configuration. So, if the property of strong progress holds for a language, then we cannot get to a stuck configuration applying the rules to execute each statement.

For **While**, as we defined it, the property of strong progress holds.

Abort

We could extend **While** to **While**' with a statement, say "abort" and the matching rule [abort$_{sos}$] that only consists of $\langle abort, s\rangle$. That would mean we have no step that we can make from the non terminal configuration $\langle abort, s\rangle$, and thus that this is a stuck state. Then, it would be easy to prove that there is no strong progress, in our new language **While**'. However, in this thesis, we will work with the original **While** language and its other properties that we can use, without considering the lack of strong progress.

3.2.3 Comp

A few details on the [comp$^1_{\text{sos}}$] and [comp$^2_{\text{sos}}$] rules: They express the fact that to execute $\langle S_1; S_2, s\rangle$, you need to execute S_1 first. Thus, trying to execute S_1, $\langle S_1; S_2, s\rangle$ could result either in $\langle S_2, s'\rangle$, if the entire S_1 was executed, or in $\langle S_1';\ S_2, s'\rangle$, where there would still be steps of S_1 to finish. As expressed in "Semantics with Applications, A Formal Introduction" [4]

- If the execution of S_1 has not been completed we have to complete it before embarking on the execution of S_2.
- If the execution of S_1 has been completed we can start on the execution of S_2.

3.2.4 Induction on the Length of the Derivation Sequence

Once we have made the formalisation, we will consider some examples from the course Semantics and Correctness. Because of that, it is useful to give the general strategy in "Semantics with Applications, A Formal Introduction" [4] for proofs by induction, that we encounter in SOS context:

Induction on the length of the derivation sequence

Base case Prove that the property holds for all derivation sequences of length zero.

General case Prove that the property holds for all other derivation sequences:

IH Assume that the property holds for all derivation sequences of length at most k.

IS Show that it holds for derivation sequences of length $k + 1$.

3.3 Why Structural Operational Semantics?

Natural semantics is useful in many situations, when we want to prove that a program is in the expected state when it terminates. But it is not as helpful, when we are talking about intermediate steps or concurrent programming - latter example given in the chapter Small-Step Semantics of “Software Foundations”, [6]. There execution is not only about mapping an input to an output state. Instead, there are intermediate states that the program passes through, and those states can be observed by concurrently executing code.

Schirmer [8] presents a more general differentiation between big- and small- step semantics: “A big-step semantic executes the whole program at once and relates the initial with the final states. A small-step semantics is a single step relation between configurations of the computation.”

And finally, we will give the summary from “Semantics with Applications, A Formal Introduction” [4]. We mention non-determinism here for completeness's sake, and will not discuss it in detail.

Natural Semantics versus Structural Operational Semantics

- In a natural semantics the execution of the immediate constituents is an *anatomic entity*, so we cannot express interleaving of computations.
- In a natural semantics we cannot distinguish between looping and abnormal termination.
- In a natural semantics non-determinism will suppress looping, if possible.

- In a structural operational semantics looping is reflected by infinite derivation sequences and abnormal termination by finite derivation sequences ending in a stuck configuration.
- In a structural operational semantics we concentrate on the *small steps* of the computation so we can easily express interleaving.
- In a structural operational semantics non-determinism does not suppress looping.

Chapter 4

Formalisation

4.1 Expressions

The course Semantics and Correctness works with a simple programming language, **While**, that we already discussed. Based on the work of Pierce [6] on another simple imperative language - **Imp**, here we will define our rules for **While**. The full files can be found on *GitHub*.

4.1.1 Var

As in all the other programming languages, a variable is a string of characters, that can be assigned a value. We could make a custom type `Var` in **Coq**, but it would just be a coercion with `string` with no additional properties. Therefore, we will directly use `string` for the type **Var**.

4.1.2 Num

Formalising **Num** may look like an easy task at first, but it takes many steps to comfortably apply the definition of numeric values from **While**. It might be tempting to just use `Z`, as more often than not, we will use $\mathcal{N}$ to evaluate a numeral to an integer. Simpler still, we can make the observation that the definition of the numeral as binary string is unassigned and even depict them as `nat`.

However, since we have already started looking into the definition, we can see that this is not possible. The `nat` type in **Coq** is defined inductively with two constructors: `O` for the zero-element and `S (n:nat)` for the successor of a natural number[6].

However, in the definition of **Num** we have four constructors:

```
Inductive Num : Type :=
  | NZero
  | NOne
  | NEven (n : Num)
  | NOdd (n : Num).
```

That entails two constructors for 0 and 1 respectively, and two for n0 and n1, - where n is a member of **Num**- that concatenate 0, respectively 1, to an already defined numeral. We should remind here that this builds a binary number and using one of the latter two constructors raises the order of the number by one. In other words, the latter two constructors multiply the value so far by two and add either zero or one to it, for even and odd numbers respectively. The function $\mathcal{N}$ evaluates those numerals to integers [4], and we define it as follows:

```
Fixpoint Neval (n : Num) : Z :=
  match n with
  | NZero => 0
  | NOne => 1
  | NEven n => 2*(Neval n)
  | NOdd n => 2*(Neval n) + 1
  end.
```

We can even make a coercion, to interpret members of **Num** as integers directly:

```
Coercion Neval: Num >-> Z.
```

Now we have defined the type **Num** and we can convert from it to integers. However, using it is not very comfortable. Similar to the case, described in "Software Foundations" where the definition of nat would require for us to write S(S(S O))) to represent 3 [6], if we want to represent the numerical with value 3, we need to use nested constructors: NOdd NOne. To illustrate how progressively complicated that would be for bigger numbers, 5 would be NOdd (NEven None) and so on. It would be useful to be able to write digits as well to represent numerals. In other words, we need a coercion in the opposite direction:

```
Coercion z_to_num: Z >-> Num.
```

That requires slightly deeper understanding of the type Z in **Coq**. It is defined with three constructors: a zero one - Z0 - and a positive and negative one, which both take another type - positive: Zpos and Zneg. We coerce the zero constructors of Z with the zero constructor of Num and dwell into the definition of positive for the rest.

```
 Fixpoint z_to_num (z:Z) : Num :=
  match z with
   | Z0 => NZero
   | Zpos n => pos_to_num n
   | Zneg n => pos_to_num n
   end.
```

The type `positive` is more useful for our purpose than `nat`. Fortunately, instead of having a definition based on immediate successor, `positive` has constructors for the unit-element - 1 - and constructors for even and odd numbers respectively. That matches perfectly our remaining three constructors and allows us to define:

```
Fixpoint pos_to_num (n:positive): Num:=
match n with
  |xH => NOne
  |xO n' => NEven (pos_to_num n')
  |xI n' => NOdd (pos_to_num n')
  end.
```

Note that in order for those definitions to work, the latter three listings need to be written in **Coq** in reverse order.

Using the above, we can finally write digits, which are going to be interpreted as with integers and they, in turn, can be coerced with numerals. This will make the further examples much easier to read.

4.1.3 State

A small interlude before we continue to **Aexp** and **Bexp**, as both of these expressions use the notion of state.

The structures we have left to define in **While**, are Statements - **Stm**. As we noticed in the description of the rules of SOS, a statement is basically a command that is to be given in a certain **State**. Thus, here, we will consider how we can depict states in **Coq**. Remember, **State** was defined as a map s : **Var** → **Z** in **While**. So, we will first need to define a map, that takes a string - the name of the variable and can returns its matching numeric value, as is done in "Software Foundations" [6].

```
Definition total_map (A : Type) := string -> A.
```

For **Imp**, that would be `nat`. For **While**, the matching numeric value would be of type **Z**. This suggest that we can use digits to depict the value of a state: our state will have to map strings to **Z**.

```
Definition State := total_map Z.
```

Note, that we can also evaluate **Num**-s to **Z**. Since both of those sets can be depicted with digits, it will be easy to assume writing the same non-negative integers for the two.

In a program, we are not going to have all the variables initialised with a value. Thus, we will need a default empty state to start with and develop on. So, we define

```
Definition empty_st := (_ !-> 0).
```

We need to be able to find and update a variable in the state, so we can update the state:

```
 Definition eqb_string (x y : string) : bool :=
  if string_dec x y then true else false.
Definition t_update {A : Type} (m : total_map A)
                    (x : string) (v : A) :=
  fun x’ => if eqb_string x x’ then v else m x’.
```

And finally, we make a nice recursive notation to make new states by adding variables to an empty state.

```
Notation "’_’ ’!->’ v" := (t_empty v)
  (at level 100, right associativity).
Definition empty_state := (_ !-> 0).
Notation "x ’!->’ v ’,’ m" := (t_update m x v)
         (at level 100, v at next level, right associativity).
Notation "x ’!->’ v" := (x!->v,empty_state)
         (at level 100, v at next level, right associativity).
```

4.1.4 Formalisation of Aexp

In this and the next subsections, we will observe the iterative reasoning behind the formalisation of the arithmetical expressions.

Constructor

First, we will consider our initial guess - and it, indeed, is correct for big step evaluation. For most of the thesis, we will work with that and only execute the statements in small steps.

Above, we defined **Aexp** as

$$a ::= \; n \mid x \mid a_1 \texttt{ + } a_2 \mid a_1 \star a_2 \mid a_1 \texttt{ - } a_2$$

This same definition is used in **Imp**. We can write this syntactic definition in Coq using `Inductive Aexp` and set those options as the constructors.

```
Inductive Aexp : Type :=
  | ANum (n : nat)
  | AId (x: string)
  | APlus (a1 a2 : Aexp)
  | AMinus (a1 a2 : Aexp)
  | AMult (a1 a2 : Aexp).
```

Evaluation

Once we have the syntax for the arithmetical expressions, we would want to define what each of those constructors does. For that, we define a `Fixpoint aeval`:

```
Fixpoint aeval (st : state) (a : Aexp) : nat :=
  match a with
  | ANum n => n
  | AId x => st x
  | APlus a1 a2 => (aeval st a1) + (aeval st a2)
  | AMinus a1 a2  => (aeval st a1) - (aeval st a2)
  | AMult a1 a2 => (aeval st a1) * (aeval st a2)
  end.
```

Instead of writing $\mathcal{A}[\![a]\!]s$ as is done in "Semantics with Applications, A Formal Introduction" [4], we write `(aeval st a)` in Coq. To make it more similar to the notation of Semantics and Correctness, we define:

```
Notation " 'A[[' a ']]' st " := (Aeval st a)
        (at level 90, left associativity)(*: while_scope*).
```

4.1.5 Formalisation of Bexp

We do the same iterative reasoning as in the above section, this time for Boolean expressions.

Constructor

We define **Bexp** the same way as we did with **Aexp** in syntax as `Inductive bexp`:

```
Inductive Bexp : Type :=
  | BTrue
  | BFalse
  | BEq (a1 a2 : Aexp)
  | BLe (a1 a2 : Aexp)
  | BNot (b : Bexp)
  | BAnd (b1 b2 : Bexp).
```

Evaluation

And we define its semantics with `Fixpoint Beval`, with respect to the state:

```
Fixpoint Beval (st : State) (b : Bexp) : bool :=
  match b with
  | BTrue      => true
  | BFalse     => false
  | BEq a1 a2   => (Aeval st a1) =? (Aeval st a2)
  | BLe a1 a2   => (Aeval st a1) <=? (Aeval st a2)
  | BNot b1     => negb (Beval st b1)
  | BAnd b1 b2  => andb (Beval st b1) (Beval st b2)
  end.
```

Notation

```
Notation " 'B[[' b ']]' st " := (Beval st b)
        (at level 90, left associativity)(*: while_scope*).
```

4.1.6 Formalisation of Stm

Contructors

Now we are ready to finally define the statements, using all the other parts of **While** we already defined, similarly to “Software Foundations” [6]:

```
Inductive Stm : Type :=
  | ass (x : string) (a : Aexp)
  | skip
  | comp (s1 s2 : Stm)
  | if_ (b : Bexp) (s1 s2 : Stm)
  | while (b : Bexp) (s : Stm).
```

This, however is not very similar to the statements we find in Semantics and Correctness. To make it closer, we use custom notations for the statements:

```
Notation "x '::=' a" :=
  (ass x a) (at level 60).
Notation "'SKIP'" :=
   skip.
Notation "s1 ; s2" :=
  (comp s1 s2) (at level 80, right associativity).
Notation "'WHILE' b 'DO' s " :=
  (while b s) (at level 80, right associativity).
Notation "'IF_' b 'THEN' s1 'ELSE' s2 " :=
  (if_ b s1 s2) (at level 80, right associativity).
```

What we do with those constructors next, will be specified by the rules we need to formalise. The next section is a small notation interlude, after which we will take a look at the rules and explain why it was chosen.

4.2 Extra Notation

Most of the notations we are going to meet throughout this formalisation, are either quite simple, so that we don't need to pay extra attention to them, or, as above, cleared out together with the definition of what is noted. However, here are some terms and notations, that are not mentioned as often, or not at all, during classes, but still would be useful.

4.2.1 Coercion

This manipulation of notation was already used before. Now, we elaborate on it a bit more. Coercion in **Coq** is a special kind of function, that can be used by the type system to coerce - transform - a value of the input type to a value of an output type. For example:

```
Coercion AId : string >-> Aexp.
Coercion ANum : Num >-> Aexp.
```

We mentioned above, that one of the constructors of Aexp is ANum, which takes a Num. Thus, we should be able to convert a Num to an Aexp, and we use Coercion for that. Note that we just need to match the constructors, not necessarily call the coercion the same way. Instead, we can match using a normal definition as follows:

```
Definition bool_to_bexp (b : bool) : Bexp :=
  if b then BTrue else BFalse.
Coercion bool_to_bexp : bool >-> Bexp.
```

4.2.2 Bind scope

It is important to note that some notations we will be introducing might already be in use. In order to avoid clashing, we will use Bind Scope, which means that within the function scope, this notation will mean what we declared, even if it means something else in other scopes,a s follows:

```
Bind Scope while_scope with Aexp.
Bind Scope while_scope with Bexp.

Notation "x + y" := (APlus x y) (*...*) : while_scope.
```

Note that with Bind Scope we state that the notation of a function is valid in this scope alone.

Instead, with Delimit Scope we set a keyword to show that a certain expression is written, using the notations with this scope - a one-line Open Scope. Then, when we want to use it, we simply write the scope at the end of the example, as follows: Definition example_Aexp:- (3+ (X*2)%while : Aexp.

4.3 Difference between the two composition rules

There are a couple of options how to define whether a statement is completely executed or not. In this section, we will consider two of the approaches, and depict the implementation of the one we choose. Let us first refresh the difference between [comp$^1_{\text{sos}}$] and [comp$^2_{\text{sos}}$].

[skip$_{\text{sos}}$] $\langle \texttt{skip}, s\rangle \Rightarrow s$

[comp$^1_{\text{sos}}$] $$\frac{\langle S_1, s\rangle \Rightarrow \langle S_1', s'\rangle}{\langle S_1;\ \ S_2, s\rangle \Rightarrow \langle S_1';\ \ S_2, s'\rangle}$$

[comp$^2_{\text{sos}}$] $$\frac{\langle S_1, s\rangle \Rightarrow s'}{\langle S_1;\ \ S_2, s\rangle \Rightarrow \langle S_2, s'\rangle}$$

Figure 4.1: A reminder of the three rules in question

We will discuss them in reversed order, as [comp$^2_{\text{sos}}$] is a little bit easier to understand.

[comp$^2_{\text{sos}}$] We use this rule when the statement before `;` can be executed in one step. Let us take an example, that will be further discussed in the application chapter: in the case of `z ::= x ; x ::= y` we will use [comp$^2_{\text{sos}}$], and the first statement, `z::=x` will be easily executed by applying [ass$_{\text{sos}}$], and then we can go straight to the execution of `x ::= y` with another [ass$_{\text{sos}}$]. Now, it is important to notice that the ; operator is right - associative. That means, that `z ::= x; x ::= y; y ::= z`, would be interpreted as `z ::= x; (x ::= y; y ::= z)`. Thus `z ::= x` will be S_1 and `x ::= y; y ::= z` will be S_2.

[comp$^1_{\text{sos}}$] This rule is used when the statement before `;` requires more than one step to be executed. Let us modify the example we used above slightly by adding parentheses: `(z ::= x; x ::= y); y ::= z`. Notice the brackets - we want to associate to the left here and handle the `;` after the brackets first. Now S_1 is `z ::= x; x ::= y` and S_2 is `y ::= z`. We cannot use [comp$^2_{\text{sos}}$], as we cannot execute S_1 in one step - instead we have to apply [comp$^1_{\text{sos}}$], where the first part of S_1, which will initially get executed is `z ::= x` and the part S_1', which is left to do afterwards is `x ::= y`.

It is easy to show that the two three-term examples are semantically equivalent. In other cases, however, consider that S_1 might be a statement that includes `WHILE`. As we know [while$_{\text{sos}}$]'s execution converts it to an `IFS` and maybe more `WHILE`-s afterwards, which for sure cannot happen in one small step.

All of these examples can be found in the appendix.

4.3.1 Skip as Stuck

"Software Foundations" [6] uses SKIP as a command value - as this rule does not change the state, it can be considered "stuck". We can place a SKIP step in the end of each statement. Then, the $[\text{comp}^1_{\text{sos}}]$ rule will be executed, until a statement SKIP is encountered, and only then it will be finished with a $[\text{comp}^2_{\text{sos}}]$ rule.
It is interesting to consider that choice in light of other programming languages. For example,in java and C++, the operation ; for composing statements together are just written after each line. If we use this method, in defining the end of a program, we can also use the same notation as new line for both the rule SKIP and ; SKIP. In other words, the same rule is applied if the end of a program is ;, as it would be if it was ;SKIP.

We can get consistent writing of ; after each command, including the last one, as long as there is an implicit new line at the end of the program. The result would be the habit of writing a ; after each line in languages such as java and C++.

4.3.2 Using a Configuration

Another approach, closer to the notations of Nielson and Nielson [4] is to differentiate between a configuration that is finished and one that still has something to execute. That will give us the difference between $[\text{comp}^1_{\text{sos}}]$: "there is more, we only executed a part of the first statement"; and $[\text{comp}^2_{\text{sos}}]$: "the first of the two statements can be completely executed in one step, we are done after it". This approach is a bit more complicated, but easier to understand.

Because of that, combined with this thesis being meant to assist studying for the course that follows this book, we will use it.

Formalisation of the Skip Configuration

We define the configuration as an Inductive type.

```
Inductive Config: Type :=
|Running (S:Stm) (s:state)
|Final (s:state).
```

As explained about this approach above, we have a Running constructor, that still has a statement that needs to be executed, and Final - which is just a state. The final configuration will be encountered not only in the end of programs, but also in the end of each statement that gets completely executed.

To make it less confusing, we will make matching notation for the two constructors and a coercion to be able to easily convert a state to a final configuration (which contains it alone).

```
Coercion Final: state >-> Config.
```

```
Notation "'≪' s '≫' " := (Final s).
Notation "'≪' S ',' st '≫'" := (Running S st).
```

4.3.3 Stuck With Configurations

Using this configuration setting, we intuitively would know that a `Final` configuration would not make progress.

In subsection 3.2.2 Stuck and Progress we defined stuck as a non-final configuration, from which there is no configuration that can be reached. In turn, in section 5.2 Strong Progress, we will show that **While**, as we defined it makes strong progress - and there is no step that can be made to get to such a configuration.

So, in order to not have to introduce another term, we will slightly alter the definition of stuck, for the sake of the formalisation - to be any configuration, from which there is no configuration that can be reached. Note that we make a small condition that always evaluates to true to make it easier to use that definition in **Coq**.

```
Definition isStuck (conf: Config) :=
0=0 -> ~ (exists (conf':Config), conf ==> conf').
```

The true-evaluation `0=0` is there to set the correct scope. Knowing this, we can now prove that a `Final` configuration is indeed a stuck one:

```
Lemma final_is_stuck:
forall s,
 isStuck ≪s≫ .
Proof.
intros.
unfold isStuck.
intros.
intro.
destruct H0.
inversion H0.
Qed.
```

4.4 Making A Step

Each small step in Structural Operational Semantics can be considered as transition from one configuration - `conf` - to another - `conf'`. In `conf`, we always expect to see one of the statements, and a state in which that statement is supposed to be executed. In `conf'`, we see how that statement changed the state and there might or might not be another statement, that the first one got converted to.

So, we define the notation and the inductive set as follows:

```
Reserved Notation " conf '==>' conf' "
                   (at level 40). (*levels?*)

Inductive sstep : Config -> Config -> Prop:=

(*...*)


Notation " t '==>*' t' " := (multi sstep t t') (at level 40).
```

This, similar to what we wrote with the arithmetical and boolean expressions earlier means that we will use this notation in its own definition.

Now that all the preparatory work is complete, we get to implementing the rules themselves, as constructors of the inductive set `sstep`. The entire function ca be found in section B.2 The basics of SOS formalisation, needed for students to understand the proofs.

[**skip**$_{\textbf{sos}}$] Regardless of which configuration is used - the one we chose or using `SKIP` as a special structure, this rule stays the same. It does not change the state and there is no statement to be executed after it.

```
|SSSkip : forall  st,
  ≪SKIP, st≫ ==> st
```

Note that here the coercion is used, to accept the state given alone as a configuration. We could have written `≪st≫` or `Final st` instead of `st` without change of meaning.

[**ass**$_{\textbf{sos}}$] The rule `x := a` takes a string variable - `x` - that can be assigned a value in the state, or an **Aexp**, and assigns the evaluation of the expression `a` to the variable.

```
|SSAss: forall x a n st,
    aeval st a = n ->
    ≪(x::=a), st≫ ==> ≪(x !-> n, st)≫(*t_update st x n)*)
```

Here, we are using big step evaluation of the arithmetical expression. If we were to try and use small-steps for it the `aeval` would be replaced with a use of `astep`, which we consider in section 7.1 Making small steps in evaluating expressions. The comment depicts an alternative way to represent the updated state.

[**comp**$^{1}_{\textbf{sos}}$] Derivation trees, like the one explaining this rule in Figure 4.1 are written from the bottom up and read and applied top down. That intuition helps to understand why IF S_1 in a state s results in a S_1' in s', THEN S_1 followed by S_2 in the same state s will result in S_1' followed by S_2 in s' and not the other way around.

```
|SSCompI: forall S1 S1' S2 st st',
  ≪S1, st≫ ==> ≪S1', st'≫ ->
    ≪S1; S2, st≫ ==> ≪S1'; S2, st'≫
```

[**comp**$^{2}_{\textbf{sos}}$] With the same reasoning as in the above rule, and the difference already described above, the stress here is on the fact that the initial configuration goes to a final state.

```
|SSCompII:  forall S1  S2 st st',
  ≪S1, st≫ ==> Final st' ->
  ≪S1;S2, st≫ ==> ≪S2, st'≫
```

[**if**$^{\textbf{tt}}_{\textbf{sos}}$] **and** [**if**$^{\textbf{ff}}_{\textbf{sos}}$] Importantly, those are two separate rules. Depending on a boolean expression either the first or second statement is in the resulting configuration, to be the next one executed.

Note that using **Coq**'s already defined `if-else` clause to combine them into one would defeat the purpose of defining those rules from scratch, and require coercing `BTrue` and `BFalse` with the predefined `true` and `false` respectively.

```
|SSIftrue: forall b S1 S2 st,
  Beval st b = true ->
  ≪IF_ b THEN S1 ELSE S2, st≫ ==> ≪S1, st≫

|SSIffalse: forall b S1 S2 st,
  Beval st b=  false ->
  ≪IF_ b THEN S1 ELSE S2, st≫ ==> ≪S2, st≫
```

Same as with [ass$_{\text{sos}}$], we are using big-step evaluation - `beval` on the **Bexp**, but that could instead be a use of `bstep` for small-step evaluation.

Additionally, note that to completely execute those rules, you will need more than one step - one to reduce the `IF-THEN-ELSE-` to one of the statements it chooses from and then at least one step to execute that statement. Thus, if there is a statement that follows after this one, you will need to use [comp$^{1}_{\text{sos}}$] instead of [comp$^{2}_{\text{sos}}$].

[**while**$_{\textbf{sos}}$] Finally, **While** would not be called that way without the [while$_{\text{sos}}$] rule. It can be reduced to execution of the statement and recursive call to itself if the **Bexp** is evaluated to **tt**, or doing nothing otherwise.

```
|SSWhile: forall b S st,
  ≪WHILE b DO S, st≫ ==>
       ≪IF_ b THEN (S; WHILE b DO S) ELSE SKIP, st≫
```

Just like with [if$^{\text{tt}}_{\text{sos}}$] and [if$^{\text{ff}}_{\text{sos}}$], we need more than one - in this case at least three - steps to execute this statement and, if there is a rule following, we should use [comp$^{2}_{\text{sos}}$].

4.5 Making Many Steps

Each of the rules above makes a single step. However, programs are rarely single-statement; if that was the case, we would not even need differentiation between small- and big-step semantics. To be able to prove that programs do what we expect them to, we need to be able to make more than one step. In other words, we need to be able to not only make $\langle \_, \_ \rangle \Rightarrow \langle \_, \_ \rangle$ transitions, but also $\langle \_, \_ \rangle \Rightarrow^* \langle \_, \_ \rangle$ ones, where the latter denotes zero or more of the former transitions.

To do that, we will consider the function `sstep` as a relation between two configurations. Instead of the transition from `conf` to `conf'`, now we will say that configuration `conf` is in relation `sstep` with configuration `conf'` if and only if we can transition from `conf` to `conf'` with one of the rules from `sstep`.

We do that, because we can define a relation on **Coq** as:

```
Definition relation (X : Type) := X -> X -> Prop.
```

Then, we will make that relation reflexive and transitive.

In **Coq**,, this is an inductive definition with:

Reflexive constructor being the 0 case: Making zero steps should still be a star transition. We cannot make a step go from one configuration to the same one, but the we can have a multistep (star transition): $\gamma \Rightarrow *\gamma$, that means we do now make any steps.

Transitive constructor being the induction step: if a transition $\langle S, s \rangle \Rightarrow \langle S', s' \rangle$ is a step and we have a multistep $\langle S', s' \rangle \Rightarrow^* \langle S'', s'' \rangle$, then $\langle S, s \rangle \Rightarrow^* \langle S'', s'' \rangle$ is also a multistep.

In **Coq** we will first define a reflexive and transitive relation more generally. Then we are just going to notate `sstep` as such a relation:

```
Inductive multi {X : Type} (R : relation X) : relation X :=
  | multirefl : forall (x : X), multi R x x
  | multistep : forall (x y z : X),
                    R x y ->
                    multi R y z ->
                    multi R x z.

Notation " t '==>*' t' " := (multi sstep t t') (at level 40).
```

4.6 Making k Steps

Other than eventually reaching a configuration from another, in some cases we also want to explicitly count how many steps are needed to make that transition. Thus, we define a type of cumulative relations, similar to the one we did for the star, that also keeps track of the number of steps, and then we will make our notation for Structural Operational Semantics steps with recorded number:

```
Inductive multi_k {X : Type} (R : relation X) : nat -> relation X :=
  | multikrefl : forall (x : X), multi_k R 0 x x
  | multikstep : forall (x y z : X) (k:nat),
                    R x y->
                    multi_k R k y z ->
                    multi_k R (S k) x z.
Notation " t ==>[ k ]  t' " := (multi_k sstep k t t' ) (at level 50).
```

Using that notation, we will consider some some small but important cases:

4.6.1 Zero Steps

There are a couple of interesting considerations with zero steps. Note, that this is quite different than using any of the rules of Structural Operational Semantics, as we are not actually making a step.

4.6.2 Identity Zero Steps

From any configuration, we can get to the same configuration in zero steps.

```
Lemma identity_zero:
forall c,
 c ==>[0] c.
Proof.
apply multikrefl.
Qed.
```

If and only if two configurations `c` and `c'` are equal, then there can be a transition in zero steps from one to the other.

We will usually only need one of f the directions for our proofs, so we make two separate lemmas:

```
Lemma zero_steps:
forall c c',
c ==>[0] c' -> c = c'.
Proof.
intros.  inversion H. reflexivity.
Qed.
```

```
Lemma zero_steps_rev:
forall c c',
c = c' ->
c ==>[0] c'.
Proof.
intros. induction H.
apply identity_zero.
Qed.
```

4.6.3 Stuck and Stuck Stops

In subsection 4.3.3 Stuck With Configurations me made a small alteration, to call a stuck statement one that does not make progress, regardless if it is final or not, and we defined it as:

```
Definition isStuck (conf: Config) :=
0=0 -> ~ (exists (conf':Config), conf ==> conf').
```

We also promised that section 5.2 Strong Progress we will show that **While** makes progresses with all of its rules. Hence, the only way to get to the end of a derivation sequence is to get to a final configuration. Now we want to prove that indeed if a statements is stuck (either final or really stuck, per the definition of "Semantics with Applications, A Formal Introduction" [4]), it indeed cannot make progress, and that takes us to the end of a derivation sequence.

From subsection 4.6.2 Identity Zero Steps we know, that we can get to the same configuration in zero steps. So, we can conclude, that if we go somewhere from a stuck configuration, then it can only be to the same configuration and in zero steps.

```
Lemma stuck_stops:
forall c c' k,
  isStuck c ->
  c ==>[k] c' ->
  k = 0 /\ c' = c.
Proof.
intros.
unfold isStuck in H.
induction k.
- inversion H0; subst.
 split. reflexivity. reflexivity.
- destruct H0. split. reflexivity. reflexivity.
  destruct H. reflexivity.
  exists y. assumption.
Qed.
```

4.6.4 Final Configuration and Number of Steps

Adding on subsection 4.6.3 Stuck and Stuck Stops, we know, from the altered definition, that `Final` is a stuck configuration. So, we can derive the same thing as in the above-mentioned section, in the specific case, when we try to go to anther configuration from a `Final` one.

```
Lemma state_k_zero:
forall k s conf,
  ≪s≫ ==>[k] conf
  ->
  k = 0 /\ conf = ≪s≫.
Proof.
intros.
apply stuck_stops.
apply final_is_stuck.
assumption.
```

4.6.5 One Step

The `multi_k` relation is, still executing Structural Operational Semantics steps. Thus, making one step with the `multi_k` relation would be the same as making a normal single step. With this clear intuition, we just need to prove it in both directions, so that we can use it later on, when needed.

From single step to one step:

```
Lemma one_step:
forall c c’,
  c ==> c’ ->
  c ==>[1] c’.
Proof.
intros.
apply (multikstep sstep c c’).
assumption. apply zero_steps_rev.
reflexivity.
Qed.
```

And from one step to a single one.

```
Lemma one_step_rev:
forall c c’,
  c ==>[1] c’ ->
  c ==> c’.
```

```
Proof.
intros.
inversion H. subst.
apply zero_steps in H2. subst.
assumption.
Qed.
```

Now we are ready to go and use all that formalisation to make some proofs with Structural Operational Semantics.

4.7 Revisit State

There is a final step we need to make before we are ready to use the framework. We still cannot prove trivial statements like the following:

```
Example triv_3:

≪ x::=y, (x!->1, y!->4, z!->3)≫ ==> ≪ x!->4, y!->4, z!->3≫.
```

That is because somewhere in the proof we would need to use the $[\text{ass}_{\text{sos}}]$ rule, which in turn, will call the function `t_update` on the initial state. The resulting state will be notated as:

```
≪ x !-> 4, x !-> 1, y !-> 4, z !-> 3 ≫
```

instead of

```
≪ x !-> 4, y !-> 4, z !-> 3 ≫
```

which we need. Note that the two states are semantically equivalent but syntactically different. As such, we can easily prove that one is equal to the other, using the below tactic, but **Coq** cannot infer that automatically.

```
Ltac eq_states :=
  apply functional_extensionality; intros; unfold t_update; simpl;
  repeat match goal with
  |- context [eqb_string ?v ?x] =>
    destruct (eqb_string v x)
  end;
  reflexivity.
```

This is enough to show that a state is equal to another as long as their mappings are the same, regardless of what values the variables had before. However, in our formalisation of Structural Operational Semantics, a state is rarely met alone - it is always a part of a configuration - be it the only part, or together with a statement. Therefore now, we want to prove and use the fact that if two configurations consist of the same (or lack of) statements, and equal states, then those configurations are equal as well. The following two theorems show that, with and without a statement in the configurations respectively. Note that both proofs are very simple, including only simple rewriting and evaluating equivalence.

```
Theorem conf_eq_rn:
  forall S s s',
  s = s' -> ≪S,s≫ = ≪S,s'≫.
Proof.
intros.
rewrite H.
reflexivity.
Qed.


Theorem conf_eq_fn:
  forall s s',
  s = s' -> ≪s≫ = ≪s'≫.
Proof.
intros.
rewrite H.
reflexivity.
Qed.
```

Now that we have equality of configurations, we will need to show that, if from one configuration `conf` we can get in one step to another - `conf'` and `conf'` is equal to a `conf''`, then we can get in one step from `conf` to `conf''`. Note that in the **Coq** theorem, we stress on the fact that we can only make a one step transition if in `conf` there is a statement to execute and make at least one step.

```
Theorem stm_eq:
  forall S s conf' conf'',
  ≪S,s≫==> conf' ->
  conf' =conf'' ->
  ≪S,s≫ ==> conf''.
Proof.
intros.
rewrite H0 in H.
apply H.
Qed.
```

We are going to use that to finish those proofs that require a one step transition that changes a variable already present in the state.

In section 4.8 Examples and [comp$^1_{\text{sos}}$] vs [comp$^2_{\text{sos}}$] you will notice that all examples, that require the possibility to make zero or more step, use the transition $\Rightarrow^*$ and finish with applying its constructor `multirefl` we defined in section 4.5 Making Many Steps. That is, we are trying to say that from a final configuration we can get to the same final configuration in exactly zero steps and nowhere else. The next theorem proves that that is the case, regardless of the notation of that state.

```
Theorem no_stm_eq:
  forall s conf' conf'',
  ≪s≫==>* conf' ->
  conf' =conf'' ->
  ≪s≫ ==>* conf''.
Proof.
intros.
rewrite H0 in H.
apply H.
Qed.
```

Finally, we generalise the above statement for all star-transitions. Note that the following theorem can completely replace the usage of the previous one, but we will mostly need it in the above very specific case and thus it is justified that we mention them both.

```
Theorem conf_stm_eq:
  forall conf conf' conf'',
  conf==>* conf' ->
  conf' =conf'' ->
  conf ==>* conf''.
Proof.
intros.
rewrite H0 in H.
apply H.
Qed.
```

Now, with those easy proofs, we can finally proceed to proving some neat examples in section 4.8 Examples and [comp$^1_{\mathrm{sos}}$] vs [comp$^2_{\mathrm{sos}}$].

4.8 Examples and [comp$^1_{\mathrm{sos}}$] vs [comp$^2_{\mathrm{sos}}$]

Most of the simple examples of using this framework are pretty straightforward. To execute many steps, we iteratively apply `multistep` (or `multikstep` in the case when we are interested in knowing exactly how many steps we are talking about) which makes us consider what the first step we need to make is - one of the constructors of `sstep`, and then repeat on the rest of the sequence, until we get to a transition from one state to the same, in which case we can use `multirefl` (or respectively `multikrefl`).

In this section we will illustrate that, together with answering a question that might rise quite often while we talk about Structural Operational Semantics- When to use [comp$^1_{\mathrm{sos}}$]

and when - [comp$^2_{\text{sos}}$]?

We are going to visit in detail an example from the lectures in Semantics and Correctness (in the full file, some simpler examples are discussed beforehand, hence the odd numbering):

```
Example triv_7_lecture':
≪ z ::=x; x ::= y; y ::=z, (x!->3, y!->4,  z!->5)≫ ==>*
    ≪ (y!->3, x!->4, z!->3)≫.
```

Here we will only be using [comp$^2_{\text{sos}}$].

When we start the proof we will use `multistep` three times, which makes for the three steps that need to be executed. Finally, to close the proof, we will use `multirefl`, for the zero step. We have two options:

We can use the standard `apply` when we will need a `with` clause and tell **Coq** what state will the first step result in:

```
Proof.
  – apply multistep with ≪x ::= y; y ::=z, z!->3, x!->3, y!-> 4, z!-> 5≫.
```

Alternatively, we can use `eapply`, which we described in subsection 2.3.3 `eapply`, and let **Coq** infer the goal by itself. In that case, it will use undefined variables, and assume their value as the need comes to assert equality between the variable and another of the same type, using the principle that “it must be it”.

```
Proof.
  – eapply multistep.
```

After that first step, where for clarity, we used the explicit `apply`, we have two subgoals:

```
<< z ::= x; x ::= y; y ::= z, x !-> 3, y !-> 4, z !-> 5 >> ==>
<< x ::= y; y ::= z, z !-> 3, x !-> 3, y !-> 4, z !-> 5 >>
```

Where we need to prove that the second configuration can indeed by reached in one step. And:

```
<< x ::= y; y ::= z, z !-> 3, x !-> 3, y !-> 4, z !-> 5 >> ==>*
(y !-> 3, x !-> 4, z !-> 3)
```

That concerns the rest of the sequence.

Then we continue pretty straightforward - for the first subgoal - we know what the first step leads to, we only need to apply the appropriate constructors of `sstep` we need in order to execute this step.

```
    + apply SSCompII. apply SSAss. reflexivity.
```

Then we use `multistep` again, execute the next step with constructors of `sstep`, and then `multistep` again, on the remaining sequence of length zero or more, until all the steps are executed. In most examples, we do all that with `eapply`, as it is faster and easier to do, but, as mentioned, here we state the intermediate steps explicitly, for clarity.

```
    + apply multistep with (≪y ::=z, x!->4, z!-> 3, x!->3, y!->4,  z!
  ->5≫).
      * apply SSCompII. apply SSAss. reflexivity.
      * apply multistep with ≪(y!->3, x!->4, z!->3, x!->3, y!-> 4, z!
  -> 5)≫.
        apply SSAss. reflexivity.
```

Finally, we need to prove that the resulting state is equal to the one we expected. That will take a use of `multirefl`, to state that a final state can go in zero or more steps - in this case zero - to the same state. Then, we need to use a few more commands due to the problem discussed in section 4.7 Revisit State, and the custom tactic defined there:

```
  eapply no_stm_eq.
  eapply multirefl. apply conf_eq_fn. eq_states.
```

And with that, it all comes together to make a complete proof.

```
  Qed.
```

A different version of this example is on that uses [comp$^1_{\text{sos}}$], with a very small tweak in the definition:

```
Example triv_7_lecture'':
≪ (z ::=x; x ::= y); y ::=z, (x!->3, y!->4,  z!->5)≫ ==>*
    ≪ (y!->3, x!->4, z!->3)≫.
```

Now the first statement to be executed is the one in the brackets. It takes two steps to execute it, and thus needs an application of [comp$^2_{\text{sos}}$].

```
Proof.
  - apply multistep with ≪x::=y; y ::=z,  z!->3, x!->3, y!-> 4, z!-> 5≫.
    + eapply SSCompI. (*<-- Used Comp 1! *)
```

Here, after we applied the `multistep`, we needed to prove:

```
<< (z ::= x; x ::= y); y ::= z, x !-> 3, y !-> 4, z !-> 5 >> ==>
<< x ::= y; y ::= z, z !-> 3, x !-> 3, y !-> 4, z !-> 5 >>
```

Which happens in one step, using [comp$^1_{\text{sos}}$].

Then we finish this subgoal with:

```
      * apply SSCompII. apply SSAss. reflexivity.
```

And further on, the proof is identical to the one described above, starting from the next `multistep` and can be found at section B.2 The basics of SOS formalisation, needed for students to understand the proofs.

Additionally, in the accompanying files on GitHub an example, that uses [while$_{\text{sos}}$] can be found with a proof.

```
Definition triv_8_while: Stm :=
  x ::= 2;
  WHILE (~ (x=1))
  DO x ::= (AId x - 1)
  .
```

Without the use of [comp$^1_{\text{sos}}$]:

```
Example proof_8_while:
  ≪triv_8_while , empty_State≫ ==>* ≪ x!->1 ≫.
```

And with it:

```
Example proof_8_while':
  ≪triv_8_while;SKIP, empty_State≫ ==>* ≪ x!->1 ≫.
Proof.
  unfold triv_8_while.
  - eapply multistep.
    + apply SSCompI. (* <-- NEW*)
```

We will not give the full proofs here, as they can be a bit long, but they can be found on our GitHub page. The notations used there are already discussed in previous chapters.

Chapter 5

Properties

Now that we have formalised Structural Operational Semantics, it is time to discuss some of its properties and prove them. For many of those proofs we use induction on the length of the derivation sequence, as shown in subsection 3.2.4 Induction on the Length of the Derivation Sequence. Those will give some more detail on how Structural Operational Semantics can be applied.

Most of these properties are proven in natural language in “Semantics with Applications, A Formal Introduction” by Nielson and Nielson [4]. Here, we will define them in natural language and then in **Coq**, after which we will prove them only in **Coq**, explaining the steps taken.

5.1 Determinism

First we will consider some properties of single steps rules. We start by proving that Structural Operational Semantics is deterministic.

The book “Semantics with Applications, A Formal Introduction” [4] defines determinism as follows:

If for all choices of S, s, γ, γ':

$$\langle S, s\rangle \Rightarrow \gamma \text{ and } \langle S, s\rangle \Rightarrow \gamma' \text{ implies } \gamma = \gamma'$$

In **Coq**, with our generic naming for configurations, that looks a bit less pretty.

```
Theorem one_step_deterministic:
forall conf' conf'' conf,
  conf ==> conf' ->
  conf ==> conf'' ->
  conf' = conf''.
Proof.
```

We make a case distinction on all the possible rules that can be applied. Not that `c` is `conf'`, `c1 - conf''`, `c2 -conf` and `Hc1` and `Hc2` the two transitions that wipp prove tho be the same.

```
intros c c1 c2 Hc1 Hc2.
generalize dependent c1.
induction Hc1; intros c2 Hc2.
```

In this case `Hc1` is a step transition of the type `conf` ⇒ `conf'`. However, first we need to do a small trick - we generalise one of the variables that `intros` gives us, and put it in the context only after we make the case distinction - for **Coq** this is just another induction.

The [skip_{sos}] and [ass_{sos}] are quite trivial.

```
 - (*SKIP*) inversion Hc2. reflexivity.
 - (*ASS*) inversion Hc2. subst. reflexivity.
```

Next, [$\text{comp}^1_{\text{sos}}$] and [$\text{comp}^2_{\text{sos}}$] prove to be quite difficult. That is because we are doing distinction in the rule applied in the first transition. But, in the case of `;`, for the second transition - the one to `conf''` - there are still two options - whether we use [$\text{comp}^1_{\text{sos}}$] or [$\text{comp}^2_{\text{sos}}$] there.

```
 - (*COMPI*)inversion Hc2; subst.
    + assert (≪S1', st'≫ = ≪S1'0, st'0≫).
      {
        rewrite <- (IHHc1 ≪S1'0, st'0≫).
        reflexivity. assumption.
      }
      inversion H. subst. reflexivity.
    + assert (≪S1', st'≫ = ≪ st'0≫).
      {
        rewrite <- (IHHc1 ≪ st'0≫).
        reflexivity. assumption.
      }inversion H.
```

The proofs for both cases go quite similarly. We first case distinct on the second transition's `;` use. For either case, we assert that the configurations after executing just the first statement are the same in both transitions, and then we use that to prove that indeed the same composition rule was used.

```
 - (*COMPII*)inversion Hc2; subst.
    + assert (≪S1', st'0≫ = ≪ st'≫).
      {
        rewrite <- (IHHc1 ≪ S1', st'0≫).
        reflexivity. assumption.
      } inversion H.
    + assert (≪ st'0≫ = ≪ st'≫).
      {
        rewrite <- (IHHc1 ≪  st'0≫).
        reflexivity. assumption.
      } inversion H. subst. reflexivity.
```

That done, the remaining cases are clear. The [$\mathrm{if}^{\mathrm{tt}}_{\mathrm{sos}}$] and [$\mathrm{if}^{\mathrm{ff}}_{\mathrm{sos}}$] are completely the same, as using the `IF_ ... THEN` rule suggest the correct value of a $\mathcal{B}[\![b]\!]s$ in the state it is used in.

```
- (*IF TRUE*) inversion Hc2; subst.
    + reflexivity.
    + rewrite H5 in H. inversion H.
- (*IF FALSE*) inversion Hc2; subst.
    + rewrite H5 in H. inversion H.
    + reflexivity.
```

And finally, the [$\mathrm{while}_{\mathrm{sos}}$] rule, is as trivial, as the [$\mathrm{ass}_{\mathrm{sos}}$] one - as it can only do one thing.

```
- (*WHILE*)  inversion Hc2; subst.
    reflexivity.
```

With that we have proven that all the rules in our Structural Operational Semantics are deterministic.

```
Qed.
```

We must note here, that if we add some extensions to it - some more rules - we will have to revisit this proof.

5.2 Strong Progress

In subsection 3.2.2 Stuck and Progress we defined the notion of strong progress as per “Software Foundations” [6] that each configuration is either final, or can make a step to another configuration. Now we will prove that that is indeed the case for **While**.

To do that, we first need to define a predicate to check if an arbitrary configuration is indeed final.

```
Definition is_final (conf:Config) :=
  exists s, conf = Final s.
```

Now we will prove in **Coq** that **While**, as we defined it, indeed makes a strong progress.

```
Theorem strong_progress:
forall conf:Config,
 is_final conf
\/
 (exists conf', conf ==> conf').
Proof.
```

The proof goes quite similar to the one for determinism that we just discussed. We will first distinguish the cases of a `Running` and a `Final` configuration.

```
intro.
induction conf.
```

`Running` is the type that comes first on order to be proven, which is the non final option - right in the distinction from our definition. That would leave us to prove:

```
    exists conf', <<S,s>> ==> conf'
```

```
- right. induction S.
```

Unlike the determinism proof, now the case distinction will not be on different ways to make steps, but on the possibilities for a statement. That is, because in our context we have no knowledge that a step is actually made, but only that there is some statement. It is our goal to prove that steps are made.

[ass$_{\text{sos}}$] and [skip$_{\text{sos}}$] are pretty trivial, as they will be for most of the proofs.

```
  + eexists. apply SSAss. reflexivity.
  + eexists. apply SSSkip.
```

For [comp$^1_{\text{sos}}$] and [comp$^2_{\text{sos}}$], we make a case distinction on the what can we get to in one step by only trying to execute the first statement.

```
  + inversion IHS1. induction x.
    * exists ≪S; S2,s0≫. apply SSCompI. assumption.
    * exists ≪S2, s0≫. apply SSCompII. assumption.
```

That case distinction is achieved by the induction on `x`.

[if$^{\text{ff}}_{\text{sos}}$] and [if$^{\text{tt}}_{\text{sos}}$] are the most tricky part of this proof, as we need to manually add to our context that the boolean expression as a condition can either evaluate to true or false and then distinguish on those two cases.

```
  + assert (Beval s b = true \/ Beval s b = false).
    { induction (B[[b]]s).
      - left. reflexivity.
      - right. reflexivity.
    }
   destruct H.
    * exists ≪S1,s≫. apply SSIftrue. assumption.
    * exists ≪S2,s≫. apply SSIffalse. assumption.
```

The [while$_{\text{sos}}$] is pretty easy after this.

```
  + exists ≪IF_ b THEN (S; WHILE b DO S) ELSE SKIP,s≫. apply
   SSWhile.
```

Finally, we are only left to consider the case that a configuration can be final. We first focus on the left side of the disjunction and then the predicate `is_final` will finally come into use.

```
- left. unfold is_final. exists s. reflexivity.
Qed.
```

And with that we have proven that we indeed defined **While** in such a way that it can make a strong progress.

5.3 Composition in $k_1 + k_2$ steps

This lemma, along with its natural language proof is given as Lemma 2.19 in "Semantics with Applications, A Formal Introduction" [4]

Lemma If $\langle S_1; S_2, s\rangle \Rightarrow^k s''$, then there exists a state s' and natural numbers k_1 and k_2 such that $\langle S_1, s\rangle \Rightarrow^{k_1} s'$ and $\langle S_2, s'\rangle \Rightarrow^{k_2} s''$ and $k = k_1 + k_2$.

```
Lemma comp_complete:
forall k Ss1 Ss2 s s'',
  ≪Ss1; Ss2, s≫ ==>[k] ≪s''≫
  ->
  exists s' k1 k2,
    ≪Ss1,s≫==>[k1] ≪s'≫ /\ ≪Ss2,s'≫ ==>[k2] ≪s''≫ /\ k = (k1 + k2).
```

The proof of this lemma is by strong induction on the length of the derivation sequence. The base case - $k = 0$ vacuously holds, as a composition is not a stuck state.

```
Proof.
(*induction on the length of the derivation seqence*)
induction k using strong_induction.
- (*0*)
  intros.
  inversion H.
```

For the induction step we assume the property holds for all $k \leq k_0$, and want to prove it for $k_0 + 1$. We fist get to the same notation in the proof, with the `rename` and then enrich our context a bit.

```
– (*IS*)
  rename k into k0.
  intros.
  inversion H0. subst.
  inversion H2; subst.
```

Now, we have two options what the first of those $k_0 + 1$ steps would be: an execution of either [$\text{comp}^1_{\text{sos}}$] or [$\text{comp}^2_{\text{sos}}$]. We consider them in order of appearance.

5.3.1 [$\text{comp}^1_{\text{sos}}$]

```
  +(*compI*)
```

Here, the context of the **Coq** proof is:

```
    1 subgoal
k0 : nat
H : forall k : nat,
    k <= k0 ->
    forall (Ss1 Ss2 : Stm) (s s'' : State),
    << Ss1; Ss2, s >> ==>[ k] << s'' >> ->
    exists (s' : State) (k1 k2 : nat),
      << Ss1, s >> ==>[ k1] << s' >> /\
      << Ss2, s' >> ==>[ k2] << s'' >> /\ k = k1 + k2
Ss1, Ss2 : Stm
s, s'' : State
H0 : << Ss1; Ss2, s >> ==>[ S k0] << s'' >>
S1' : Stm
st' : State
H3 : << S1'; Ss2, st' >> ==>[ k0] << s'' >>
H2 : << Ss1; Ss2, s >> ==> << S1'; Ss2, st' >>
H7 : << Ss1, s >> ==> << S1', st' >>
______________________________________(1/1)
```

where `H` is our induction hypothesis. It however is quite cumbersome and difficult to use. We extract a small piece to put in the context by itself, by asserting that is true and proving that assertion using `H`.

```
    assert (exists (s' : State) (k1 k2 : nat),
       ≪ S1', st' ≫ ==>[ k1] ≪ s' ≫ /\
       ≪ Ss2, s' ≫ ==>[ k2] ≪ s'' ≫ /\ k0 = k1 + k2).
      {apply H.
        * reflexivity. (*omega.*)
        * assumption.
      }
```

Then we break this substantial piece of information we just asserted down to several smaller hypotheses.

```
destruct H1. rename x into s0. destruct H1. rename x into k1.
destruct H1. rename x into k2. destruct H1. destruct H4.
```

Finally, it is time to break our goal into its three main parts and prove them one by one.

```
exists s0.
exists (S k1).
exists k2.
```

Note that, since we know we are using [comp$^1_{\text{sos}}$], we know that if we assume

$$\langle S_1; S_2, s\rangle \Rightarrow^{k_0+1} s''$$

then the derivation sequence can be written as

$$\langle S_1; S_2, s\rangle \Rightarrow \langle S_1'; S_2, s'\rangle \Rightarrow^{k_0} s''$$

because $\langle S_1, s\rangle \Rightarrow \langle S_1', s'\rangle$ and we can therefore apply the induction hypothesis on

$$\langle S_1'; S_2, s'\rangle \Rightarrow^{k_0} s''$$

So, if $\langle S_1', s\rangle \Rightarrow^{k_1} s''$ and $\langle S_2, s_0\rangle \Rightarrow^{k_2} s''$ then $\langle S_1, s\rangle \Rightarrow^{k_1+1} s''$ and thus $\langle S_1; S_2, s\rangle \Rightarrow^{k_0+1} s''$:

From here, proving the three subgoals is quite easy.

```
split.
 -- apply (multikstep sstep≪ Ss1, s ≫ ≪ S1', st' ≫).
    assumption. assumption.
 -- split.
 --- assumption.
 --- omega.
```

5.3.2 [comp$^2_{\text{sos}}$]

Here, the proof is easier, as we know that we can go from $\langle S_1; S_2, s\rangle$ to $\langle S_2, s'\rangle$ in just one step. Using that, we break the goal in its three separate parts and solve them directly with relative ease.

```
+(*compII*)
  exists st'.
  exists 1.
  exists k0.
  split.
  * apply one_step. assumption.
```

```
    * split.
      assumption.
      reflexivity.
```

Thus, by induction, the lemma is proved.

```
Qed.
```

If this proof's explanation is unclear, it is advised to look at the natural language explanations in “Semantics with Applications, A Formal Introduction” [4].

5.4 Composition in k steps

We will next prove the lemma in the opposite direction:

Lemma If $\langle S_1, s\rangle \Rightarrow^{k_1} s'$ and $\langle S_2, s'\rangle \Rightarrow^{k_2} s''$, then there exists a natural number k such that $\langle S_1; S_2, s\rangle \Rightarrow^{k} s''$ and $k = k_1 + k_2$.

In other words, if two statements can be executed in a respective amounts of steps, the composition of those two statements will be executed in as many steps as the sum of the steps for each of the statements.
We can define it as:

```
Lemma exec_comp:

forall k1 k2 S1 s S2 s' s'',
≪S1, s≫ ==>[k1] ≪ s'≫
->
≪ S2, s'≫ ==>[k2] ≪s''≫
->
exists k,
≪S1; S2, s≫ ==>[k] ≪s''≫ /\ k = k1+k2.
```

To prove that, we will need to do strong induction on k_1 as to that first statement we will ad our extra transition, when we prove the induction step.

Thus, we want to add only k_1 to the context and keep all other variables encapsulated in the `forall` so that we can have a nice, generic, and easy to work with induction hypothesis. That will make for a bit of an uncomfortable use of the lemma later on, as we will see on in Chapter 6, but we need it in order to prove it and use it.

So we define the property we need as:

```
Definition prop_exec_comp (k1:nat) :=
forall k2 S1 s S2 s' s'',
≪S1, s≫ ==>[k1] ≪ s'≫
->
≪ S2, s'≫ ==>[k2] ≪s''≫
->
exists k,
≪S1; S2, s≫ ==>[k] ≪s''≫ /\ k = k1+k2.
```

And the lemma using it:

```
Lemma exec_comp:
forall k1,
prop_exec_comp k1.
```

With that done, the proof is straightforward by strong induction on k_1

```
Proof.
intros.
unfold prop_exec_comp.
induction k1 using  strong_induction; intros.
```

The base case holds vacuously, as it assumes that `S1` is completely executed in zero steps.

```
- exists (k2).
  split.
  assert (≪ S1, s ≫ = ≪ s' ≫).
  {apply zero_steps. assumption. }
  inversion H1.
  omega.
```

When we make the induction step, there is one interesting note that needs to be made:

```
- remember (k1 + k2) as k0.
  exists ( S k0).
```

We `remember` the sum of k_1 and k_2 as its own variable K_0. And when we want to prove the case for $k_0 + 1$ - the induction step - we actually do it for $k_1 + k_2 + 1$. From then on we simply have a case distinction between [$\text{comp}^1_{\text{sos}}$] and [$\text{comp}^2_{\text{sos}}$], quite similarly to the previous proof.

The full proof can be found on our GitHub page.

5.5 Half a Composition

There is another way to consider the composition. This is Essential Exercise 2.21 in the book "Semantics with Applications, A Formal Introduction" [4] and might seem quite intuitive:

Lemma If $\langle S_1, s\rangle \Rightarrow^k s'$ then $\langle S_1; S_2, s\rangle \Rightarrow^k \langle S_2, s'\rangle$.

When we define this property, and, as before, we will try to poorve that it holds for all k.

```
Definition property_half_comp(k:nat) :=
(forall S1 s s',
    ≪S1,s≫ ==>[k] ≪s'≫
   ->
    forall S2,
      ≪S1; S2, s≫ ==>[k] ≪S2, s'≫).

(*2.21*)
Theorem half_comp:
forall k,
  property_half_comp k.
```

We again unfold the property and do a straightforward proof by strong induction.

```
Proof.
intros.
unfold property_half_comp.
induction k using strong_induction.
```

There is nothing interesting in the proof to show here - it can be found in the accompanying files.

Note, that instead of using this trick, which we do here for clarity, we can use the quantifier with k in the first place - `forall k S1 s s'` - and then, when we start the proof use `intro` (which will only add `k` to the context and nothign else) instead of `intros` followed by an `unfold`.

5.6 Equivalence between $*$ and k

In Chapter 4 we defined the notion of a star ($*$) - making an unknown amount of steps to get from one configuration to another - and k - going from one to another configuration in exactly k steps. As they both use the same rules, they must be equivalent in the sense that if two configurations are linked with a star relation, so are they with k relation - and vice versa.

We define it as follows:

```
Lemma k_equiv_star:
forall conf conf',
  conf ==>* conf'
 <->
  exists k,
    conf ==>[k] conf'.
```

And prove it easily by applying two properties - first that star implies k, and then that k implies star, which we will prove directly after this lemma.

```
Proof.
intros. split.
- apply star_implies_k.
- intros.
  inversion H.
  apply k_implies_star with x. assumption.
Qed.
```

Those two statements are proven by induction on the respective relations. In both cases, we use the reflexive constructor of the relation for the base case, and apply once the step-constructor to get to finally apply the induction hypothesis.

```
Lemma star_implies_k:
forall conf conf',
  conf ==>* conf'
 ->
  exists k,
    conf ==>[k] conf'.
Proof.
intros.
induction H.
 - exists 0. apply zero_steps_rev. reflexivity.
 - inversion IHmulti.
   exists (S x0).
   apply multikstep with y.
    + assumption.
    + assumption.
Qed.

Lemma k_implies_star:
forall conf conf' k,
  conf ==>[k] conf'
 ->
  conf ==>* conf'.
Proof.
intros.
induction H.
- apply multirefl.
```

```
– apply multistep with y.
  + assumption.
  + assumption.
Qed.
```

Here we use a small lemma called `zero_steps_rev`, discussed in subsection 4.6.2 Identity Zero Steps, but now it nothing more than a shorthand that a configuration can go to itself in exactly zero steps. The tactics `apply zero_steps_rev. reflexivity.` in the first lemma can be replaced by `apply multikrefl.`

Chapter 6

Equivalence with Natural Semantics

Structural Operational Semantics is very useful for building compilers, stack machines, and handling many language functionalities. However, some applications, such as proving semantic equivalence, relation to Axiomatic Semantics, proving soundness and completeness, etc. are much easier done using Natural Semantics. Many works [6] [4] [8] [5] focus on that and do those difficult proofs only in Natural Semantics. Small step relation to Axiomatic Semantics is done in some works, for example in “Axiomatic Semantics for Compiler Verification” by Schäfer, Schneider and Smolka [7], or “The Formal Semantics of Programming Languages An Introduction” by Glynn Winskel [9]. However, they do not treat some problems specific for our topic, such as describing the rule $[\text{comp}^1_{\text{sos}}]$ with Axiomatic semantics and take very different approaches to defining Structural Operational Semantics.

So, instead of dwelling in those complex proofs, we will follow the example of “Semantics with Applications, A Formal Introduction” [4] and leave all that to Natural Semantics. Here, we will prove that Structural Operational Semantics is equivalent to Natural Semantics and thus that all the properties that hold for Natural Semantics hold for Structural Operational Semantics as well.

6.1 Brief description of Natural Semantics

We mentioned before the thesis “A Formalization of Natural Semantics of Imperative Programming Languages in Coq” by Loes Kruger [3], written in parallel with this thesis. It discusses Natural Semantics in depth and proves many properties that Structural Operational Semantics, talked about in this thesis, inherits because of the equivalence. Here we will cite parts of the formalisation in her work, that we will need to prove the equivalence - more specifically the **While** rules. Note that it uses the same definition of **While** we have discussed so far.

6.1.1 Seval

The most important part of Loes Kruger's work [3] we will need is her formalisation of Natural Semantics:

```
(* semantics Stm and notation for transitions *)
Reserved Notation "conf '-->' st'"
                  (at level 40).

Inductive Seval : Config -> State -> Prop :=
  | ass_ns  : forall st a1 n x,
    Aeval st a1 = n ->
    Seval (Running (x ::= a1) st) (t_update st x n)
  | skip_ns : forall st,
    Seval (Running SKIP st) st
  | comp_ns : forall s1 s2 st st' st'',
    Seval (Running s1 st) st' ->
    Seval (Running s2 st') st'' ->
    Seval (Running (s1;s2) st) st''
  | if_tt_ns : forall st st' b s1 s2,
    Beval st b = true ->
    Seval (Running s1 st) st' ->
    Seval (Running (IF_ b THEN s1 ELSE s2) st) st'
  | if_ff_ns : forall st st' b s1 s2,
    Beval st b = false ->
    Seval (Running s2 st) st' ->
    Seval (Running (IF_ b THEN s1 ELSE s2) st) st'
  | while_tt_ns : forall st st' st'' b s,
    Beval st b = true ->
    Seval (Running s st) st' ->
    Seval (Running (WHILE b DO s) st') st'' ->
    Seval (Running (WHILE b DO s) st) st''
  | while_ff_ns : forall b st s,
    Beval st b = false ->
    Seval (Running (WHILE b DO s) st) st.
```

Until now we have used our notation for a configuration in **Coq** quite a lot, which, together with the notation for a Natural Semantics transition `-->` would make for an inductive definition, that is a little bit easier to read, while equivalent to the above.

```
(* semantics Stm and notation for transitions *)
Reserved Notation "conf '-->' st'"
                  (at level 40).

Inductive Seval : Config -> State -> Prop :=
  | ass_ns  : forall st a1 n x,
      Aeval st a1 = n ->
      ≪x ::= a1, st≫ --> (x!->n, st)
  | skip_ns : forall st,
     ≪SKIP, st≫ --> st
  | comp_ns : forall s1 s2 st st' st'',
      ≪s1, st≫ --> st' ->
      ≪s2, st'≫ --> st'' ->
      ≪(s1;s2), st≫ --> st''
  | if_tt_ns : forall st st' b s1 s2,
      Beval st b = true ->
      ≪ s1, st≫ --> st' ->
      ≪ IF_ b THEN s1 ELSE s2, st≫ --> st'
  | if_ff_ns : forall st st' b s1 s2,
      Beval st b = false ->
      ≪s2, st≫ --> st' ->
      ≪IF_ b THEN s1 ELSE s2, st≫ --> st'
  | while_tt_ns : forall st st' st'' b s,
      Beval st b = true ->
      ≪ s, st≫ --> st' ->
      ≪WHILE b DO s, st'≫ --> st'' ->
      ≪WHILE b DO s, st≫ --> st''
  | while_ff_ns : forall b st s,
      Beval st b = false ->
      ≪WHILE b DO s, st≫  --> st
where "conf '-->' st'" := (Seval conf st').
```

Importing it to our project enables us to prove the equivalence.

6.2 Equivalence between Natural Semantics and Structural Operational Semantics

As defined in "Semantics with Applications, A Formal Introduction" [4], we will want to prove that $\langle S, s\rangle \rightarrow s'$ if and only if $\langle S, s\rangle \Rightarrow^* s'$. We define a theorem for it in **Coq**

```
Theorem ns_eq_sos:
forall S s s',
  ≪S,s≫-->s'
 <->
  ≪S,s≫==>*s'.
```

And prove it, using two lemmas that we will discuss afterwards.

```
Proof.
intros.
split.
apply ns_implies_sos.
apply sos_implies_ns.

Qed.
```

How we prove those lemmas `ns_implies_sos` and `sos_implies_ns`, however, is a much longer matter. Note that a natural language proof can be found in “Semantics with Applications, A Formal Introduction” [4]. In that book the two implementations are in opposite order, but here we start with proving that Structural Operational Semantics implies Natural Semantics, as that is closer to what we have been doing so far, and only after that we get to Natural Semantics implies Structural Operational Semantics.

In both directions, we will first prove that this is indeed the case for a SOS transition in k steps, and then use the equivalence between star and k relation - proven in section 5.6 Equivalence between $*$ and k - to prove the lemmas in the theorem above.

6.2.1 Structural Operational Semantics in k steps implies Natural Semantics

We will use the trick we already showed in some of the properties, and define a property depending on the variable we will do induction on, and then use that in the lemma itself:

```
Definition sos_ns_k (k:nat):=
forall S s s’,
  ≪S,s≫==>[k]≪s’≫
 ->
  ≪S,s≫-->s’.


Lemma sos_k_implies_ns:
forall k,
  sos_ns_k k.
Proof.
```

Then we will do induction on the length of the derivation sequence - that is only possible since we use the k relation instead of the star one.

```
Proof.
intros.
unfold sos_ns_k.
induction k using strong_induction.
```

The base case holds vacuously.

```
- intros. inversion H.
```

In the induction step, we add all of our assumptions to the context, and then make a case distinction based on what rule we could apply first. We are doing strong induction - that means that for any number of steps smaller or equal to k_0, we know that the property indeed holds. So, we want to prove that it will still be the case when we add a different step in the start - putting the total of $k_0 + 1$ - and after we make that new initial step, there will be only k_0 steps to make, for which we can just apply the induction hypothesis.

We will not give the entire proof here.

It is important to note that here we want to prove transitions of Natural Semantics. That is the reason why the rule ordering, among other things, is a bit different - using the strong iduction on `k` makes cases that match the possible **While** constructions.

[ass$_{\mathbf{ns}}$]

For the case of [ass$_{\text{ns}}$], we first assert that executing the [ass$_{\text{sos}}$] indeed leads to a state where the value of the arithmetical expression is assigned to the variable. Then we assert that this state leads in zero steps to s'.

```
  + (*ass*)
    assert (y = ≪(x !-> A[[ a]] s, s)≫).
    {
      apply one_step_deterministic with ≪ x ::= a, s ≫.
      assumption. apply SSAss. reflexivity.
    }
    subst.
    assert ( k0 = 0 /\ ≪s'≫ = ≪(x !-> A[[ a]] s, s)≫).
    {
      apply state_k_zero. assumption.
    }
    inversion H1. inversion H5.
    subst.
    apply ass_ns. reflexivity.
```

In fact most of the cases have similar assertion based structure, and thus we will not repeat it more than necessary .

[skip$_{\mathbf{ns}}$]

This case is similar, with the difference of that the state does not change, and we assert that it is indeed the case.

[$\mathbf{comp_{ns}}$]

We first apply the property we proved in section 5.4 Composition in k steps, to split our k composition into executing two separate statements. Then we apply the rule [comp$_{\text{ns}}$] on the goal, which gives us two subgoals, as per the definition of this NS rule.

```
+ (*comp*)
  apply comp_complete in H0.
  destruct H0. destruct H0. destruct H0. destruct H0.
  apply comp_ns with x.
```

For each of them, we do induction on the length of the derivation sequence for executing S_1 and S_2 respectively. Then for each of those two induction steps, we do induction on the opposite statement.

Here you can see the application of the nested inductions, in one of the cases:

```
*     induction x1.
      --  assert ( ≪ S2, x ≫ = ≪ s' ≫).
          apply zero_steps. destruct H1.  assumption.
        inversion H4.
      -- apply  H with x0.
```

...

$\left[\mathbf{if^{tt}_{ns}}\right]$ and $\left[\mathbf{if^{ff}_{ns}}\right]$

This case has the structure `IF_` b `THEN` s_1 `ELSE` S_2. We first assert that the statement transits in one step in either S_1 or S_2 to be executed in the same state.

```
+ (*if*)
  assert (y = ≪S1, s≫ \/ y = ≪S2, s≫).
  {
    inversion H2; subst. left. reflexivity.
                         right. reflexivity.
  }
```

Now, we split those two cases. Those are the cases, where Structural Operational Semantics leads to S_1 or S_2. Now, we want to prove that those cases can indeed happen in Natural Semantics- by case switching on what b can evaluate in the starting state.

```
* (* where y = ≪S1, s≫ *)
  inversion H2; subst.
  --  (*b evalates to true*)
     apply if_tt_ns.
        assumption.
        apply H with k0. reflexivity. assumption.
```

```
      --(*b evalates to false*)
          apply if_ff_ns.
            assumption.
            apply H with k0. reflexivity. assumption.
```

The case of `y = ≪S2,s≫` is identical.

$\left[\text{while}_{\text{ns}}^{\text{tt}}\right]$ and $\left[\text{while}_{\text{ns}}^{\text{ff}}\right]$

Finally, in the case of `WHILE`, we need to get a few more steps before we can prove the implication. First, we assert that the $[\text{while}_{\text{sos}}]$ rule does indeed what we expect it to do, and add that to the context.

```
  + (*while*)
assert (y = ≪IF_ b THEN (S; WHILE b DO S) ELSE SKIP, s≫).
        {
          inversion H2; subst. reflexivity.
        }
```

Now we treat the proof similarly like the above case, apply the `IF_` b `THEN` s_1 `ELSE` S_2 structure, only now we know exactly what S_1 and S_2 are. Here `y` is the next intermediate configuration.

```
...
assert (y = ≪S; WHILE b DO S, s≫ \/ y = ≪SKIP, s≫).
        {
          inversion H1. left. reflexivity.
                        right. reflexivity.
        }
```

As we now know exactly what statements there are, we do not need the second case distinction. Instead, in the first case, we apply the lemma we proved in section 5.3 Composition in $k_1 + k_2$ steps.

```
        * (* y = ≪S; WHILE b DO S, s≫, hence b evaluates to true*)
          apply comp_complete in H4.
```

Then we separate that assumption, as there are a few cluttered properties there. Finally we can apply $\left[\text{while}_{\text{ns}}^{\text{tt}}\right]$ and take care of all the goals generated by that.

```
          eapply while_tt_ns with x.
            -- inversion H1; subst. assumption.
            -- apply H with x0. omega. assumption.
            -- apply H with x1. omega. assumption.
```

Note that here, H is the induction hypothesis generated for this case from our strong induction all the way in the beginning:

```
H : forall k0 : nat,
    k0 <= S k ->
    forall (S : Stm) (s s' : State),
    << S, s >> ==>[ k0] << s' >> -> << S, s >> --> s'
```

Lastly, after some transformations of the context, in the `SKIP` case, we can apply $\left[\text{while}^{\mathbf{ff}}_{\text{ns}}\right]$.

```
          apply while_ff_ns. assumption.
```

.

With that, we have proven that Structural Operational Semantics in k steps implies Natural Semantics.

```
Qed.
```

6.2.2 Structural Operational Semantics implies Natural Semantics

Now, that we have proven that Structural Operational Semantics implies Natural Semantics when executed in k steps, we can use the equality between the k and the star relation, that we proved in section 5.6 Equivalence between $*$ and k and prove that it is the case when for a star sequence as well.

```
Lemma sos_implies_ns:
forall S s s',
  ≪S,s≫==>*≪s'≫
 ->
  ≪S,s≫-->s'.
```

We prove it by adding everything to the context, turn the star into k relation using the equality, generalise it back, and apply the lemma we just proved in the previous section.

```
Proof.
intros.
apply star_implies_k in H.
inversion H.
rename x into k. clear H.
(*careful to have the genralizations in the same order.
Otherwise, the premises will not match.*)
generalize dependent s'.
generalize dependent s.
generalize dependent S.
generalize dependent k.
apply sos_k_implies_ns.
Qed.
```

6.2.3 Natural Semantics implies Structural Operational Semantics in k steps

Just like we approached the previous problem from the k steps side, we will first prove that Natural Semantics implies Structural Operational Semantics when a configuration is reached from another in exactly k steps. We define it as:

```
Lemma ns_k_implies_sos:
forall S s s’,
  ≪S,s≫-->s’
  ->
exists k,
  ≪S,s≫==>[k]s’.
```

We will now need to do induction on the shape of the derivation tree. That is, our case distinction is going to be on the possible rules that can be applied from Natural Semantics.

```
Proof.
intros.
  induction H.
```

[ass$_{\text{ns}}$] and [skip$_{\text{ns}}$] are trivial - we apply the appropriate rules to the Structural Operational Semantics transition, and then, as they were just one specific step, we evaluate that the resulting states are the same. We include their proofs here, as an example of using the `multi_k` constructors.

```
  - (* ass *) exists 1. eapply multikstep. apply SSAss.
    apply H. eapply multikrefl.
  - (* skip *)exists 1. eapply multikstep. apply SSSkip. eapply
   multikrefl.
```

For [comp$_{\text{ns}}$], we will have to apply the rule that we proved in section 5.4 Composition in k steps. Because of the generalisation trick we used there, we need to rewrite it here, and only afterwards use it.

We already have `x` in our context, being the number of steps to get from $\langle S_1, s\rangle$ to the final state configuration s'.

```
                    assert (
                   forall k2 S1 s S2 s’ s’’,
                        ≪S1, s≫ ==>[x] ≪ s’≫
```

```
                 ->
                  ≪ S2, s'≫ ==>[k2] ≪s''≫
                 ->
                  exists k,
           ≪S1; S2, s≫ ==>[k] ≪s''≫ /\ k = x+k2).

             { apply exec_comp.
             }
```

With everything that this assertion adds to the context when broken down, it is now only a matter of applying assumptions to finish this case.

The proofs for $\left[\mathrm{if}^{\mathbf{tt}}_{\mathrm{ns}}\right]$ and $\left[\mathrm{if}^{\mathbf{ff}}_{\mathrm{ns}}\right]$ are identical, with applying the `multikstep` constructor and respectively either the $[\mathrm{if}^{\mathrm{tt}}_{\mathrm{sos}}]$ or the $[\mathrm{if}^{\mathrm{ff}}_{\mathrm{sos}}]$ rules. Here is just one of the cases.

```
  – (*if_tt*) destruct IHSeval. clear S.
    exists (S x). apply multikstep with ≪ s1, st ≫.
     * apply SSIftrue. assumption.
     *  assumption.
```

Finally, we consider $\left[\mathrm{while}^{\mathbf{tt}}_{\mathrm{ns}}\right]$ and $\left[\mathrm{while}^{\mathbf{ff}}_{\mathrm{ns}}\right]$. In both those cases, we need two small steps, to get from the `WHILE... DO s0` statement (where `s0 is a statement`, to either actually start executing the statement `s0`, or to got to the end of it - a `SKIP` statement. In the $\left[\mathrm{while}^{\mathbf{tt}}_{\mathrm{ns}}\right]$ case, we represent that with calling the successor of the number of steps needed to finalise it twice:

```
  – (*while_tt*)
    destruct IHSeval2. destruct IHSeval1. clear S.
    exists (S (S (x0+x))).
```

We then apply $[\mathrm{while}_{\mathrm{sos}}]$, followed by $[\mathrm{if}^{\mathrm{tt}}_{\mathrm{sos}}]$.

```
    apply multikstep with
        ≪IF_ b THEN (s0; WHILE b DO s0) ELSE SKIP, st≫.
        * apply SSWhile.
        * apply multikstep with
          ≪ s0; WHILE b DO s0, st≫.
          —  apply SSIftrue. assumption.
```

Finally, we fall back into proving the composition case all over again, and we will not show that again. At the very end of this proof, we get to the $\left[\mathrm{while}^{\mathbf{ff}}_{\mathrm{ns}}\right]$ rule, which is just three steps away from terminating - applying $[\mathrm{while}_{\mathrm{sos}}]$, $[\mathrm{if}^{\mathrm{ff}}_{\mathrm{sos}}]$ and $[\mathrm{skip}_{\mathrm{sos}}]$ in that order, just like we would do in a non-generic case:

```
  - (*while_ff*)
    exists 3.
    apply multikstep with
        ≪IF_ b THEN (s0; WHILE b DO s0) ELSE SKIP, st≫.
    + apply SSWhile.
    + apply multikstep with ≪SKIP, st≫.
      * apply SSIffalse. assumption.
      * apply multikstep with ≪st≫.
        apply SSSkip. apply zero_steps_rev. reflexivity.
```

With that, our proof - that Natural Semantics implies Structural Operational Semantics with k steps - is complete.

```
Qed.
```

6.2.4 Natural Semantics implies Structural Operational Semantics

Now, we can use the above proved lemma, to show that if we can go from one configuration to a state in Natural Semantics then that is the case with the Structural Operational Semantics star relation as well.

```
Lemma ns_implies_sos:
forall S s s’,
  ≪S,s≫-->s’
 ->
  ≪S,s≫==>*s’.
```

We use the Natural Semantics transition that is already in our context to add there the Structural Operational Semantics with k steps, for some k.

```
Proof.
   intros.
   assert(exists k,
  ≪S,s≫==>[k]s’).
  {
    apply ns_k_implies_sos. assumption.
  }
```

Now the context and our goal are as follows:

```
1 subgoal
S : Stm
s, s’ : State
H : << S, s >> --> s’
```

```
H0 : exists k : nat, << S, s >> ==>[ k] << s’ >>
______________________________________(1/1)
<< S, s >> ==>* << s’ >>
```

We then only need to get a specific k - call it `x` in this case, and use it to prove the lemma.

```
  destruct H0.
  apply k_implies_star with x.
  assumption.
Qed.
```

6.2.5 Again: Equivalence between Structural Operational Semantics and Natural Semantics

With that we have all the parts we used in our theorem that Natural Semantics and Structural Operational Semantics are equivalent.

```
Theorem ns_eq_sos:
forall S s s’,
  ≪S,s≫-->s’
 <->
  ≪S,s≫==>*s’.
Proof.
intros.
split.
apply ns_implies_sos.
apply sos_implies_ns.
Qed.
```

The proof is now complete. The full proof can be found in accompanying files.

Chapter 7

Extensions

7.1 Making small steps in evaluating expressions

The evaluation functions in section 4.1 Expressions are enough for the goal of this thesis, that focuses on small step statement execution. However, there the definition of boolean and arithmetical expression have one way of execution - completely. It is possible, instead, to make small steps with arithmetical and boolean evaluations too.

We mentioned in the previous chapter, that Structural Operational Semantics in **While** is deterministic. Thus, we know that a configuration can lead to at most one state. So, won't small steps for evaluating the expressions change that?

No. Whether we execute the entire statement, or just a part of it in Structural Operational Semantics depends not only on the operator, but also on the form of the arguments. In a way, we can consider the small step of the execution as an elaborate way to stress on the associativity.

7.1.1 Making steps with Aexp

To do small step evaluation, instead of a `Fixpoint`, we will use an `Inductive` evaluation function, simiarly to how we defined the small step statements.

Before that, we will want to be able to convert all the numbers we will work with (e.g. the ones in`1+1+1`), into an **Aexp**, which can be added, subtracted and multiplied, before, in the end, the result is given back as a number.
We are denoting this operation with`==>a`.

```
Reserved Notation "'A[[' a ']]' st '==>a' a' "
                  (at level 40, st at level 39).
```

```
Inductive astep : state -> Aexp -> Aexp -> Prop :=
  | AS_Num: forall st n,
     A[[n]] st ==>a n
  | AS_Id : forall st i,
      A[[AId i]] st ==>a ANum (st i)
  | AS_Plus1 : forall st a1 a1' a2,
      (A[[a1]] st ==>a a1') ->
      (A[[APlus a1 a2]] st ==>a (APlus a1' a2))
  | AS_Plus2 : forall st v1 a2 a2',
      aval v1 ->
      A[[a2]] st ==>a a2' ->
      A[[APlus v1 a2]] st ==>a (APlus v1 a2')
  | AS_Plus : forall st n1 n2,
      A[[APlus (ANum n1) (ANum n2)]] st ==>a ANum (n1 + n2)
  | AS_Minus1 : forall st a1 a1' a2,
      A[[a1]] st ==>a a1' ->
      A[[(AMinus a1 a2)]] st ==>a (AMinus a1' a2)
  | AS_Minus2 : forall st v1 a2 a2',
      aval v1 ->
      A[[a2]] st ==>a a2' ->
      A[[(AMinus v1 a2)]] st ==>a (AMinus v1 a2')
  | AS_Minus : forall st n1 n2,
      A[[AMinus (ANum n1) (ANum n2)]] st ==>a (ANum (minus n1 n2))
  | AS_Mult1 : forall st a1 a1' a2,
      A[[a1]] st ==>a a1' ->
      A[[AMult a1 a2]] st ==>a (AMult a1' a2)
  | AS_Mult2 : forall st v1 a2 a2',
      aval v1 ->
      A[[a2]] st ==>a a2' ->
      A[[AMult v1 a2]] st ==>a (AMult v1 a2')
  | AS_Mult : forall st n1 n2,
      A[[AMult (ANum n1) (ANum n2)]] st ==>a (ANum (mult n1 n2))

    where "'A[[' a ']]' st '==>a' a' " := (astep st a a').
```

This constructor evaluates just one step of the equation. We will use the `multi` relation we defined in section 4.5 Making Many Steps to evaluate a small-step expression:

```
Notation "'A*[[' a ']]' st '==>a' a' " := (multi astep st a a') (
at level 40).
```

It is difficult to miss that there are three versions for Plus, Minus and Mult each in this implementation. The `astep` is a recursive call that only does operation on expressions if they are fully reduced. So, expressions like $11 + (x - (4 + 1))$ will be converted to:

```
AS_Num(AS_Plus2(11 (AS_Minus2 (AS_ID X) (AS_Plus 4 1))))
```

and solved form the inside constructor out. Here, associativity could matter, depending on the order in which we write the constructors.

Note, that in "Semantics with Applications, A Formal Introduction" [4], this notation is defined for the relation definition of arithmetical expressions. However, the two definitions are later proven to be equivalent. As we will not dwell deeper into the difference between those definitions, it suffices to say, that the operation denoted `A[[_]] ==>a _` is going to evaluate one step of an arithmetical expression, similarly the notation `_ ==> _` which executes a statement. In turn `A*[[_]] ==>a _` could evaluate the entire expression, regardless of how many steps it takes, similarly to`==>*` for executing statements, making many steps.

7.1.2 Making steps with Bexp

Just like in **Aexp**, we can do small step evaluation of **Bexp**. To do that, we will need an inductive definition. Here, we do not need to evaluate boolean values, as in our simple language, we can just take the appropriate constructor - `BTrue` or `BFalse`.
We are denoting this operation with`==>b`.

```
Inductive bstep : state -> Bexp -> Bexp -> Prop :=
| BS_Eq1 : forall st a1 a1' a2,
    A[[a1]] st ==>a a1' ->
    B[[BEq a1 a2]] st ==>b (BEq a1' a2)
| BS_Eq2 : forall st v1 a2 a2',
    aval v1 ->
    A[[a2]] st ==>a a2' ->
    B[[BEq v1 a2]] st ==>b (BEq v1 a2')
| BS_Eq : forall st n1 n2,
    B[[BEq (ANum n1) (ANum n2)]] st ==>b
    (if (n1 =? n2) then BTrue else BFalse)
| BS_LtEq1 : forall st a1 a1' a2,
   A[[a1]] st ==>a a1' ->
    B[[BLe a1 a2]] st ==>b (BLe a1' a2)
| BS_LtEq2 : forall st v1 a2 a2',
    aval v1 ->
    A[[a2]] st ==>a a2' ->
    B[[BLe v1 a2]] st ==>b (BLe v1 a2')
| BS_LtEq : forall st n1 n2,
   B[[BLe (ANum n1) (ANum n2)]] st ==>b
            (if (n1 <=? n2) then BTrue else BFalse)
| BS_NotStep : forall st b1 b1',
    B[[b1]] st ==>b b1' ->
    B[[BNot b1]] st ==>b (BNot b1')
| BS_NotTrue : forall st,
    B[[BNot BTrue]] st ==>b BFalse
| BS_NotFalse : forall st,
    B[[BNot BFalse]] st ==>b BTrue
| BS_AndTrueStep : forall st b2 b2',
    B[[b2]] st ==>b b2' ->
    B[[BAnd BTrue b2]] st ==>b (BAnd BTrue b2')
| BS_AndStep : forall st b1 b1' b2,
    B[[b1]] st ==>b b1' ->
    B[[BAnd b1 b2]] st ==>b (BAnd b1' b2)
| BS_AndTrueTrue : forall st,
    B[[BAnd BTrue BTrue]] st ==>b BTrue
| BS_AndTrueFalse : forall st,
    B[[BAnd BTrue BFalse]] st ==>b BFalse
| BS_AndFalse : forall st b2,
    B[[BAnd BFalse b2]] st ==>b BFalse

where "'B[[' t ']]' st '==>b' t' " := (bstep st t t').
```

Similarly to `astep`, this function only makes one small step. And to evaluate the entire sequence of small steps, we use the `multi` relation.

```
Notation "'B*[[' b ']]' st '==>b' b' " := (multi bstep st b b') (
at level 40).
```

7.1.3 Conclusion - small step expressions

We have shown that it is possible to indeed define a step in evaluation the arithmetical expression in many steps, using `Inductive`. Now we can return to freely using the `Fixpoint` which evaluates the entire expression in one goal. That way of evaluating - without making the steps separately - is overall simpler and allows us to focus more on the Structural Operational Semantics of statements.

Chapter 8

Experiment

This formalisation of Structural Operational Semantics, as well as the one of Natural Semantics written in parallel by Loes Kruger [3], is primary meant to serve as a studying aid in the course Semantics and Correctness. Therefore, we need to test whether it is indeed helpful. The main challenge in that is that the course does not, at the time of writing this, feature a part in **Coq**. So, the experiment that needs to be made, has to include an explanation that it introduces enough **Coq** knowledge to make the formalisation understandable, but not too much, to completely defeat the purpose of aiding the studying process by giving too much of information and confusing students.

8.1 Research Question

The research question of this experiment is: **Does the formalisation of Natural Semantics and Structural Operational Semantics in Coq help Semantics and Correctness students to make proofs on paper?**

8.2 Experiment

We make a comparative study on whether being introduced to the formalisation of Natural Semantics and Structural Operational Semantics. In other words, we want to test whether introducing the formalisation next to the regular study is better than just the regular study materials. Note that we at no point make the claim that this formalisation can replace work with the regularmaterial.

8.2.1 Participants

While this formalisation covers all the basic terms in **While**, Structural Operational Semantics and (in Loes Kruger's work [3]) Natural Semantics, it is not aimed specifically towards people with no background in the topics and it matches the course Semantics and Correctness in Radboud University, so we are sure that this course provides the background needed. Thus, we need participants that are following, or have followed the course Semantics and Correctness. Additionally, if the participants already completely understand all concepts to a passable level, do not find them difficult, and do not need to study them anymore, then this formalisation hardly has something to add for them. Therefore, we need participants, who are currently students in the course Semantics and Correctness, and have not concluded it successfully yet. From that very limited pool, sampling is easy - we would invite all students currently in that situation to participants, as they are not that many, and the ones who agree are the participants in this experiment.

8.2.2 Variables

For our analysis, we use just one independent binary variable - ”the participants have had no contact with the formalisation” versus ”the formalisation has been introduced to them and some exercises were done with it”. The dependent variable, that we will measure is how difficult do the students find the relevant topics of Semantics and Correctness - Natural Semantics and Structural Operational Semantics - on a scale from one to ten.

8.2.3 Research Hypothesis

We hypothesise that there will be a decrease, relative per participant, in the perceived difficulty of the two Semantics and Correctness topics after introduction and practising with the formalisation in comparison to beforehand. In other words, the hypothesis is that the participants will find the two Semantics and Correctness topics easier after using the formalisation than before.

8.3 Setup

The experiment is set up to be of within subject design. We can afford to do that, as we are comparing whether spending time on the formalisation is better than not doing so, and not whether spending time on the formalisation is better than spending the same amount of time on studying the book, slides or homework. That might also be an interesting point to examine, but we will not discuss it here.

8.3.1 Baseline - Before

The participants are first asked to fill in a questionnaire, found at section A.1 Before the Meeting, to establish how they perceive the difficulty of Natural Semantics and Structural Operational Semantics. Later on, their updated opinions will be compared to this baseline.

8.3.2 Task

A meeting, with the length of a study block - an hour and forty five minutes is conducted with the participants. During it, in a panel lecture context, the formalisation is presented to the participants and all their questions regarding it are answered. For more consistent results and replicability, that meeting could be replaced by a written text. However, since the volume of information is very big, and we do not want this experiment to take the participants an unreasonable amount of time, we consider a meeting to be the better option, as a text would take longer to read. Regardless of whether it is text or a lecture, the topic would be the same - an explanation of the basic formalisation and some proofs made with it - theones for SOS can be found in Appendix B Experiment Structural Operational Semantics- **Coq** files for Structural Operational Semantics, and the explanations - in the appropriate chapters above.

Both for Natural Semantics and Structural Operational Semantics the files with formalisation to explain, contain one proof that will not be discussed. Participants are asked to try and prove the lemma this proof is for, in natural language, using the **Coq** proof alone. They are allowed to spend as much time as they would like, but are asked to deliver the proofs within a few days from the meeting. That is the actual task, where they can test for themselves the formalisation, see how it works and decide whether using it helps them to understand the topics, hence they would perceive them as less difficult.

8.3.3 Measuring the Depending Variables - After

After the new proofs are submitted, the participants are asked to fill in a second questionnaire (found at section A.2 After the Meeting) with their updated perception of the difficulty of the two topics. A comparison can then be made, per participant and in general, to see if there is a significant positive change in their opinion. We are making a one directional statistical test, as we are only looking for positive change, and do not check for or expect a negative change, as there is no reason for one to occur.

Additionally, in the case of few participants, the learning progress can be seen from the quality of the proofs they made.

Finally, a final meeting is conducted, to take the participants' opinion on the formalisation and receive advice, if some is given, on how to make it more suitable to fulfil its goal.

8.3.4 Avoiding Confounding Variables and Possible Modification

Now we will mention some variables that might be confounding and explain how we avoid them or how they can be avoided in the future. Additionally, we will mention some things we could have done differently and why we didn't.

Different study ways per participant

Naturally, the participants would have different levels and capacity of understanding the material. However, since we are comparing for each of the Semantics and Correctness students to their own baseline, we are only looking whether their own perception of difficulty improved. Additionally, for some participants the improvement might be bigger than for others. Hence, both a mean of improvement and separate per person would both be interesting observations.

Varying amount of time invested

This variable's effect is limited by the short time given the students to submit their proof - there is a maximum total time participants can spend on it. Additionally, if a student needs more time to understand the formalisation, that will affect their response of perceived difficulty (if you to spend a lot of time trying to understand something, you would not mark it as easy).

Any time spent on the topics would result in the topic being easier

This variable is also limited by the short time to make the proof, but not as much.

A way to avoid both this and the previous point is to make a between-subject designed experiment, where a control group spends a set amount of time studying the already provided material, and a test group spends the same amount of time studying the formalisation. However, in the current situation, that is not ideal, as time to get used to **Coq** and understanding how the formalisation is used needs to be factored in - in this case we use a meeting or a text, but that is inapplicable on the control group. In the future, that design could be tested with more carefully selected participants, who are familiar with both **Coq** and the material of Semantics and Correctness. In section section 8.4 Execution we will show another reason why such a design is not well applicable in this experiment.

Finally, our participants, outside of the experiment, have an incentive to spend time on the course - it is a mandatory one for some programs. Thus they are assumed to have spent a lot of time on studying the conventional materials, and if they already have an opinion on how difficult they find the topics and spending a few more hours on them might improve their understanding of the material, but not the perceived difficulty.

Using grades as measuring the effect, instead of perceived difficulty

If we were to use testing and grades on the proofs to measure our depending variable, we would have the three confounding variables mentioned above with much bigger effects, as the experiment would test how well do students study instead of how easy it is for them. Additionally, we would need to account for students' performance under pressure, decisiveness and ability to do well on tests and most of all their learning capacity. Our result would be much more dependent on the effort the participants put in the experiment, than on the learning aid capabilities of the formalisation.

8.4 Execution

During execution the experiment, we follow the above design closely. Here the specific conditions of the execution are described.

The biggest challenge in executing this experiment is its timing. This thesis is written in the third and fourth quarter of the study year. However, in this year, the course Semantics and Correctness is run during the first quarter. Therefore, the only students, that fulfil the requirements we discussed above for participants, are ones, who have started the course, but for some reason did not pass the final exam. Out of this very limited set of possible participants, only five students agreed to participate in an experiment that is related for a course that is not currently running.

Another challenge is the two questionnaires participants needed to fill in. We could not give them an anonymous identification to add at each questionnaire, and hence the two got disjointed, which rendered individual comparisons impossible. If the experiment is replicated this is a notable point for improvement.

8.4.1 Pros

The small sample size makes for a nice personal relation with the participants, and provides ample time for questions. Because of this, it is possible to read trough all the proofs made and comment on that, as well as receive much needed feedback. As this is the first version of this formalisation, feedback on improving the study aid is very much appreciated.

Additionally, most of these participants have spent a lot of time trying to understand the material. Therefore, the few hours they can spend on the formalisation are just a small percentage of what they had spent and the confounding variable **any time spent on the topics is an improvement** is severely limited.

8.4.2 Cons

This small set of participants is not a representative sample. It makes the **different ways and speeds of studying** confounding variable have a very big effect. In fact, our results will now not be whether the formalisation is a good studying aid for Semantics and Correctness students, but whether the formalisation is a good studying aid for students, who need to redo Semantics and Correctness. Additionally, so few samples are not enough to make a viable statistical test and hence we are going to only look at the values we gain instead. Therefore it is highly recommended that the experiment, with the same design, is repeated while the course Semantics and Correctness is running.

8.5 Results

Over all, the experiment did not provide positive results. Throughout, the sample size dropped from five to four people, who did the proofs to only three,w ho filled in the second questionnaire. When measuring the baseline, before any tasks were completed, the participants were quite optimistic. They predicted that using the **Coq** formalisation would be helpful to them and were in fact excited to try.

Unfortunately, executing the tasks took the participants a lot of time and effort and, while there were some ideas how the formalisation could be useful, the overall opinion was that using the formalisation as is was quite confusing for them.

However, with this sample size and bias, and the fact that not all the participants had the same experience, the results of this experiment were by no means conclusive. Additionally, with forty percent (two) of the participants not giving all the quantitative data we needed, a statistical analysis was rendered useless. Below, we will list those results in more detail, as well as the feedback given by the participants.

A vital future step would be to incorporate that feedback and repeat the experiment with a larger sample size. That second experiment's result might be completely different from the current one.

8.5.1 Baseline

We use a questionnaire to get quantitative data before the **Coq** tutorial - the meeting described in the above section - which we will later compare to the later results from another questionnaire.

The first question in the questionnaire checked whether the choice for the topics induction on the shape/length of the derivation tree/sequence was appropriate.

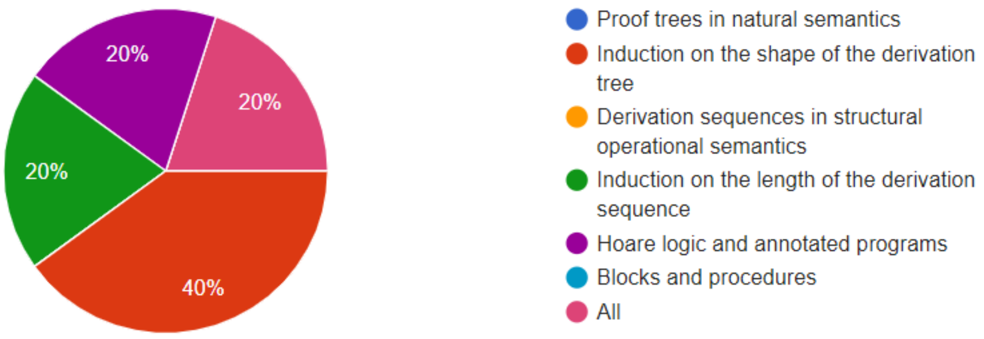

Figure 8.1: Before Questionnaire, Question 1

As can be seen in Figure 8.1, induction proofs were chosen by three out of five students and another participant chose “All” which includes induction. This indicates that these topics were indeed considered difficult and the choice to show them in the tutorial was appropriate.

The following questions dived deeper into the induction. According to participants, induction on the shape of the derivation tree has an average difficulty of 6.8/10. This is the case because they find it difficult to understand the concept, apply the induction step correctly and use the complex terms included in the proof. Induction on the length of the derivation sequence has an average difficulty of 6.2/10. Some people find this more difficult because there is no visual representation, while others find it easier because it uses sequences instead of trees.

With that established, we asked the participants how much they like building proofs in **Coq**. This question was present to establish participants’ previous opinion of **Coq**. The previous experiences might influence their reception of the **Coq** formalisation. The average score for this question was 5.4/10 - sightly above average.

Moreover, we asked the participants if they think a **Coq** formalisation would help them understand the proofs better.

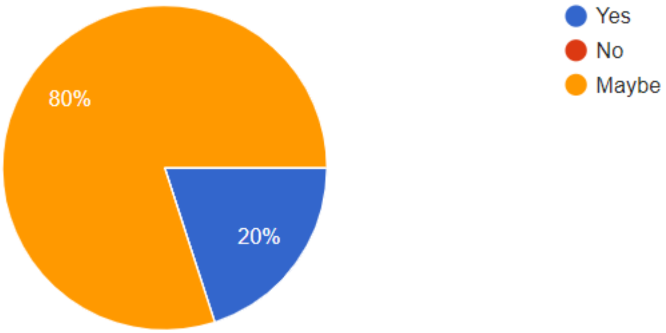

Figure 8.2: Before Questionnaire, Question 8

As can be seen in Figure 8.2, the participants think a formalisation in **Coq** might help them. Reasons for that could be that it easier to correct mistakes in **Coq** and the fact that **Coq** can give real-time feedback on whether the step taken is possible. According to participants it is easier when you can directly see the results of your actions. However, there might also be downsides because it is necessary to learn to work with the formalisation while learning how the actual proofs work. Because you are learning multiple things at once, using the formalisation might get confusing.

8.5.2 Task Execution

As we mentioned above, the task execution took much more time than expected. In the planning of the experiment, we expected not more than two days for that. However, because the experiment was run during exam period, most of the students only had the time to work on them two weeks later. The proof they were asked to do, indeed did not take more than two days once they were started. Therefore, for future running of the experiment, a motivation to complete the task at a time closer to the **Coq** tutorial is recommended. In this case, as the proof were started later on, it is possible that that affected their perceived difficulty, as some information form the **Coq** tutorial might have been forgotten.

Another problem that participants encountered was that in order to make the natural language proofs, they needed to pull even older knowledge about how to do those proofs from Semantics and Correctness lectures from more than half a year ago. Again, that would be easier for students if it was parallel with the course.

8.5.3 Questionnaire After the Tasks

After completing the task, participants were asked to fill in a second questionnaire, to compare with their initial expectations. Only three participant did that, and hence those results are hardly reliable.

They were asked first how helpful they found the tutorial on a scale form one to ten. The average score for this question was 3.67/10. This score is very low, partially because of the confounding factors discussed before. Moreover, in subsection 8.5.4 Student Feedback we will mention all the improvements that the participants suggested that, if implemented, would be able to raise this result.

It is interesting to note that the individual scores were 1, 2 and 7, which indicates that for one student, it was noticeably helpful.

Then, the participants were asked of they were able to finish the proofs and which materials they used to make their proof.

While the participants were initially enthusiastic that they could complete them for two days, two weeks later the results in this question were not very positive, as can be seen in Figure 8.5. None of the participants completely finished the proofs. It was commonly mentioned that they did not have the time and were busy with other work. It is noteworthy, that all of the participants were indeed students with exams near the time of the experiment. One participant finished the Natural Semantics proof but not the Structural Operational Semantics one. The reason for that is probably that in the course Semantics and Correctness, the order of teaching is first Natural Semantics and then Structural Operational Semantics, and hence the participant started in the same order and did not have the time to write both proofs.

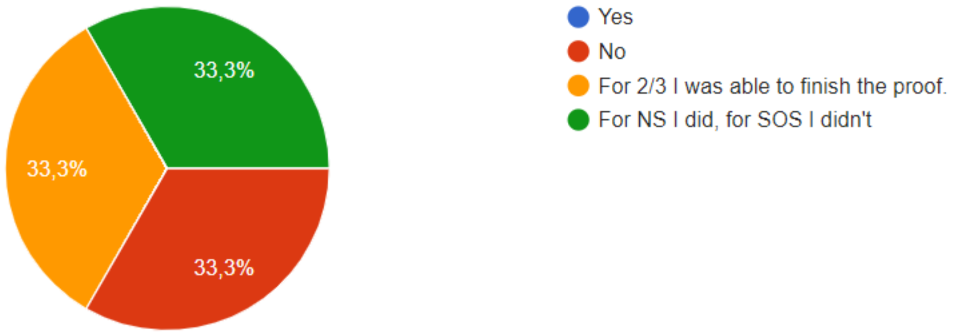

Figure 8.3: Proof finish rate

Most participants focused on using the **Coq** formalisation but also needed to consult the slides, as shown on Figure 8.4. That could partially be attributed to the necessity of improvements, as well as the fact that the participants needed to extensively consult the slides in order to remember how a natural language proof should look like. If the experiment is repeated during Semantics and Correctness, we expect highly different result.

One participant managed to base their whole work on the **Coq** formalisation, which makes us hopeful that it is indeed possible to do so.

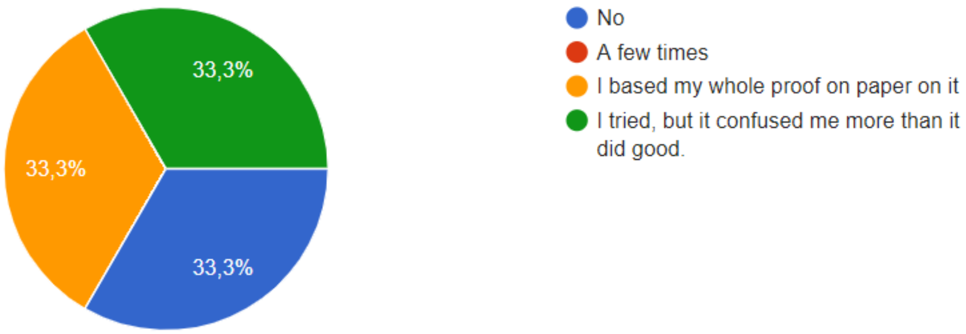

Figure 8.4: Usage of **Coq** during proofs creation

Afterwards, they were asked to rate how difficult induction on the shape of the derivation tree and induction on the length of the derivation sequence were respectively on a scale form one to ten. This was done to compare the new result to the initial one, from before the tutorial. However, as the first questionnaire was filled by five participants and the second one - by three, we cannot make conclusions about changes on average. Small changes do occur, but the two values are in the range of error due to the sample size. Unfortunately, we did not foresee participant dropping out an did not make identifier per participant, to make individual compassion possible.

Finally, the participants were asked if they would like to see some **Coq** proofs in the course Semantics and Correctness.

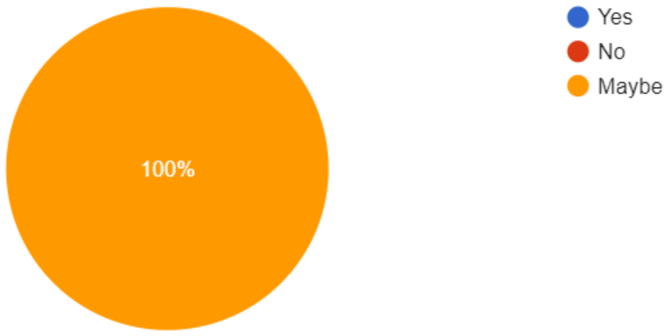

Figure 8.5: After Questionnaire, Question 9

All participants answered 'Maybe'. When asked to elaborate, they recommended a number of improvements, discussed in subsection 8.5.4 Student Feedback. However, worry was expressed that adding it to Semantics and Correctness increase the workload of the already heavy subject.

8.5.4 Student Feedback

Finally, qualitative data was gathered using a free form meeting,where each participant had the opportunity to share their own experience and give feedback. Here, we will describe their points.

Visualisation The participants mentioned that it was difficult to understand the proving part in **Coq**. In the previous course Logic and Applications, there was a visual representation of the tree that was made. All participants agreed that having a visual representation along side the proof would be helpful. In Logic and Applications the interface was also easier to use. The rules had names similar to the names used in the course. If we could make a simple interface like that, it might also make the formalisation more understandable. This point is more valid for Natural Semantics than for Structural Operational Semantics.

Many learning topics It was mentioned that the course Semantics and Correctness was quite a while ago and it took some time to understand how the induction proofs work again. Because of this, it also felt like the participants needed to learn multiple things at once: the **Coq** formalisation, the used notation, induction proofs in **Coq** and induction on the shape/length of the derivation tree/sequence.

Meaning of Coq rules There needs to be a better introduction and more explanation to be used in the course or in a bigger experiment. It was difficult to understand how certain rules work, especially the `inversion` and `generalize dependent` rules were considered difficult. This could be accomplished by writing an elaborate manual on how to use all the rules. Another suggestion was to start with small proofs, show a normal proof by induction and slowly work up towards a proof by induction on the shape/length of the derivation tree/sequence. Possibly using small assignments in between so the participants can figure out how the rules work themselves. The suggested structure is similar to the one followed in this thesis. However, reading it would take the participants more time.

Workload With all the effort needed to understand **Coq**, the participants showed concern that and increase of the workload that might occur if **Coq** proofs are incorporated in the course. The course is already quite time consuming and has difficult concepts. Adding work with **Coq** might make the workload too high. On the other hand, it could not hurt to have a working formalisation available for students that are interested and understand **Coq** better.

Induction - handling mistakes Most of the students found the induction proofs difficult to understand and it did not help them to make the proofs on paper. However, we also asked if the proofs that were discussed during the tutorial would be useful. Most participants said that they would find that very helpful if they understood it because making trees and sequences in **Coq** is easier. That is, because it is easier to correct mistakes and check if certain rules can be applied.

Additional formalisation We asked if there were other parts of the course where a **Coq** formalisation would help. Most participants agreed that Blocks and Procedures, which also consist of single derivation trees, would be useful if a visual representation would be provided.

Automatism One participant mentioned that **Coq** might not be helpful because most of the work is performed by the inversion rule that was difficult to understand. According to that participant, looking at the Coq proof is similar to looking at the solution for the proof on paper. It was also mentioned that looking at solutions does not increase understanding. In other words, there was concern that the **Coq** proof does not teach students how the proof should be made, but only shows a solution.

Benefit Next to all the improvement suggestions and concerns, the participants did mention that they were glad that it was looked into ways to make the course easier. They also mentioned that the group might be biased as they already found the topics difficult. Other students might find the formalisation more useful.

8.5.5 Discussion

The experiment will most definitely need to be made a second time with a bigger sample size on order too get definitive results. Its setup proved to be viable, but there are some changes that can be made to it as well.

As we already mentioned, the period for task execution was longer than previously estimated. In the planning of the experiment, we expected around two days for the participants to make their proofs on paper. However, because the experiment was run during exam period, most of the students only had the time to work on the proofs two weeks later. The proof they were asked to do, indeed did not take more than two days once they started. It is possible that the long period between the tutorial and making the proofs on paper affected the perceived difficulty, as some information from the **Coq** tutorial might have been forgotten. This might also explain why the average score for how useful the tutorial was very low. However, in the questionnaire we found that some participants did not or barely made use of the actual **Coq** formalisation while making the proofs. Because the tutorial was set up to help the participants translate a **Coq** proof to a proof on paper, this might also affect the tutorial score. It is important to note that one participant did score the usefulness of the tutorial with a seven. This leads us to believe that the usefulness of the tutorial and the **Coq** formalisation depends heavily on the participants themselves.

Another problem that participants encountered was that in order to make the proofs on paper, they needed to pull even older knowledge about how to do those proofs from Semantics and Correctness lectures from more than half a year ago. Because of that, results for the experiment are less reliable. It might also be a factor in why the participants were not able to complete the proofs.

If a similar experiment will be conducted in the future, it should include Semantics and Correctness students during a run of the course and with a stricter deadline for submitting the proof on paper after the tutorial. It would also be better to link the first and second questionnaire of a single participant: that would allow for individual comparisons of the results and accounting for participants dropping out.

Finally, the results are also not very reliable because the experiment started with five students but was only finished completely by three students. This group is too small to generalise the findings to all the Semantics and Correctness students.

8.5.6 Conclusion Experiment

Currently, the formalisation does not help much students to understand the proofs better. Despite that there seems to be enough possible improvements that would make the formalisation better. For example, a visual representation of the trees, exercises with the **Coq** formalisation to get used to it and better documentation might make the formalisation more helpful. However, the workload must be taken into account if the **Coq** formalisation is incorporated in the course.

Chapter 9

Conclusion

This thesis aimed to answer the following research question: **Is it possible to formalise Structural Operational Semantics for While in Coq in a way that can help learn it in the context of the course Semantics and Correctness?**

We have showed that it is indeed possible to create a formalisation of Structural Operational Semantics, that follows the material of Semantics and Correctness - most of which overlaps with the book “Semantics with Applications, A Formal Introduction” by Nielson and Nielson [4]. However, we did not get definitive results of whether that formalisation is indeed helpful to studying the topic. Instead, we gained useful insight on how to improve the formalisation in such a way that it can be a good studying aid.

It is worth to note, however, that there is a difference between how good a studying aid the students find this formalisation on average, and how many of the students find it a good studying aid. While the improvements aim to make it as good source of information as possible for as many students as possible, this formalisation's goal is to only be used by students who find it helpful. Therefore, if there is at least one student for whom it is useful, which as the experiment already showed is the case, then that formalisation is already a helpful study aid.

9.1 Challenges in formalising Structural Operational Semantics

Building the formalisation was not an easy task. We first needed to formalise **While** and the states. Then, before we could proceed to the rules of Structural Operational Semantics, we needed to make a choice as to how to differentiate between a configuration that can make progress and one, that is final and has led to the end of the execution of the statements. We chose a typical functional programming tactic to making a configuration type that can be either running - in which case it takes a statement to execute and a state, or final where there is only a state to stop at.

Then we got to formalise the rules and proofs of many properties of Structural Operational Semantics: We proved that Structural Operational Semantics for **While** is deterministic and that if makes strong progress. Then we dove into making not only a single small step, but a sequence of them, where we either kept track how many steps we are taking or not (star transition). That allowed us to make proves using induction on the length of the derivation sequence. We used this technique to to prove the following properties:

- That if and only if two statement get executed in k_1 and k_2 respectively, then their composition's execution takes $k_1 + k_2$ steps.
- That if a statement gets executed in k steps, then a composition of this statement to a second one, would get to needing to execute only the second one in k steps.
- That if and only if we can get from one configuration in k steps, then we can make those steps without counting them - using the star notation.

With those we covered all the most common properties, usually associated with Structural Operational Semantics.

The rest of the properties that Structural Operational Semantics has, are quite difficult to prove using SOS. Usually Natural Semantics is used to prove them. Hence, we proved that Structural Operational Semantics is equivalent to Natural Semantics, which was one of the most challenging proofs in this thesis. That done, now we know that the properties of Natural Semantics depicted in the thesis about Natural Semantics written in parallel by Loes Kruger [3] also hold for Structural Operational Semantics.

The full formalsiation can be found on our GitHub page.

9.2 Comparison to related work

In chapter 1 Introduction it was mentioned that quite a few formalisations already exist but they do not match the course Semantics and Correctness. The current formalisation does match the course based on "Semantics with Applications, A Formal Introduction" by Nielson and Nielson [4] better, with respect to the other works referenced, for the following reasons:

- "Software Foundations" [6] uses only natural numbers and not Numerals, as the definition of **While** does, while our formalisation redefines numbers as numerals.
- "Software Foundations" [6] uses `SKIP` as a sort of stuck configuration, which is not the case in Semantics and Correctness. Hence, we make our own type for a configuration that distinguishes between `Running` and `Final`.
- "Software Foundations" [6] focuses is on small step evaluation of expressions. While we do mention them as an extension, we do not focus on it at all, and just directly

evaluate it, similarly to “Semantics with Applications, A Formal Introduction” [4].

- “Software Foundations” [6] does introduce the star transition, but not the *k* one, where we count the number of steps made. As a result, it does not consider induction on the length of the derivation sequence at all, while it is a main building block in this thesis.
- In “Software Foundations” [6] most of the examples are with an even simpler toy language than **While** which does not extend to its use in Semantics and Correctness, that we follow.
- There are very few examples on how to prove equivalence between Natural Semantics and Structural Operational Semantics and the ones that were found used completely different notations and rules.
- This formalisation has notation that matches more closely to the notation in the book. For example, the transitions in Software Foundations look like : `-->`, which here are reserved for Natural Semantics. We try to stick as close as possible to “Semantics with Applications, A Formal Introduction” [4], with its $\langle \_, \_ \rangle \Rightarrow' \langle \_, \_ \rangle$ by using ≪ _ , _ ≫ `==>` ≪ _ , _ ≫, which is as close as we can get without overriding already existent notations like `<,>` or `=>`, using ascii symbols. Moreover, the names of the types match the names used in the course as much as possible.
- This formalisation has many examples directly taken from the courses book - “Semantics with Applications, A Formal Introduction” [4] - or the slides. This helps to understand it and make a direct comparison between proofs in the book and in the formalisation.
- The formalisation is in **Coq**, which is already familiar to most of the students. Other formalisation were made in Isabelle, which is an unfamiliar language.
- From the custom type selection and notation, it follows, that all the proofs have their own way of progress, that might be similar to some but not exactly compatible with other works.

This list indicates that this formalisation definitely matches the course Semantics and Correctness better than the other sources used (primarily, but not only “Software Foundations” [6]).

9.3 Future work

The formalisation could be improved in several ways. Naturally, there are more properties and extensions of **While** that can be included. For example the course Semantics and Correctness has a follow-up course Semantics and Rewriting. The first lecture covers the Break and Continue statements. It would be very interesting to extend the thesis with those rules as they are quite relevant for Structural Operational Semantics. However, this would not be the primary concern.

The more obvious focus of future work should be on making the existent formalisation a better studying aid.

It would be interesting to perform the experiment during the Semantics and Correctness course. This would give more reliable results and would provide more feedback on how to further improve the formalisation and its explanation.

Before that however, there is already existent feedback that can be incorporated. Better documentation of the proofs and explaining what the **Coq** tactics do would be a good first step. A build up on that would be to redefine some tactics to better match the steps taken in a natural language proof, similar to the way tactics are redefined to bear the name of the respective rule in the course Logic and Applications at Radboud University.

Another interesting extensions would be to find alternatives for the mapping of the states. Currently, the formalisation uses the definition for total and partial maps from "Software Foundations" [6]. However, this definition does not match nicely with the notation used in the course.

Acknowledgements

Firstly, I would like to thank my supervisors, dr. Engelbert Hubbers and dr. Freek Wiedijk, who were there every step of the way, ready to assist if need be. Thank you for agreeing to give all the advice, help and effort, for making it feel like every question was worth asking and for making your passion for Logic and Semantics contagious.

I would also like to thank Ms. Loes Kruger, who did a parallel thesis in Natural Semantics for sharing the burden. You did the other half.

And finally, I would like to thank my family, without whose support this thesis would have been a very sad endeavour.

Bibliography

[1] The Coq development team. *The Coq proof assistant reference manual*, 2004. Version 8.0.

[2] Engelbert Hubbers. Proposals BSc thesis projects. In *Semantics of imperative programming languages in Coq*, 2020.

[3] Loes Kruger. A Formalization of Natural Semantics of Imperative Programming Languages in Coq. 2020.

[4] Hanne Riis Nielson and Flemming Nielson. Semantics with applications; a formal introduction, 1992.

[5] Tobias Nipkow and Gerwin Klein. *Concrete Semantics with Isabelle/HOL.* Springer International Publishing, 2014.

[6] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Software Foundations.* Electronic textbook, 2017.

[7] Steven Schäfer, Sigurd Schneider, and Gert Smolka. Axiomatic Semantics for Compiler Verification. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2016, page 188–196, New York, NY, USA, 2016. Association for Computing Machinery.

[8] Norbert Schirmer. A Verification Environment for Sequential Imperative Programs in Isabelle/HOL. In Franz Baader and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 398–414, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[9] Glynn Winskel. *The Formal Semantics of Programming Languages An Introduction.* The MIT Press Cambridge, Massachusetts London, England, 1994.

Appendix A

Questionnaire for the experiment

A.1 Before the Meeting

1. Which Semantics and Correctness topic did you find the most difficult?
    - Proof trees in natural semantics
    - Induction on the shape of the derivation tree
    - Derivation sequences in structural operational semantics
    - Induction on the length of the derivation sequence
    - Hoare logic and annotated programs
    - Blocks and procedures
    - Other
2. Why do you think this was difficult?
3. How difficult do you think induction on the shape of the derivation tree is on a scale from 1 to 10?
4. Why do you think this was easy/difficult?
5. How difficult do you think induction on the length of the derivation tree is on a scale from 1 to 10?
6. Why do you think this was easy/difficult?
7. How much do you like making proofs in Coq on a scale from 1 to 10?
8. Do you think seeing natural semantics and operational structural semantics proofs in Coq would help you to make proofs on paper?

  - Yes
  - No
  - Maybe

9. Why do you think that seeing proofs in Coq would help/not help?
10. Do you have any further questions or comments?

A.2 After the Meeting

1. How helpful did you think the Coq tutorial was on a scale from 1 to 10?
2. Did you manage to finish the proofs on paper?
    - Yes
    - No
    - Other
3. Did you use the proofs in Coq while making the proof on paper?
    - No
    - A few times
    - I based my whole proof on it
    - Other
4. Did you use any other materials when you were working on the proofs?
    - Yes, the book
    - Yes, the slides
    - Yes, the online solutions
    - No, I only used the Coq proof
5. How much did the tutorial/proofs in Coq help when making the final proofs on paper on a scale from 1 to 10?
6. How difficult do you think derivation on the shape of the derivation tree is on a scale from 1 to 10 after the tutorial?
7. How difficult do you think derivation on the length of the derivation sequence is on a scale from 1 to 10 after the tutorial?
8. Did you give different ratings than in the first questionnaire and why?

9. Would you like to see some Coq proofs in the course Semantics and Correctness?
10. Why would you like to see/ not see Coq proofs in the course Semantics and Correctness?
11. Do you have any suggestions to improve the formalization?
12. Do you have any suggestions to improve the explanation of the formalization?
13. Do you have further questions or comments?

Appendix B

Experiment Structural Operational Semantics- Coq files

B.1 While description, common for SOS and NS

```
(* This file explains the basic framework and the NS and SOS
   files import
   from this file. *)
(* Imports from the Coq standard library *)
Set Warnings "-notation-overridden,-parsing".
From Coq Require Import Bool.Bool.
From Coq Require Import Init.Nat.
From Coq Require Import Arith.Arith.
From Coq Require Import Arith.EqNat.
From Coq Require Import omega.Omega.
From Coq Require Import Lists.List.
From Coq Require Import Strings.String.
From Coq Require Import Logic.FunctionalExtensionality.
From Coq Require Import BinNat.
Import ListNotations.

(* The language While uses integers, we need to open the Z scope
   We also use a special scope for defining the Statements *)
Local Open Scope Z_scope.
Declare Scope while_scope.
Open Scope while_scope.

(*Num *)
Inductive Num : Type :=
  | NZero
  | NOne
  | NEven (n : Num)
  | NOdd (n : Num).

Fixpoint Neval (n : Num) : Z :=
  match n with
  | NZero => 0
  | NOne => 1
  | NEven n => 2*(Neval n)
  | NOdd n => 2*(Neval n) + 1
  end.

Coercion Neval: Num >-> Z.

(* We want integers to be reconginized as numerals and
   numerals as integers *)
Fixpoint pos_to_num (n:positive): Num:=
 match n with
  |xH => NOne
  |xO n' => NEven (pos_to_num n')
  |xI n' => NOdd (pos_to_num n')
  end.

Fixpoint z_to_num (z:Z) : Num :=
  match z with
   | Z0 => NZero
   | Zpos n => pos_to_num n
   | Zneg n => pos_to_num n
   end.

Coercion z_to_num: Z >-> Num.

(* The state is defined as a total map. Total maps can be defined
   in Coq by making a general total_map function. This function
   maps a string it to an element of the specified type A *)
Definition total_map (A : Type) := string -> A.

(* We need a total map from strings to integers *)
Definition State := total_map Z.

(* Total maps need a default element, the t_empty function maps
   an
   undefined element to the default element v of some type A *)
Definition t_empty {A : Type} (v : A) : total_map A :=
  (fun _ => v).

(* eqb_string compares strings *)
Definition eqb_string (x y : string) : bool :=
  if string_dec x y then true else false.

(* t_update updates the mapping of string x to new value v of
   some
   type A *)
Definition t_update {A : Type} (m : total_map A)
                    (x : string) (v : A) :=
  fun x' => if eqb_string x x' then v else m x'.

(* We add some notation to make it easier to use these functions
   in proofs *)
Notation "'_' '!->' v" := (t_empty v)
  (at level 100, right associativity).

(* The default element for the state should be 0.
   This means that every string that doesn't have a specific
   mapping, maps to integer 0 *)
Definition empty_State := (_ !-> 0).

(* Some more notations *)
Notation "x '!->' v ',' m" := (t_update m x v)
  (at level 100, v at next level, right associativity).
Notation "x '!->' v" := (x !-> v,empty_State)
  (at level 100, v at next level, right associativity).

(* Syntax Aexp *)
Inductive Aexp : Type :=
  | ANum (n : Num)
  | AId (x : string)
  | APlus (a1 a2 : Aexp)
  | AMinus (a1 a2 : Aexp)
  | AMult (a1 a2 : Aexp).

(* Semantics Aexp *)
Fixpoint Aeval (st : State) (a : Aexp) : Z :=
  match a with
  | ANum n => Neval n
  | AId x => st x
  | APlus a1 a2 => (Aeval st a1) + (Aeval st a2)
  | AMinus a1 a2  => (Aeval st a1) - (Aeval st a2)
  | AMult a1 a2 => (Aeval st a1) * (Aeval st a2)
  end.

(* We want a string and numeral to be recognized as Aexp to do
   calculations *)
Coercion AId : string >-> Aexp.
Coercion ANum : Num >-> Aexp.

(* Syntax Bexp *)
Inductive Bexp : Type :=
  | BTrue
  | BFalse
  | BEq (a1 a2 : Aexp)
  | BLe (a1 a2 : Aexp)
  | BNot (b : Bexp)
  | BAnd (b1 b2 : Bexp).

(* Semantics Bexp *)
Fixpoint Beval (st : State) (b : Bexp) : bool :=
  match b with
  | BTrue       => true
  | BFalse      => false
  | BEq a1 a2   => (Aeval st a1) =? (Aeval st a2)
  | BLe a1 a2   => (Aeval st a1) <=? (Aeval st a2)
  | BNot b1     => negb (Beval st b1)
  | BAnd b1 b2  => andb (Beval st b1) (Beval st b2)
  end.

(* This is the mapping to tt and ff *)
Definition bool_to_bexp (b : bool) : Bexp :=
  if b then BTrue else BFalse.
Coercion bool_to_bexp : bool >-> Bexp.

(* More notation, it is not needed to understand how this works
   *)
Bind Scope while_scope with Aexp.
Bind Scope while_scope with Bexp.

Notation "x + y" := (APlus x y) (at level 50, left associativity)
  : while_scope.
```

```
Notation "x - y" := (AMinus x y) (at level 50, left associativity)
  : while_scope.
Notation "x * y" := (AMult x y) (at level 40, left associativity)
  : while_scope.
Notation "x <= y" := (BLe x y) (at level 70, no associativity)
  : while_scope.
Notation "x = y" := (BEq x y) (at level 70, no associativity)
  : while_scope.
Notation "x && y" := (BAnd x y) (at level 40, left associativity)
  : while_scope.
Notation "'~' b" := (BNot b) (at level 75, right associativity)
  : while_scope.

(* Instead of writing Aeval and Beval, the writing style in the
   book can be used *)
Notation " 'A[[' a ']]' st " := (Aeval st a)
        (at level 90, left associativity).
Notation " 'B[[' b ']]' st " := (Beval st b)
        (at level 90, left associativity).
Notation " 'N[[' n ']]'" := (Neval n)
        (at level 90, left associativity).

(* Syntax Statements *)
Inductive Stm : Type :=
  | ass (x : string) (a : Aexp)
  | skip
  | comp (s1 s2 : Stm)
  | if_ (b : Bexp) (s1 s2 : Stm)
  | while (b : Bexp) (s : Stm).

(* More notations, when defining proofs capitals are used *)
Notation "x '::=' a" :=
  (ass x a) (at level 60).
Notation "'SKIP'" :=
   skip.
Notation "s1 ; s2" :=
  (comp s1 s2) (at level 80, right associativity).
Notation "'WHILE' b 'DO' s " :=
  (while b s) (at level 80, right associativity).
Notation "'IF_' b 'THEN' s1 'ELSE' s2 " :=
  (if_ b s1 s2) (at level 80, right associativity).

(* The transitions use configurations.
   In NS there are only general configurations <S, s> and
   terminal configurations s.
   In SOS there are running configurations -> <S, s> *)
Inductive Config: Type :=
|Running (S:Stm) (s:State)
|Final (s:State).

(* We always want a state to be recognized as a configuration *)
Coercion Final: State >-> Config.
Notation "'≪' s '≫' " := (Final s).
Notation "'≪' S ',' st '≫'" := (Running S st).

(* Some standard names for variables to use in proofs *)
Definition w : string := "w".
Definition x : string := "x".
Definition y : string := "y".
Definition z : string := "z".
```

B.2 The basics of SOS formalisation, needed for students to understand the proofs

```
(* First, we need some imports from the Coq library *)
Set Warnings "-notation-overridden,-parsing".
From Coq Require Import Bool.Bool.
From Coq Require Import Init.Nat.
From Coq Require Import Arith.Arith.
From Coq Require Import Arith.EqNat.
From Coq Require Import omega.Omega.
From Coq Require Import Lists.List.
From Coq Require Import Strings.String.
From Coq Require Import Logic.FunctionalExtensionality.
From Coq Require Import BinNat.
Require Import tut_common.
Import ListNotations.

(* The language While uses integers, we need to open the Z scope.
   We also open the scope where the statements are defined in. *)
Local Open Scope Z_scope.
Open Scope while_scope.

(*SOS frameworks*)
Definition w : string := "w".
Definition x : string := "x".
Definition y : string := "y".
Definition z : string := "z".

Reserved Notation " conf '==>' conf' "
                  (at level 99).

Inductive sstep : Config -> Config -> Prop:=
|SSSkip : forall  st,
  ≪SKIP, st≫ ==> st

|SSAss: forall x a n st,
    Aeval st a = n ->
    ≪(x::=a), st≫ ==> ≪(x !-> n, st)≫(*t_update st x n)*)

(*don't forget - a derivation tree is written bottom up but read
   top-down.
This is reading*)
|SSCompI: forall S1 S1' S2 st st',
  ≪S1, st≫ ==> ≪S1', st'≫ ->
    ≪S1; S2, st≫ ==> ≪S1'; S2, st'≫

|SSCompII:  forall S1  S2 st st',
  ≪S1, st≫ ==> Final st' ->
  ≪S1;S2, st≫ ==> ≪S2, st'≫

|SSIftrue: forall b S1 S2 st,
  Beval st b = true ->
  ≪IF_ b THEN S1 ELSE S2, st≫ ==> ≪S1, st≫

|SSIffalse: forall b S1 S2 st,
  Beval st b=  false ->
  ≪IF_ b THEN S1 ELSE S2, st≫ ==> ≪S2, st≫

|SSWhile: forall b S st,
  ≪WHILE b DO S, st≫ ==>
       ≪IF_ b THEN (S; WHILE b DO S) ELSE SKIP, st≫
  where " conf '==>' conf' " := (sstep conf conf').


(* The SOS version of stm_eq for one step*)
Theorem stm_eq:
  forall S s conf' conf'',
  ≪S,s≫==> conf' ->
  conf' =conf'' ->
  ≪S,s≫ ==> conf''.
Proof.
intros.
rewrite H0 in H.
apply H.
Qed.

(* Use the tactic from Experiment_NS *)
Ltac eq_states :=
  apply functional_extensionality; intros; unfold t_update; simpl
   ;
  repeat match goal with
  |- context [eqb_string ?v ?x] =>
    destruct (eqb_string v x)
  end;
  reflexivity.
Close Scope while_scope.

(* For a running configuration *)
Theorem conf_eq_rn:
  forall S s s',
  s = s' -> ≪S,s≫ = ≪S,s'≫.
Proof.
intros.
rewrite H.
reflexivity.
Qed.

(* And for a final one *)
Theorem conf_eq_fn:
  forall s s',
  s = s' -> ≪s≫ = ≪s'≫.
Proof.
intros.
rewrite H.
reflexivity.
Qed.

(* Relation *)
Definition relation (X : Type) := X -> X -> Prop.
```

```
(* Making many steps *)
Inductive multi {X : Type} (R : relation X) : relation X :=
  | multirefl : forall (x : X), multi R x x
  | multistep : forall (x y z : X),
                    R x y ->
                    multi R y z ->
                    multi R x z.

Notation " t '==>*' t' " := (multi sstep t t') (at level 40).

(* The SOS version of stm_eq for many steps *)
Theorem no_stm_eq:
  forall s conf' conf'',
  ≪s≫==>* conf' ->
  conf' =conf'' ->
  ≪s≫ ==>* conf''.
Proof.
intros.
rewrite H0 in H.
apply H.
Qed.

(* Making K steps *)
Inductive multi_k {X : Type} (R : relation X) : nat -> relation X
   :=
  | multikrefl : forall (x : X), multi_k R 0 x x
  | multikstep : forall (x y z : X) (k:nat),
                    R x y->
                    multi_k R k y z ->
                    multi_k R (S k) x z.
Notation " t ==>[ k ]  t' " := (multi_k sstep k t t' ) (at level
   50).
Print multi_k.


(* Some simple exersice - how do we used comp1 and comp2 *)

Example triv_1:
≪ x::= (1+3); SKIP, (y!->3, z!->3)≫ ==> ≪SKIP, x!->4, y!->3, z!->3≫.

Proof.
 - eapply SSCompII.
   + eapply SSAss. reflexivity.
```

```
Qed.

(* When will we need to use comp1? *)
Example triv_7_lecture'':
≪ (z ::=x; x ::= y); y ::=z, (x!->3, y!->4,  z!->5)≫ ==>*
    ≪ (y!->3, x!->4, z!->3)≫.
Proof.
  - eapply multistep with ≪x::=y; y ::=z,  z!->3, x!->3, y!-> 4, z!->
   5≫.
    + eapply SSCompI. (*<-- Used Comp 1! *)
      * apply SSCompII. apply SSAss. reflexivity.
      (* We can strictly say what the resulting state needs to be
    *)
    + apply multistep with ≪y ::=z, x!->4, z!-> 3, x!->3, y!->4,  z!->
   5≫.
      * apply SSCompII. apply SSAss. reflexivity.
        (*...or let coq guess it*)
      * eapply multistep.
         apply SSAss. reflexivity.
   eapply no_stm_eq.
   eapply multirefl. apply conf_eq_fn. eq_states.
   Qed.

(*----------*)
(* And now for the big bad induction on the derivation sequence
   - from the slides, and from the homework*)
Local Close Scope Z_scope.
Close Scope while_scope.
(* First we need some tiny lemmas to tell Coq obvious things*)

Lemma zero_steps_rev:
forall c c',
c = c' ->
c ==>[0] c'.
Proof.
intros. induction H.
apply multikrefl.
Qed.

Lemma one_step:
forall c c',
  c ==> c' ->
  c ==>[1] c'.
```

```
Proof.
intros.
apply (multikstep sstep c c').
assumption. apply zero_steps_rev.
reflexivity.
Qed.

(* And a small definition of what is a stuck configuration
   - it is not only a Final one*)

Definition isStuck (conf: Config) :=
0=0 -> ~ (exists (conf':Config), conf ==> conf').

Lemma final_is_stuck:
forall s,
 isStuck ≪s≫ .
Proof.
intros.
unfold isStuck.
intros.
intro.
destruct H0.
inversion H0.
Qed.

Lemma stuck_stops:
forall c c' k,
  isStuck c ->
  c ==>[k] c' ->
  k = 0 /\ c' = c.
Proof.
intros.
unfold isStuck in H.
induction k.
– inversion H0; subst.
 split. reflexivity. reflexivity.
– destruct H0. split. reflexivity. reflexivity.
  destruct H. reflexivity.
  exists y0. assumption.
Qed.

(* We will use strong induction, without bothering to prove it.*)
```

```
Check nat_ind.
Theorem strong_induction:
forall P : nat -> Prop,
  P 0 ->
  (forall n : nat, (forall k : nat, ((k <= n)%nat -> P k)) -> P (S
   n))
  ->
  forall n : nat, P n.
Proof.
Admitted.

(* The example from the slides
   - with this one you can always check the natural lanugage
   proof there *)

(*
For all statements S1 and S2 and for all states s and s':
  if ≪S1;S2,s≫ ==>[k]s''
  then there exists a state s'and natural numbers k1 and k2
    such that
    ≪S1;s≫ ==>[k1] s' and ≪S2;s'≫ ==>[k2] s''
   where k = k1 + k2
*)
Lemma comp_complete:
forall k Ss1 Ss2 s s'',
  ≪Ss1; Ss2, s≫ ==>[k] ≪s''≫
  ->
  exists s' k1 k2,
    ≪Ss1,s≫==>[k1] ≪s'≫ /\ ≪Ss2,s'≫ ==>[k2] ≪s''≫ /\ k = (k1
   + k2).
Proof.
(*induction on the length of the derivation seqence*)
induction k using strong_induction.
- (*0*)
  intros.
  inversion H.
- (*IS - we can use one compf rule or the other*)
  rename k into k0.
  intros.
  inversion H0. subst.
  inversion H2; subst.
  +(*compI*)
```

```
   assert (exists (s' : State) (k1 k2 : nat),
      ≪ S1', st' ≫ ==>[ k1] ≪ s' ≫ /\
      ≪ Ss2, s' ≫ ==>[ k2] ≪ s'' ≫ /\ k0 = k1 + k2).
     {apply H.
       * reflexivity. (*omega.*)
       * assumption.
     }
   destruct H1. rename x0 into s0. destruct H1. rename x0 into k1
   .
   destruct H1. rename x0 into k2. destruct H1. destruct H4.
   exists s0.
   exists (S k1).
   exists k2.
   split.
    -- apply (multikstep sstep≪ Ss1, s ≫ ≪ S1', st' ≫ ).
       assumption. assumption.
    -- split.
    --- assumption.
    --- omega.
  +(*compII*)
    exists st'.
    exists 1.
    exists k0.
    split.
    * apply one_step. assumption.
    * split.
      assumption.
      reflexivity.
Qed.

(* This example is from the homework - as it seems, the
   natural lanugage proof is online too. However, plase try to do
    it
   yourself before you consult it - the question here is will the
   Coq file make it easier as is. *)

(* For all statements S1 and S2, all states s and s' and all
   natural numbers k:
   if ≪S,s≫==>[k] s' then ≪S1;S2,s≫ ==>[k] ≪S2,s'≫ *)

Definition property_half_comp(k:nat) :=
(forall S1 s s',
    ≪S1,s≫ ==>[k] ≪s'≫
  ->
  forall S2,
   ≪S1; S2, s≫ ==>[k] ≪S2, s'≫).

(*2.21*)
Theorem half_comp:
forall k,
 forall S1 s s',
  ≪S1,s≫ ==>[k] ≪s'≫
  ->
  forall S2,
   ≪S1; S2, s≫ ==>[k] ≪S2, s'≫.
Proof.
intro.
(* We only want k in the context to do induction over the whole
  thing on k*)
induction k using strong_induction.
 - intros. inversion H.
 - rename k into k0.
   intros.
   inversion H0. subst.
   induction y0.
   (* if it is compi or compii in other words
    if S1 is executed in many or 1 steps *)
   + rename s0 into s''.
     rename S into S1'.
     apply (multikstep sstep≪ S1; S2, s ≫ ≪ S1';S2, s'' ≫).
     * apply SSCompI. assumption.
     * apply H. reflexivity. assumption.
   + rename s0 into s''.
     apply (multikstep sstep≪ S1; S2, s ≫ ≪S2, s'' ≫).
     * apply SSCompII. assumption.
     * assert (k0 = 0 /\ ≪s'≫ = ≪s''≫).
     { apply stuck_stops. apply final_is_stuck. assumption.
     }
     inversion H1. inversion H5. subst. apply zero_steps_rev.
  reflexivity.
Qed.
```