Radbound University

A Formalization of Natural Semantics of Imperative Programming Languages in Coq

Loes Kruger (s1001459)

Bachelor thesis in Artificial Intelligence
Radboud University Nijmegen

dr. E.M.G.M. (Engelbert) Hubbers
Digital Security,
Institute for Computing
and Information Sciences

dr. F. (Freek) Wiedijk
Software Science,
Institute for Computing
and Information Sciences

July 9, 2020

Abstract

In this thesis, we construct a formalization of natural semantics of imperative programming languages in Coq. Specifically to use in the course Semantics and Correctness. We use already existing formalizations, like Software Foundations [13], to make a formalization that matches the theory and notation used in the course. The formalization covers the topics discussed during the course Semantics and Correctness and a few extensions of **While**. Then, we construct non-trivial proofs using the framework to show how it should be used. Lastly, we conducted an experiment to test whether the formalization can help students of the course Semantics and Correctness to better understand the proofs. Due to small sample size there are inconclusive results on whether the formalization actually helps the students to better understand the proofs. However, there are many possible improvements possible to make the formalization more useful.

Contents

Chapter 1

Introduction

1.1 Problem

It is difficult to build reliable software. Semantics is a field concerned with the meaning of programming languages. The semantics of a programming language can be used to analyze the correctness of the language and increase the reliability of software. Therefore, it is useful to learn how to perform proofs with semantics. The three main types of semantics are operational, denotational and axiomatic semantics. We will focus on operational semantics in this thesis.
Operational semantics is a type of semantics concerned with how the effect of a computation is produced. Natural semantics and structural operational semantics are examples of operational semantics. Natural semantics is concerned with big steps while structural operational semantics is concerned with small steps.
At Radboud University, there is a course called Semantics and Correctness (S&C). The course used the book *Semantics with Applications: A formal introduction* by Nielson and Nielson [10] as course material. In S&C several types of semantics are discussed: natural semantics, structural operational semantics and axiomatic semantics. Unfortunately, many students consider the proofs in this course to be difficult. This is partially caused by the fact that the proofs may use several levels of induction, but also because the book by Nielson and Nielson often leaves out details.
The students that follow the S&C course have also followed the course Logic and Applications (L&A). This course introduces them to Coq. However, in S&C Coq is not used yet. The idea is that having a Coq formalization of the theory of the course, together with proofs of the theorems and exercises used in the course could help students to better understand the proofs.
The idea is not new and many formalizations already exist. For example, the book Software Foundations [13] includes chapters on a formalization of operational semantics in Coq. The notation differs from the notation used in the course Semantics and Correctness but the book can be used to make a formalization that does match the course.

1.2 Research Question

The main research question of this thesis is:
Is it possible to make a formalization of the natural semantics of imperative programming languages in Coq that is based on the theory and notation used in the course Semantics and Correctness and can be used as an educational tool?

This research question consists of two parts.

1. Is it possible to make a Coq formalization of natural semantics of imperative programming languages that follows the theory and notation of the course Semantics and Correctness?
2. Can the formalization be used by students to better understand the proofs in the course?

The first sub-question will be tested by making a formalization and using the formalization to prove theorems and examples from the course.
We hypothesize that it is possible to make a formalization that matches the book better than the formalizations that are currently available. However, the formalization will probably not match the book perfectly. Moreover, we think that it will be possible to formalize all topics discussed in S&C in Coq.

The second sub-question will be tested using an experiment. This sub-question will be discussed in chapter 6.

1.3 Approach

In this thesis, we will formalize the natural semantics of imperative programming languages in Coq following the theory and notation of Nielson and Nielson [10].
There is another student, Elitsa Bahovska, who is doing a closely related thesis, formalizing structural operational semantics in Coq. Both the background chapter and experiment chapter will be done as a group.
The following natural semantics concepts will be formalized in this thesis:

1. The basic setup of natural semantics: $[\text{ass}_{\text{ns}}]$, $[\text{skip}_{\text{ns}}]$, $[\text{comp}_{\text{ns}}]$, $[\text{if}^{\text{tt}}_{\text{ns}}]$, $[\text{if}^{\text{ff}}_{\text{ns}}]$, $[\text{while}^{\text{tt}}_{\text{ns}}]$, $[\text{while}^{\text{ff}}_{\text{ns}}]$.
2. Semantic equivalence and induction on the shape of the derivation tree.
3. Determinism for natural semantics.
4. Hoare logic to prove soundness and completeness.
5. Soundness: If a partial correctness property can be proven using the axiomatic inference system then it does indeed hold according to the semantics in question.
6. Completeness: If some partial correctness property does hold according to the semantics in question, then we can also find a proof for it using the axiomatic inference system.

7. Blocks and procedures with different scoping rules: An extension of the language **While** including variable and procedure declarations.

We will also discuss some extensions of the language **While** that are not covered in the course Semantics and Correctness but are interesting to formalize. These extensions will be non-determinism and the break and continue statements.
After the formalization, non-trivial proofs will be constructed that S&C students can use to increase their understanding of the proofs performed in the course. To make the material easier to understand, the formalization should be compact and easy to use. Therefore, trivial steps should be easily solvable and should not require most of the proving time. Finally, we will run an experiment with S&C students to test whether the formalization helps them understand the course material better and how the formalization could be improved.

1.4 Related work

The idea of having a Coq formalization of natural semantics is not new and quite some work has already been done in this area. A formalization in Coq has already been made in Software Foundations by Pierce [13], however this does not follow the theory and notation used in the course. Formalizations in Isabelle also exist. For example the formalizations by Schirmer [14] and Nipkow and Klein [11]. However, the S&C students are already familiar with Coq so it is preferred that the formalization is in Coq. Moreover, there are other problems with these formalizations. For example: different notations, examples and programming languages are used, the formalization is not written as a textbook or not all parts of the course are formalized.

1.5 Contribution

There is no Coq formalization of natural semantics that matches the theory and notation used in the course Semantics and Correctness. This is the novelty that this thesis will bring. Moreover, all the S&C topics will be formalized. Topics like blocks and procedures are not formalized in other Coq formalizations. Furthermore, we will also provide a series of examples for the students to use and we will test whether the formalization actually has useful applications as an educational tool. Finally, the entire Coq code for the formalization as well as the proven theorems and examples used in the course S&C are available at `https://github.com/lkruger27/NS_SOS_Formalization`.

Chapter 2

Background

Before we explain the formalization of natural semantics, we will summarize important concepts used in S&C: the language **While**, natural semantics, induction and Coq.

2.1 While

While is the example programming language used in Nielson and Nielson [10]. The proofs performed in the book concern this language, therefore the formalization will also concern this language. We will quickly describe the language here.
While is a simple imperative language, it consists of syntactic categories and the meta-variables that range over constructs of each category. The syntactic categories and their meta-variables can be found in Table 1.

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**

Table 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 Table 2.

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

Table 2: The structure of the constructs

2.2 Semantics of expressions

Syntax describes what programs should look like. Semantics describe the mathematical meaning of a program. Semantic functions provide the semantics of **While** for each of the syntax categories. The semantics for **Num**, **Aexp** and **Bexp** are equivalent for both structural operational semantics and natural semantics. However, the semantic functions for **Stm** depend on the type of semantics.

2.2.1 The semantics of Num

The syntax definition for **Num** is $n ::= 0 \mid 1 \mid n\ 0 \mid n\ 1$. The function $\mathcal{N}$: **Num** $\rightarrow$ **Z** is used to express the value of a numeral n as an integer. The semantics of **Num** are defined in Table 3.

$$\begin{gathered}
\mathcal{N}[\![0]\!] = 0 \\
\mathcal{N}[\![1]\!] = 1 \\
\mathcal{N}[\![n0]\!] = 2 \cdot \mathcal{N}[\![n]\!] \\
\mathcal{N}[\![n1]\!] = 2 \cdot \mathcal{N}[\![n]\!] + 1
\end{gathered}$$

Table 3: The semantics of numerals

2.2.2 The semantics of Aexp

The syntax definition for **Aexp** is $a ::= n \mid x \mid a_1 \texttt{+} a_2 \mid a_1 \star a_2 \mid a_1 \texttt{-} a_2$. To define a semantic function that expresses the value of a numeral or variable as an integer, we need states. The states are needed because the value of a variable may be changed by statements. This means that the value of the variable is not necessary equal in every step of the program. A state is a function s: **Var** $\rightarrow$ **Z** which assigns the value of a variable at a certain moment. The function $\mathcal{A}$: **Aexp** $\rightarrow$ (**State** $\rightarrow$ **Z**) is used to express the value of an arithmetic expression as an integer. The semantics of **Aexp** are defined in Table 4.

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

Table 4: The semantics of arithmetic expressions

2.2.3 The semantics of Bexp

The syntax definition for **Bexp** is $b ::= \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg\, b \mid b_1 \wedge b_2$. The function $\mathcal{B} : \textbf{Bexp} \rightarrow (\textbf{State} \rightarrow \textbf{T})$ is used to express the value of a boolean expression as **T**, where $\textbf{T} = \{\textbf{tt}, \textbf{ff}\}$. The semantics of **Bexp** are defined in Table 5.

$$\mathcal{B}[\![\texttt{true}]\!]s = \textbf{tt}$$

$$\mathcal{B}[\![\texttt{false}]\!]s = \textbf{ff}$$

$$\mathcal{B}[\![a_1 = a_2]\!]s = \begin{cases} \textbf{tt} & \text{if } \mathcal{A}[\![a_1]\!]s = \mathcal{A}[\![a_2]\!]s, \\ \textbf{ff} & \text{if } \mathcal{A}[\![a_1]\!]s \neq \mathcal{A}[\![a_2]\!]s, \end{cases}$$

$$\mathcal{B}[\![a_1 \leq a_2]\!]s = \begin{cases} \textbf{tt} & \text{if } \mathcal{A}[\![a_1]\!]s \leq \mathcal{A}[\![a_2]\!]s, \\ \textbf{ff} & \text{if } \mathcal{A}[\![a_1]\!]s > \mathcal{A}[\![a_2]\!]s, \end{cases}$$

$$\mathcal{B}[\![\neg b]\!]s = \begin{cases} \textbf{tt} & \text{if } \mathcal{B}[\![b]\!]s = \textbf{ff}, \\ \textbf{ff} & \text{if } \mathcal{B}[\![b]\!]s = \textbf{tt}, \end{cases}$$

$$\mathcal{B}[\![b_1 \wedge b_2]\!]s = \begin{cases} \textbf{tt} & \text{if } \mathcal{B}[\![b_1]\!]s = \textbf{tt} \text{ and } \mathcal{B}[\![b_2]\!]s = \textbf{tt}, \\ \textbf{ff} & \text{if } \mathcal{B}[\![b_1]\!]s = \textbf{ff} \text{ or } \mathcal{B}[\![b_2]\!]s = \textbf{ff}, \end{cases}$$

Table 5: The semantics of boolean expressions

2.3 Natural Semantics

The statements in **While** are able to modify the state the program is in. Operational semantics is mostly concerned with how the execution of a statement modifies the state. As mentioned before, natural semantics is a type of operational semantics. Natural semantics describes how the overall results of the executions are obtained. It uses big steps while structural operational semantics, another type of operational semantics, uses small steps. Structural operational semantics will also be formalized in Coq in a parallel thesis by Elitsa Bahovska [1]. The transition system specifies the meaning of statements using two types of configurations:

General configurations represent that statement S is executed from state s: $\langle S,\ s\rangle$.
Terminal configurations represent a final state: s'.

The transition relations describe how the steps are executed. For natural semantics the transition relations will specify the relationship between the initial and final state. If S is executed in s and this leads to state s', the transition is written down as $\langle S,\ s\rangle \rightarrow s'$. The natural semantics for **While** can be found in Table 6.

$[\text{ass}_{\text{ns}}]$	$\langle x := a, s\rangle \to s[x \mapsto \mathcal{A}[\![a]\!]s]$	
$[\text{skip}_{\text{ns}}]$	$\langle \texttt{skip}, s\rangle \to s$	
$[\text{comp}_{\text{ns}}]$	$\dfrac{\langle S_1, s\rangle \to s' \quad \langle S_2, s'\rangle \to s''}{\langle S_1; S_2, s\rangle \to s''}$	
$[\text{if}^{\text{tt}}_{\text{ns}}]$	$\dfrac{\langle S_1, s\rangle \to s'}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s\rangle \to s'}$	if $\mathcal{B}[\![b]\!]s = \mathbf{tt}$
$[\text{if}^{\text{ff}}_{\text{ns}}]$	$\dfrac{\langle S_2, s\rangle \to s'}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s\rangle \to s'}$	if $\mathcal{B}[\![b]\!]s = \mathbf{ff}$
$[\text{while}^{\text{tt}}_{\text{ns}}]$	$\dfrac{\langle S, s\rangle \to s' \quad \langle \texttt{while}\ b\ \texttt{do}\ S, s'\rangle \to s''}{\langle \texttt{while}\ b\ \texttt{do}\ S, s\rangle \to s''}$	if $\mathcal{B}[\![b]\!]s = \mathbf{tt}$
$[\text{while}^{\text{ff}}_{\text{ns}}]$	$\langle \texttt{while}\ b\ \texttt{do}\ S, s\rangle \to s$	if $\mathcal{B}[\![b]\!]s = \mathbf{ff}$

Table 6: The natural semantics for **While**

2.4 Induction

2.4.1 Mathematical induction

Mathematical induction is a mathematical proving technique. It can be used to prove that a certain property $P(n)$ holds for all natural numbers [3]. Induction has two steps: the base case and the induction step. In the base case we prove that the property holds for a first element, usually 0 or 1. In the inductive step we prove that if the property holds for some number k, then it also holds for $k + 1$. In this case, $P(k)$ is called the induction hypothesis.

2.4.2 Structural induction

Structural induction is another type of induction. It can be used to prove that a certain property P holds for all elements [3]. Structural induction consists of the induction basis and the inductive step. In the induction basis we prove that P holds for all minimal elements. In the inductive step we prove that P holds for a complex element if property P holds for all immediate substructures. The induction hypothesis is that P holds for all immediate substructures.
Structural induction can be used to prove that we end up in the final state s' after execution of statement S in initial state s [10]. The base case consists of the axioms. The immediate substructures are the premises, they are written above the line in the natural semantics rules. By proving that the property holds for all immediate substructures, we can prove that execution in the initial state will lead to the final state.

2.5 Coq

Logic is a field concerned with making proofs. Proof assistants are software tools that help construct logical proofs, however, the user still needs to guide the proof. Coq is a well known proof assistant. The S&C students have already been introduced to Coq in a previous course. Moreover, a textbook called Software Foundations [13] is based on proofs in Coq and it includes a formalization of natural semantics that can be used to make our formalization. Therefore, we will use Coq to make the formalization of natural semantics. We assume the reader is familiar with Coq, otherwise we suggest to go through Software Foundations [13].

2.5.1 Tactics

In the course Logic and Applications, the students worked on a special server that used different tactics. The formalization of natural semantics will not be on this server. In this subsection, we will give a quick overview of tactics used later on. More information and tactics can be found at the Coq tactics webpage [9] or Software Foundations [13].

`reflexivity.` If the goal looks like an equation that is the equal on both sides, e.g. $1 = 1$, then reflexivity finishes the proof. It can also be the case that the sides are complex and do not look equal, but they are equal after some simplifications.
`apply.` This can be used to prove the goal using a hypothesis or lemma. We can also use the apply on hypothesis in the context by writing `apply ... in H` where H represents the hypothesis. Sometimes a value for a variable needs to be provided to apply a lemma or hypothesis because Coq cannot determine the value using pattern matching. This can be done by writing `apply ... with ...`.
`intros.` This can move hypothesis or variables in the goal to the context. For example, if there is a quantification or an implication in the goal.
`simpl.` This simplifies the goal, but it can also be used in the context by writing `simpl in H`. The tactic `reflexivity` can also simplify the goal if it is immediately provable after simplification.
`rewrite.` This can be used to rewrite the goal using a lemma or hypothesis that has an equality. It can also be used in the context. It is possible to give the rewriting direction using `rewrite -> H` or `rewrite` <– `H`.
`symmetry.` This swaps the sides of an equality, e.g. $x = y$ becomes $y = x$.
`split.` This is used with conjunctions. It splits the conjunction into two subgoals that can be proven independently.
`subst.` This substitutes away all assumptions of form $x = y$ or $y = x$ for a variable x. It can be used with a variable as input or without. Without a given variable it substitutes away as much as possible.
`inversion.` This tries to find out as much about a hypothesis and use that to generate more hypothesis in the context. It can be used on inductive types. It can detect that certain cases of inductive types cannot apply and others can apply. `subst` is often used immediately after inversion to make the context more readable.
`generalize dependent.` This moves a variable from the context back to the goal. For example, when we used an `intros` to eliminate a forall over variables m and n

and we want to put the forall back but only for variable m.
`assert`. This can be seen as a local lemma. It needs to be proven but after that it can be used as a hypothesis.
`assumption`. This can be used to apply a hypothesis that is in the context without explicit naming the hypothesis. Coq tries to find a hypothesis in the context that can be used to solve the current goal.
`discriminate`. It uses the fact that constructors of inductively defined types are disjoint. It is applied on a hypothesis involving an equality between different constructors and immediately proves the goal.
`unfold`. This replaces a definition name in the goal by its definition.
`induction`. This will generate a subgoal for all values of inductively defined type.
`destruct`. This performs a case analysis on values of inductively defined type.

There are also special automation tactics. These can be used to make proofs shorter:
`auto`. This can apply a sequence of tactics that are a combination of `intros` and `apply` on hypotheses in the context. This means we do not have to search all the hypotheses by ourselves.
`eapply`. Sometimes it is necessary to provide an extra argument when applying certain hypotheses or lemmas. However, soon after the application it will be obvious which argument should be added. Instead of using `apply` we can write `eapply` and Coq will use a variable that is not yet used as placeholder. If Coq has found the value it updates the goal. We will later see that this is useful when building a proof tree for natural semantics. Multiple tactics can be applied after each other by typing `;` in between the tactics. Using the keyword `try` before tactics indicates that the tactics will be tried but if they cannot be applied they will not give an error. This is useful when many subgoals have a similar but not completely equal proof.

2.5.2 Structuring a proof

In Logic and Applications, there was no structuring of the proof because an older version of Coq was used. However, in recent versions of Coq you can easily give levels to a proof by using `-,+,*,{}`. Every bullet can be used for a different subgoal. This structuring is not needed, but it helps for readability. Usually, the first level uses $-$, the second $+$ and the third $*$. Big proofs might require more levels, this can be done using $\{\}$. In $\{\}$, we can use $-$, $+$, $*$ again. The following example taken from chapter *Basics* of Software Foundations [13] shows how this structuring works:

```
Theorem andb_commutative : forall b c, andb b c = andb c b.
Proof.
  intros b c. destruct b eqn:Eb.
  - destruct c eqn:Ec.
    + reflexivity.
    + reflexivity.
  - destruct c eqn:Ec.
    + reflexivity.
    + reflexivity.
Qed.
```

Chapter 3

Formalization of the framework

The framework describes the definitions, functions and notation needed to perform the proofs. The framework is ordered like the book by Nielson and Nielson [10] and a similar structure is in the next chapter, application of the framework where we will prove theorems and exercises performed in the course.

3.1 Semantics of Expressions

We have already seen the structure of the constructs of **While**. Software Foundations also has a chapter about imperative programming languages, this chapter is called *Simple Imperative Programs*. In the Software Foundations book they use the language **Imp** which uses almost the same categories but with different names. For instance, **Aexp** is called **aexp**, and **Stm** is called **com**. They also use natural numbers instead of numerals.

3.1.1 Num

Remember that the syntactic definition for **Num** is n ::= 0 | 1 | n0 | n1. We can write this syntactic definition in Coq using `Inductive`. `Inductive` indicates that we are defining a new set of data values, or a type, and in this case we name our type `Num`:

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

The semantics should then use the syntax to evaluate the value of the numeral to an integer value, or `Z` in Coq. To work with `Z` we need to open the `Z Scope` in Coq.

```
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 add a coercion, to interpret members of **Num** as integers directly:

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

However, there is a problem with this definition of **Num**. To represent the integer `3`, we would need to write `NOdd NOne` in Coq. Moreover, the bigger the integer the longer the numerical representation for that integer. We would like to write digits to represent **Num** instead of this complicated representation using `NZero`, `NOne`, `NEven` and `NOdd`. This means we also need a coercion from **Z** to **Num**.

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

We still need to define the function `z_to_num` which should take an integer and return a numeral. For this, we need a further understanding of the type `Z` in Coq. It has three constructors `Z0`, `Zpos` and `Zneg`. Here `Z0` represents the integer 0 and the other `constructors` take an argument of the type `positive`.

```
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 already defined in the standard library of Coq [8]. The type is similar to **Num**, it builds strictly positive integers in a binary way. However, `positive` has only three `constructors`, it does not have `NZero`. This makes the definition to convert `positive` to `Num` easy to understand:

```
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.
```

With these definitions it is possible to write `3` as a digit in a proof without any problems.

3.1.2 Var

Var is represented as a string. Strings are imported from the Coq standard library.

3.1.3 State

Before we can define **Aexp**, we need to define the concept **State**. A **State** can be defined as a total map. A total map looks up which value a certain string has and returns that value. Using a general function of a total map, the returned value will be of type `A`. Here, `A` represents an arbitrary type. Using this definition, we can make a total map using different types. For example, a total map that uses natural numbers, integers or lists. If we try to find a value for a string that is not defined, a default value is returned. This is the total map function in Coq:

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

We also have to define a default element in case the key is not in the map. We can do this using a function, or `fun`, that will map any key to a value v if the key is not in the map. This can be represented in Coq as the following:

```
Definition t_empty {A : Type} (v : A) : total_map A :=
  (fun _ => v).
```

In proofs, it is often the case that we start with a state where no variables have a value. We want all variables to have value 0 in the initial state if no other value is given. Therefore, we will be using 0 as default element.

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

We can check if two strings are equal using the following function which uses the function `string_dec` from the standard Coq library:

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

We also need to be able to update the map if some variable gets a new value. This function should take a map, a key and a value and make sure that the key maps to the value and the map remains unchanged for all other keys:

```
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 can introduce notation for updating the state `t_update`, this makes it easier to read and write the state. We also have to define the level and associativity of the notation, it is not important to understand how this works.

```
Notation "x '!->' v ',' m" := (t_update m x v)
  (at level 100, v at next level, right associativity).
```

To update the state beginning with the empty string we use the same notation:

```
Notation "x '!->' v" := (x !-> v, empty_State)
  (at level 100, v at next level, right associativity).
```

Now, we can simply say that the state is a total map of type `Z`:

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

Further explanation can be found in the chapters *Maps* and *Polymorphism* of Software Foundations [13].

3.1.4 Aexp

Remember that the syntactic definition for **Aexp** is a ::= n | x | a_1 + a_2 | $a_1 \star a_2$ | a_1 - a_2. We can write this syntactic definition of **Aexp** in Coq as follows:

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

The semantics of **Aexp** look like this:

```
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.
```

In the book [10], we write $\mathcal{A}[\![a]\!]s$ instead of `Aeval`. We can easily add a notation in Coq to make this similar:

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

This can be used in a computation as follows:

```
Compute (A[[x+1]] (x!->3)).
```

This will give `4` as output in Coq. In the formalization of the framework, we used `Aeval` instead of the short hand notation. This means that the notation can easily be adjusted without having to change the formalization.

3.1.5 Bexp

Similar to **Aexp** we will define the syntax and semantics for **Bexp**. The syntactic definition for **Bexp** is b ::= `true` | `false` | $a_1 = a_2$ | $a_1 \leq a_2$ | $\neg\, b$ | $b_1 \wedge b_2$. The syntax and semantics can be represented in Coq as follows:

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

Here, we also add notation to change `Beval` into $\mathcal{B}[\![b]\!]s$:

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

This can be used in a computation as follows:

```
Compute (B[[x=1+2]] (x!->3)).
```

This will give `true` as output in Coq.

3.1.6 Notation

In the Coq script where this formalization is made, some notations and coercions are added at this point. This makes the proofs easier to read and write. In Software Foundations [13] it is mentioned that understanding of the notations and coercions is not needed. It is also not needed to apply the formalization. Therefore, we will not explain this part of the formalization.

3.2 Natural Semantics

3.2.1 Syntax

In the imperative programming language of Software Foundations, **Imp**, the statements are called commands and the syntax looks a bit different. The syntax for **Imp** looks like this:

c ::= `skip` | $x := a$ | c ; c | `test` b `then` c `else` c `fi` | `while` b `do` c `end`

This is represented in Coq as:

```
Inductive com : Type :=
  | CSkip
  | CAss (x : string) (a : aexp)
  | CSeq (c1 c2 : com)
  | CIf (b : bexp) (c1 c2 : com)
  | CWhile (b : bexp) (c : com).
```

We would like to have the syntax in Coq as similar as possible to the syntax used in Nielson and Nielson. That syntax looks as follows [10]:

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

In Coq, some names or notations are already taken and cannot be used for own definitions or types. For example, `if` is a keyword in Coq so we cannot use it directly. Therefore, we will be using `if_` whenever we want to write `if`. This gives the following syntax for statements in Coq:

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

We can make the reading and writing of the programs easier by introducing notations. First, we need to define a scope and bind it to our **Stm** type. This means that every function that gets arguments of type **Stm** is interpreted in `ns_scope`. Just like with `if`, changing `::=` into `:=` gives problems because `:=` is used in Coq in various places, so we also cannot use `:=`. Moreover, we use capitals in the notation to make the difference between notation and rules clearer. In the end the notation that can be used for proofs looks like this:

```
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).
```

3.2.2 Semantics

The semantics of **Imp** are represented in Coq as the following using the notation

```
Reserved Notation "st '=[' c ']=>' st'"
```

The notation *st* =[c]=> *st′* differs quite a lot from the usual notation $\langle S, s\rangle \rightarrow\ s'$. If we change the Coq notation into the notation used in the book, we would get something similar to `<S, s> -> s'`, this causes problems with the > operator. We can fix this by setting it to a lower level, but this might cause problems in later proofs. Moreover, We cannot use `->` because this is also already used in Coq and will cause errors. Therefore, we changed the notation into `<< S, s >>-->s'`.

In the formalization of structural operational semantics [1] there is a proof that natural semantics is equivalent to structural operational semantics. For that proof the notation between natural semantics and structural operational semantics need to be similar. In the different types of semantics the configurations are used differently. The definition of the configurations that can be used by both formalization is called `Config` is defined as follows

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

Coercion Final: State>->Config.
Notation "'<<' s '>>' " := (Final s).
Notation "'<<' S ',' st '>>'" := (Running S st).
```

More information about the configurations can be found in the formalization of structural operational semantics thesis [1]. To understand the natural semantics formalization it is only necessary to understand that `<< S, s >>` in `<< S, s >>-->s'` is the same as using `Running S s`. The semantics of **While** are represented in Coq as follows:

```
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' ).
```

3.3 Semantic Equivalence

Two statements S_1 and S_2 are semantically equivalent if for all states s and s', $\langle S_1, s\rangle \to s'$ if and only if $\langle S_2, s\rangle \to s'$ holds. There is a formalization of semantic equivalence available in the *Program Equivalence* chapter of Software Foundations [13], we will adapt this slightly to fit our formalization. To prove two statements are equivalent in Coq, we first need to define that two arithmetic expressions are equivalent if they evaluate to the same value:

```
Definition Aequiv (a1 a2 : Aexp) : Prop :=
  forall (st : State),
    Aeval st a1 = Aeval st a2.
```

We also have to define that two boolean expressions are equivalent if they evaluate to the same truth value:

```
Definition Bequiv (b1 b2 : Bexp) : Prop :=
  forall (st : State),
    Beval st b1 = Beval st b2.
```

Lastly, we have to define the equivalence of two statements:

```
Definition Sequiv (S1 S2 : Stm) : Prop :=
  forall (st st' : State),
   << S1 , st >> --> st' <-> << S2 , st >> --> st'.
```

This enables us to prove two statements are semantically equivalent by using following structure:

```
Theorem ExampleSkip : forall (S : Stm),
  Sequiv
    ( SKIP ; S)
    ( SKIP ; SKIP ; S).
```

3.4 Determinism

Semantics are deterministic if for all statements S and states s, s' and s'':

$$\langle\ S\ ,\ s\ \rangle \rightarrow\ s' \text{ and } \langle\ S\ ,\ s\ \rangle \rightarrow\ s'' \text{ imply } s' = s''$$

Natural semantics is deterministic, nothing needs to be added to the framework to prove this. The proof is provided in the next chapter.

3.5 Hoare logic

The chapters concering Hoare logic will not use the **Num** type but natural numbers instead. This is because proofs like the factorial proofs need many mathematical lemmas that are available in the Coq standard library for natural numbers. Adapting all these lemmas to use the **Num** type would be too much work. Moreover, the difference between using **Num** or natural numbers in these chapters is not very noticable.

Axiomatic semantics, also called Hoare logic, is a different type of semantics than natural semantics. This type of semantics is also covered in the chapter *Hoare logic, part I* from Software Foundations [13].
Hoare logic is based on assertions: $\{\ P\ \}\ S\ \{\ Q\ \}$. Here P and Q represent the pre- and postcondition predicates and S represents a statement. We can represent this in Coq as:

```
Definition Assertion := State -> Prop.

Definition hoare_triple
  (P : Assertion) (S : Stm) (Q : Assertion) : Prop :=
  forall st st',
    << S , st >>--> st' ->
     P st ->
     Q st'.

Notation "{{ P }}  c  {{ Q }}" :=
  (hoare_triple P c Q) (at level 90, c at next level)
  : hoare_spec_scope.
```

This should be read as: If predicate P holds in the initial state and statement S terminates on that initial state, then predicate Q must hold in the resulting final state. This does not mean that S should always terminate when executed in initial state P, Hoare logic is about partial correctness. Moreover, using two different states in the postcondition causes problems. By using an abbreviation, we can use predicates and omit the state in the Hoare triples. If values from other states are needed, we can use logical variables. Logical variables are variables that are not used in the program. Therefore, the value of the logical variable will be the same in all states. We need the following rules to use this notation:

$P_1 \wedge P_2$	$P\ s = \begin{cases} \mathbf{tt} & \text{if } P_1\ s = \mathbf{tt} \text{ and } P_2\ s = \mathbf{tt}, \\ \mathbf{ff} & \text{otherwise} \end{cases}$
$P_1 \vee P_2$	$P\ s = \begin{cases} \mathbf{tt} & \text{if } P_1\ s = \mathbf{tt} \text{ or } P_2\ s = \mathbf{tt}, \\ \mathbf{ff} & \text{otherwise} \end{cases}$
$\neg P_1$	$P\ s = \begin{cases} \mathbf{tt} & \text{if } P_1\ s = \mathbf{ff}, \\ \mathbf{ff} & \text{otherwise} \end{cases}$
$P_1[x \mapsto \mathcal{A}[\![a]\!]]$	$P\ s = P_1\ (s[x \mapsto \mathcal{A}[\![a]\!]s])$
$P_1 \Rightarrow P_2$	for all states s: $P_1\ s$ implies $P_2\ s$

Table 7: The rules for predicates

We can represent these rules in Coq as follows:

$P1 \wedge P2$

```
Definition and_sub (P1 P2 : Assertion) : Assertion :=
  fun (st : State) => P1 st /\ P2 st.

Notation "P1/\pP2" := (and_sub P1 P2)
```

$P1 \vee P2$

```
Definition or_sub (P1 P2 : Assertion) : Assertion :=
  fun (st : State) => P1 st \/ P2 st.

Notation "P1\/pP2" := (or_sub P1 P2)
```

$\neg P$

Here we need a trick because $B[\![b]\!]$ does not have a state argument that can be used to evaluate the boolean. The negation symbol should be used before a proposition, but in this case $B[\![b]\!]$ is not a proposition because it is not evaluated yet. Therefore, we use `Bassn2` to make it possible to write the negation there and keep the notation as close as possible to the notation used in the course.

```
Definition bassn b : Assertion :=
  fun st => (Beval st b = true).

Notation " 'B' '[[' b ']]'" := (bassn b)

Definition bassn2 b : Assertion :=
  fun st => ~(Beval st b = true).
```

```
Notation " '~B' '[[' b ']]'" := (bassn2 b)
```

$P[x \mapsto \mathcal{A}[\![a]\!]]$

```
Definition assn_sub x a P : Assertion :=
  fun (st : State) =>
    P (x!->Aeval st a , st).

Notation "P [ x |-> a ]" := (assn_sub x a P)
```

$P1 \Rightarrow P2$

```
Definition assert_implies (P Q : Assertion) : Prop :=
  forall st, P st -> Q st.

Notation "P=>>Q" := (assert_implies P Q)
```

To prove properties of statements in axiomatic semantics we need an inference system. The proof will be an inference tree with $\{ P \}\ S\ \{ Q \}$ as conclusion. The inference rules are defined as follows:

$[\text{ass}_\text{p}]$	$\{ P[x \rightarrow \mathcal{A}[\![a]\!]] \}\ x := a\ \{ P \}$
$[\text{skip}_\text{p}]$	$\{ P \}\ \texttt{skip}\ \{ P \}$
$[\text{comp}_\text{p}]$	$\dfrac{\{ P \}\ S_1\ \{ Q \}, \quad \{ Q \}\ S_2\ \{ R \}}{\{ P \}\ S_1; S_2\ \{ R \}}$
$[\text{if}_\text{p}]$	$\dfrac{\{ \mathcal{B}[\![b]\!] \wedge P \}\ S_1\ \{ Q \}, \quad \{ \neg\mathcal{B}[\![b]\!] \wedge P \}\ S_2\ \{ Q \}}{\{ P \}\ \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \{ Q \}}$
$[\text{while}_\text{p}]$	$\dfrac{\{ \mathcal{B}[\![b]\!] \wedge P \}\ S\ \{ P \}}{\{ P \}\ \texttt{while}\ b\ \texttt{do}\ S\ \{ \neg\mathcal{B}[\![b]\!] \wedge P \}}$
$[\text{cons}_\text{p}]$	$\dfrac{\{ P' \}\ S\ \{ Q' \}}{\{ P \}\ S\ \{ Q \}}$ if $P \Rightarrow P'$ and $Q' \Rightarrow Q$

Table 8: The axiomatic system for partial correctness

The rules can be represented in Coq and proven in Coq. This is useful because by proving that the rule is correct, we can apply the rules to prove a program is correct without having to unfold the Hoare triple definition. The proofs for the rules can be found in chapter *Hoare Logic, Part I* of Software Foundations [13].

```
Theorem assp : forall Q x a,
  {{Q [x |-> a]}} x ::= a {{Q}}.
```

```
Theorem skipp : forall P,
     {{P}} SKIP {{P}}.

Theorem compp : forall P Q R S1 S2,
     {{Q}} S2 {{R}} ->
     {{P}} S1 {{Q}} ->
     {{P}} S1; S2 {{R}}.

Theorem ifp : forall P Q b S1 S2,
  {{B[[b]]/\pP}} S1 {{Q}} ->
  {{~B[[b]]/\pP}} S2 {{Q}} ->
  {{P}} IF_ b THEN S1 ELSE S2 {{Q}}.

Theorem whilep : forall P b c,
  {{B[[b]]/\pP}} c {{P}} ->
  {{P}} WHILE b DO c {{~B[[b]]/\pP}}.

Theorem consp : forall (P P' Q Q' : Assertion) S,
  {{P'}} S {{Q'}} ->
  P=>>P' ->
  Q'=>>Q ->
  {{P}} S {{Q}}.
```

After this, we can start proving assertions with the defined rules.

3.6 Annotated programs

The trees used to prove statements in Hoare logic get big and are difficult to follow. In Semantics and Correctness another way to write a correctness proof is given. This new notation is called an annotated program. To make the annotated program, we start at the bottom precondition predicate of the tree and work our way to the bottom postcondition predicate. In the annotated program, every predicate we encounter should follow from the one above. If there is a statement between predicates, the predicate above the statement indicates the precondition for the statement and the predicate below the statement indicates the postcondition. Consecutive predicates that are equal can be left out of the annotated program. In the chapter *Hoare logic, part II* of Software Foundations [13] they also discuss annotated programs, but there they are called decorated programs. The exact rules of the annotated programs are as follows:

[ass$_\text{p}$]	$\{P[x \to \mathcal{A}[\![a]\!]]\}$
	$x := a$
	$\{P\}$
[skip$_\text{p}$]	$\{P\}$
	`skip`
	$\{P\}$
[comp$_\text{p}$]	$\{P\}$
	S_1
	$\{Q\}$
	S_2
	$\{R\}$
[if$_\text{p}$]	$\{P\}$
	`if` b
	`then`
	$\quad\{\mathcal{B}[\![b]\!] \wedge P\}$
	$\quad S_1$
	$\quad\{Q\}$
	`else`
	$\quad\{\neg\mathcal{B}[\![b]\!] \wedge P\}$
	$\quad S_2$
	$\quad\{Q\}$
	$\{Q\}$
[while$_\text{p}$]	$\{P\}$
	`while` b
	`do`
	$\quad\{\mathcal{B}[\![b]\!] \wedge P\}$
	$\quad S$
	$\quad\{P\}$
	$\{\neg\mathcal{B}[\![b]\!] \wedge P\}$
[cons$_\text{p}$]	$\{P\}$
	$\{P'\}$
	S
	$\{Q'\}$
	$\{Q\}$

Table 9: The rules for annotated programs

To formalize the proofs in Coq, we first have to make a new inductive type `AStm` with embedded assertions. We add the `A` in front of `AStm` to make clear that this is for annotated programs. Every **Statement** should have a precondition and a postcondition with the statement as input to be able to prove that the **Statement** and precondition lead to the postcondition. However, we can leave out some of the pre- and postconditions because they would lead to redundant annotations. Note that in this case we define the [cons_p] rule as two separate rules `pre` and `post`. In Software Foundations [13] this is done all the time and the [cons_p] rule is just a combination of the `pre` and `post` rule. We only add it here because in annotated programs we can leave out predicates that are equal to the predicate directly above. We also create the inductive type `Annotated`, to add one precondition for the entire annotated program.

```
Inductive AStm : Type :=
  | Aass : string -> Aexp ->  Assertion -> AStm
  | Askip :  Assertion -> AStm
  | Acomp : AStm -> AStm -> AStm
  | Aif : Bexp ->  Assertion -> AStm ->  Assertion -> AStm
          -> Assertion-> AStm
  | Awhile : Bexp -> Assertion -> AStm -> Assertion -> AStm
  | Apre : Assertion -> AStm -> AStm
  | Apost : AStm -> Assertion -> AStm.

Inductive Annotated : Type :=
  | annotated : Assertion -> AStm -> Annotated.
```

Now, we can specify the rules for the annotated programs as follows:

```
Notation "x '::=' a {{ P }}"
      := (Aass x a P)
       (at level 60, a at next level) : annot.
Notation "'SKIP' {{ P }}"
      := (Askip P)
       (at level 10) : annot.
Notation " S1 ; S2 "
      := (Acomp S1 S2)
       (at level 80, right associativity)  : annot.
Notation "'IF_' b 'THEN' {{ P }} d 'ELSE' {{ P' }} d' {{ Q }}"
      := (Aif b P d P' d' Q)
       (at level 80, right associativity)  : annot.
Notation "'WHILE' b 'DO' {{ Pbody }} d {{ Ppost }}"
      := (Awhile b Pbody d Ppost)
       (at level 80, right associativity) : annot.
Notation "'=>>' {{ P }} d"
      := (Apre P d)
       (at level 90, right associativity)  : annot.
Notation "d '=>>' {{ P }}"
      := (Apost d P)
       (at level 80, right associativity)  : annot.
Notation "{{ P }} d"
      := (annotated P d)
      (at level 90)  : annot.
```

If we have an annotated program, we want to extract the **Statements** from the program by removing the annotations. This can be done with an extract function:

```
  match a with
  | Aass X y _         => X ::= y
  | Askip _            => SKIP
  | Acomp a1 a2        => (extract a1 ; extract a2)
  | Aif b _ a1 _ a2 _ => IF_ b THEN extract a1 ELSE extract a2
  | Awhile b _ a _     => WHILE b DO extract a
  | Apre _ a           => extract a
  | Apost a _          => extract a
  end.

Definition extract_ann (ann : Annotated) : Stm :=
  match ann with
  | annotated P a => extract a
  end.
```

The postcondition can be found when the precondition and the statement is known. To find the postcondition we can use the following function:

```
Fixpoint post (a : AStm) : Assertion :=
  match a with
  | Aass x y Q               => Q
  | Askip P                  => P
  | Acomp a1 a2              => post a2
  | Aif  _ _ a1 _ a2 Q       => Q
  | Awhile b Pbody c Ppost => Ppost
  | Apre _ a                 => post a
  | Apost c Q                => Q
  end.
```

To find the pre- and postcondition of the complete annotated program the following functions can be used:

```
Definition pre_ann (ann : Annotated) : Assertion :=
  match ann with
  | annotated P a => P
  end.

Definition post_ann (ann : Annotated) : Assertion :=
  match ann with
  | annotated P a => post a
  end.
```

An annotated program is correct if the following definition holds for a given annotated program:

```
Definition ann_correct (ann : Annotated) :=
  {{pre_ann ann}} (extract_ann ann) {{post_ann ann}}.
```

If the above mentioned Hoare triple is valid, an annotated program can be proven using the `ann_correct` function. The Hoare triple is valid if the annotations are logically consistent. The annotations are logically consistent if for every rule the postcondition follows from the precondition and the rule. To prove this, we need a function that describes the verification conditions. If the verification conditions hold then the decorations are logically consistent.

```
Fixpoint verification_cond (P : Assertion) (a : AStm) : Prop :=
  match a with
  | Aass x y Q =>
      (P=>>Q [x |-> y])
  | Askip Q =>
      (P=>>Q)
  | Acomp a1 a2 =>
      verification_cond P a1
     /\ verification_cond (post a1) a2
  | Aif b P1 a1 P2 a2 Q =>
      ((fun st => bassn b st /\ P st)=>>P1)
     /\ ((fun st => (bassn2 b st) /\ P st)=>>P2)
     /\ (post a1=>>Q) /\ (post a2=>>Q)
     /\ verification_cond P1 a1
     /\ verification_cond P2 a2
  | Awhile b Pbody a Ppost =>
      (P=>>post a)
     /\ ((fun st => bassn b st /\ post a st )=>>Pbody)
     /\ ((fun st => (bassn2 b st) /\ post a st)=>>Ppost)
     /\ verification_cond Pbody a
  | Apre P' a =>
      (P=>>P') /\ verification_cond P' a
  | Apost a Q =>
      verification_cond P a /\ (post a=>>Q)
  end.
```

Now the proof that verification conditions are correct.

```
Theorem verification_correct : forall a P,
  verification_cond P a -> {{P}} (extract a) {{post a}}.
Proof.
  induction a; intros P H; simpl in *.
  - (* ass *)
    eapply consp_pre.
    apply assp.
    assumption.
  - (* skip *)
```

```
    eapply consp_pre.
    apply skipp.
    assumption.
  - (* comp *)
    destruct H as [H1 H2].
    eapply compp.
    apply IHa2.
    apply H2.
    apply IHa1.
    apply H1.
  - (* if *)
    destruct H as [HPre1 HPre2].
    destruct HPre2 as [HPre2 Hd1].
    destruct Hd1 as [Hd1 Hd2].
    destruct Hd2 as [Hd2 HThen].
    destruct HThen as [HThen HElse].
    apply IHa1 in HThen. clear IHa1.
    apply IHa2 in HElse. clear IHa2.
    apply ifp.
      + eapply consp_post with (Q':=post a2); eauto.
        eapply consp_pre; eauto.
      + eapply consp_post with (Q':=post a4); eauto.
        eapply consp_pre; eauto.
  - (* while *)
    destruct H as [HPre Hbody1].
    destruct Hbody1 as [Hbody1 Hpost1].
    destruct Hpost1 as [Hpost1 Hd].
    eapply consp_pre; eauto.
    eapply consp_post; eauto.
    apply whilep.
    eapply consp_pre; eauto.
  - (* pre *)
    destruct H as [HP Hd].
    eapply consp_pre.
    apply IHa.
    apply Hd.
    assumption.
  - (* post *)
    destruct H as [Hd HQ].
    eapply consp_post.
    apply IHa.
    apply Hd.
    assumption.
Qed.
```

After this step, we can prove that an annotated program is correct.

```
Definition verification_cond_ann (ann : Annotated) : Prop :=
  match ann with
  | annotated P a => verification_cond P a
  end.

Lemma verification_correct_ann : forall ann,
  verification_cond_ann ann -> ann_correct ann.
Proof.
  intros [P a].
  apply verification_correct.
Qed.
```

To make it even easier, we can use a tactic from Software Foundations chapter *Hoare logic, Part II* [13] that can be applied to an annotated program and will automatically solve most of the proof. The parts that the tactic can not solve are usually parts of the proof where we have to apply the [cons$_\text{p}$] rule and solve some mathematical equations. We had to adjust the tactic a bit because we also adjusted the rules. The definition of this tactic looks very difficult but is not very difficult to understand. It tries to unfold, rewrite and simplify as many things as possible. Everything that can not be solved automatically is left as a subgoal that needs to be proven manually.

```
Tactic Notation "verify" :=
  apply verification_correct;
  repeat split;
  simpl; unfold assert_implies;
  unfold bassn in *;unfold bassn2 in *;
  unfold Beval in *; unfold Aeval in *;
  unfold assn_sub; intros;
  repeat rewrite t_update_eq;
  repeat (rewrite t_update_neq; [| (intro X; inversion X)]);
  simpl in *;
  repeat match goal with [H : _ /\ _ |- _] => destruct H end;
  repeat rewrite not_true_iff_false in *;
  repeat rewrite not_false_iff_true in *;
  repeat rewrite negb_true_iff in *;
  repeat rewrite negb_false_iff in *;
  repeat rewrite eqb_eq in *;
  repeat rewrite eqb_neq in *;
  repeat rewrite leb_iff in *;
  repeat rewrite leb_iff_conv in *;
  try subst;
  repeat
    match goal with
      [st : State |- _] =>
        match goal with
          [H : st _ = _ |- _] => rewrite -> H in *; clear H
        | [H : _ = st _ |- _] => rewrite <- H in *; clear H
        end
    end;
  try eauto; try omega.
```

After this, we can simply try to verify an annotated program by writing:

```
Theorem some_annotated_program_correct :
  ann_correct some_annotated_program.
Proof.
  verify.
Qed.
```

Note that this is useful for proving an annotated program, but it is not very useful for building an annotated program.

3.7 Soundness and Completeness

In the previous chapters about Hoare logic, we specified rules that can be used to prove the **validity** of a specific Hoare triple. If a Hoare triple is valid, we write it as

$$\models_p \{\ P\ \}\ S\ \{\ Q\ \}$$

This means that for all states s and s', if $P\ s = \mathbf{tt}$ and $\langle S,\ s\rangle \to s'$ then $Q\ s' = \mathbf{tt}$. Besides validity, we can also say that there exists an inference tree with the conclusion $\{\ P\ \}\ S\ \{\ Q\ \}$, this is called **provability**. An inference tree is similar to a derivation tree but for Hoare logic. We can write provability as

$$\vdash_p \{\ P\ \}\ S\ \{\ Q\ \}$$

Using these two important concepts, we can introduce the concepts **soundness** and **completeness**.
The inference system is sound: if a partial correctness property can be proven using the inference system rules, then the property holds according to the semantics, or more formally:

$$\models_p \{\ P\ \}\ S\ \{\ Q\ \} \text{ implies } \vdash_p \{\ P\ \}\ S\ \{\ Q\ \}$$

The inference system is complete: If a partial correctness property holds according to the semantics then we can also prove the property using the inference system, or more formally:

$$\vdash_p \{\ P\ \}\ S\ \{\ Q\ \} \text{ implies } \models_p \{\ P\ \}\ S\ \{\ Q\ \}$$

To formalize soundness and completeness in Coq, we first have to formalize provability. We formalize provability in a similar way as is done in the chapter *Hoare logic as a logic* of Software Foundations [13]. We need to define a new inductive type. In this inductive type we want to state that if there is a proof for the conclusion of an inference tree if there is a proof for all the premises of the rule. This can be defined in Coq as follows:

```
Inductive hoare_proof : Assertion -> Stm -> Assertion -> Type :=
  | ass_p : forall Q x a,
      hoare_proof (assn_sub x a Q) (x ::= a) Q
  | skip_p : forall P,
      hoare_proof P (SKIP) P
  | comp_p  : forall P S1 Q S2 R,
      hoare_proof P S1 Q ->
      hoare_proof Q S2 R ->
      hoare_proof P (S1;S2) R
  | if_p : forall P Q b S1 S2,
    hoare_proof (fun st => P st /\ bassn b st) S1 Q ->
    hoare_proof (fun st => P st /\ ~(bassn b st)) S2 Q ->
    hoare_proof P (IF_ b THEN S1 ELSE S2 ) Q
  | while_p : forall P b S,
    hoare_proof (fun st => P st /\ bassn b st) S P ->
    hoare_proof P (WHILE b DO S ) (fun st => P st /\ ~ (bassn b st))
  | cons_p  : forall (P Q P' Q' : Assertion) S,
    hoare_proof P' S Q' ->
    (forall s, P s -> P' s) ->
    (forall s, Q' s -> Q s) ->
    hoare_proof P S Q.
```

If we have a proof for `{{P}} S {{Q}}` the Hoare triple is valid. If we have a proof for `hoare_proof P S Q` the Hoare triple is provable.

Now, we can prove that the inference system for axiomatic semantics is sound. This proof can be found in the Application section of Soundness and Completeness, 4.7.

Before we prove completeness of the inference system for axiomatic semantics, we need to introduce the concept of the **weakest liberal precondition** for a statement S and a postcondition Q. This is defined as follows

$$\mathbf{wlp}(S,Q)s = \mathbf{tt} \text{ if and only if for all states } s', \text{ if } \langle S, s\rangle \to s' \text{ then } Q\ s' = \mathbf{tt}$$

The **wlp** has two important properties:

- $\models_p \{\ \mathbf{wlp}(S,Q)\ \}\ S\ \{\ Q\ \}$,
- if $\vdash_p \{\ P\ \}\ S\ \{\ Q\ \} \Rightarrow \mathbf{wlp}(S,Q)$

We can define the **wlp** in Coq as follows:

```
Definition wlp (S : Stm) (Q : Assertion) : Assertion :=
  fun s => forall s', << S , s >>--> s' -> Q s'.
```

The two important properties can be defined and proven in Coq as follows:

```
Lemma wlp_property1:
  forall S Q,
  {{wlp S Q}} S {{Q}}.
```

```
Proof.
  intros.
  unfold wlp.
  unfold hoare_triple.
  intros.
  apply H0.
  apply H.
Qed.

Lemma wlp_property2:
  forall P S Q,
      {{P}} S {{Q}}
    ->
      forall s, P s -> wlp S Q s.
Proof.
  intros.
  unfold wlp.
  unfold hoare_triple in H.
  intros.
  apply (H s).
  - apply H1.
  - apply H0.
Qed.
```

3.8 Blocks

We can extend **While** with blocks, in these blocks local variables declared at the start of the block can be used. We will call this new language **Block**. This extension is used in the course but is not formalized in Software Foundations [13]. It is formalized in Verification of Sequential Imperative Programs in Isabelle/HOL by Schirmer [14], however the theory and notation differs a lot from the theory and notation used in Semantics and Correctness. This means that the **Block** extension and the extension **Proc**, discussed in the next section, are not based on other formalizations.

In the language **Block** there are two types of variables: global variables and local variables. Local variables can only be used for computations in the block where the variable has been declared. Global variables are all other variables. The extended syntax of **While** looks like this:

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

Before we define the new inductive type **Stm** in Coq, we need to define what D_V is. D_V is a meta-variable ranging over variable declarations. We can define variable declarations as follows: $D_V ::= \texttt{var } x := a; D_V \mid \varepsilon$. We can represent this in Coq and immediately introduce some notation to use it later on:

```
Inductive Dv : Type :=
  | dec_v (x : string) (a : Aexp) (Dv1 : Dv)
  | empty_v.

Notation "'var' x ':=' a ';' Dv1" := (dec_v x a Dv1)
  (at level 60, a at next level, right associativity) : b_scope.

Notation "'var' x ':=' a" := (var x := a; empty_v)
  (at level 60, a at next level, right associativity) : b_scope.
```

Now we can also define the syntax for the extended version of **While** in Coq:

```
Inductive Stm : Type :=
  | ass (x : string) (a : Aexp)
  | skip
  | comp (S1 S2 : Stm)
  | if_ (b : Bexp) (S1 S2 : Stm)
  | while (b : Bexp) (S : Stm)
  | block (Dv1 : Dv) (S : Stm).
```

We can define a new type of transition for this syntactic category that looks like $\langle D_V, s\rangle \to_D s'$. These transitions can be used alongside the normal natural semantics rules. The rules for variable declarations can be defined as follows:

[none$_{\text{ns}}$]	$\langle \varepsilon, s\rangle \to_D s'$
[var$_{\text{ns}}$]	$\dfrac{\langle D_V, s[x \mapsto \mathcal{A}[\![a]\!]s\rangle \to_D s'}{\langle \texttt{var}\ x := a; D_V, s\rangle \to_D s'}$

Table 10: The natural semantics for variable declarations

We can add this to Coq as follows:

```
Reserved Notation "'<<' S ',' st '>>' '-->D ' st'"
                  (at level 40).

Inductive SDecV : Dv -> State -> State -> Prop :=
  | none_ns : forall st,
      SDecV empty_v st st
  | var_ns : forall st st' a1 n x (Dv1:Dv),
      Aeval st a1 = n ->
      SDecV Dv1 (t_update st x n) st' ->
      SDecV (dec_v x a1 Dv1) st st'.

Reserved Notation "'<<' S ',' st '>>' '-->D ' st'"
                  (at level 40).
```

If we move out of a block, the variable should get the value it had before entering the block. We can get this value back by working with sets. We define the local variables as follows:

$DV(\epsilon) = \emptyset$
$DV(\texttt{var}\ x ::= a; D_V) = \{x\} \cup DV(D_V)$

Coq has a formalization of sets in the standard library [7], we can use this to formalize DV. There are also functions available to add a variable to the set and to check if a variable is a member of a set.

```
Definition DV := list string.

Definition empty_DV : DV := nil.

Fixpoint DV_add (a:string) (x:DV) : DV :=
  match x with
  | nil => a :: nil
  | a1 :: x1 =>
      match string_dec a a1 with
      | left _ => a1 :: x1
      | right _ => a1 :: DV_add a x1
      end
  end.

Fixpoint DV_mem (a:string) (x:DV) : bool :=
    match x with
    | nil => false
    | a1 :: x1 =>
       match string_dec a a1 with
       | left _ => true
       | right _ => DV_mem a x1
       end
    end.
```

Now, we can formalize the mathematical functions defined for DV:

```
Fixpoint DVeval (dv: Dv) : DV :=
  match dv with
  | dec_v x a Dv1 => DV_add x (DVeval Dv1)
  | empty_v => nil
  end.
```

The natural semantics rule for the `block` statement can be defined as follows

$$[\text{block}_{\text{ns}}] \qquad \frac{\langle D_V, s\rangle \to_D s', \quad \langle S, s'\rangle \to s''}{\langle \texttt{begin}\ D_V\ S\ \texttt{end}, s\rangle \to s''[DV(D_V) \mapsto s]}$$

Table 11: The natural semantics rule for **Block**

We need to define what $s''[DV(D_V) \mapsto s]$ does in Coq, we can represent this with a function

```
Definition var_update (s s' : State) (X : DV) (a:string) :=
  fun a => if DV_mem a X then (s a) else (s' a).
```

Finally, we can formalize the block rule in Coq by adding the following to the `Seval` function.

```
  | block_ns : forall Dv S st st' st'' (x:string),
      SDecV Dv st st' ->
      Seval st' S st'' ->
      Seval st (BEGIN Dv, S END)
               (var_update st st'' (DVeval Dv) x).
```

As can be seen, the notation for the rule is not exactly the same as in the book, between `Dv` and `S` a comma has been added. This is needed for parsing reasons. It is also important to add the variable name when applying [block$_{\text{ns}}$] in a proof because Coq cannot find the value for string `x` using pattern matching.

3.9 Procedures

We can also make an extension of **While** that has blocks and procedures. The body of procedures can hold statements that need to be executed at certain points in the program. We want to store the body of the procedure and if we call the procedure, search for the body of the procedure and execute it. The new syntax for this extension of **While** that we will call **Proc** can be defined as follows:

S ::= $x := a$ | `skip` | S_1 ; S_1 | `if` b `then` S_1 `else` S_2 | `while` b `do` S | `begin` D_V D_P S `end` | `call` p

We can define the inductive type D_P as follows: D_P ::= proc p is $S; D_P$ | ε. Here p ranges over procedure names (syntactical category **Pname**) and D_P ranges over procedure declarations (syntactical category **DecP**). As can be seen, the definition of **Stm** uses the definition for **Dp** and the definition of **Dp** uses the definition of **Stm**. This means we cannot define these after each other like normally. To solve this problem we can use the keyword `with`. Again, we add commas between `Dv`, `Dp` and `S` for the parsing reasons. We can now define **Stm** and **Dp** and their notations in Coq as follows:

```
Inductive Stm : Type :=
  | ass (x : string) (a : Aexp)
  | skip
  | comp (S1 S2 : Stm)
  | if_ (b : Bexp) (S1 S2 : Stm)
  | while (b : Bexp) (S : Stm)
  | block (Dv1 : Dv) (Dp1 : Dp) (S : Stm)
  | call (p : string)
with Dp : Type :=
  | dec_p (p : string) (S : Stm) (Dp1 : Dp)
  | empty_p.

Bind Scope pro with Stm.
Notation "x '::=' a" :=
  (ass x a) (at level 60) : pro.
Notation "'SKIP'" :=
   skip : pro.
Notation "s1 ; s2" :=
  (comp s1 s2) (at level 80, right associativity) : pro.
Notation "'WHILE' b 'DO' s " :=
  (while b s) (at level 80, right associativity) : pro.
Notation "'IF_' b 'THEN' s1 'ELSE' s2 " :=
  (if_ b s1 s2) (at level 80, right associativity) : pro.
Notation "'BEGIN' Dv ',' Dp ',' S 'END'" :=
  (block Dv Dp S) (at level 80, right associativity) : pro.
Notation "'CALL' p" :=
  (call p) (at level 80, right associativity) : pro.

Notation "'proc' p 'is' S ';' Dp1" := (dec_p p S Dp1)
  (at level 60, S at next level, right associativity) : pro.
Notation "'proc' p 'is' S" := (dec_p p S empty_p)
  (at level 60, S at next level, right associativity) : pro.
```

We can have two types of procedure scopes: dynamic or static scopes for procedures. In the dynamic procedure scopes we execute the current definition of the procedure. In static procedure scopes we execute the original definition of the procedure at the time of declaration. First, we will handle dynamic procedure scopes.

3.9.1 Dynamic Procedures

We need some way to store the statement that currently belongs to the statement. In the book by Nielson and Nielson [10] this is done with a partial function

$$\mathbf{Env} = \mathbf{Pname} \hookrightarrow \mathbf{Stm}$$

In Coq, we do this with partial maps. Partial maps are similar to the total maps used to define states. We use the *Maps* chapter of Software Foundations [13] to define a partial map from **Pname** to **Stm**. Partial maps and our specific partial map are defined as follows:

```
Definition partial_map (A : Type) := total_map (option A).
Definition empty {A : Type} : partial_map A :=
  t_empty None.
Definition update {A : Type} (m : partial_map A)
             (x : string) (v : A) :=
  (x !-> Some v , m).
Notation "x '|->' v ',' m" := (update m x v)
  (at level 100, v at next level, right associativity).
Notation "x '|->' v" := (update empty x v)
  (at level 100).

Definition Env := partial_map Stm.
```

Here the `option` type is used. This is a type from the standard Coq library [6]. The type is often used for partial maps. It indicates that the map can go to something of the type `A` in `option A`, or the value of the map is undefined. If the value of the map is undefined it goes to `None`. The definition of the option type in the standard Coq library is as follows

```
Inductive option (A:Type) : Type :=
  | Some : A -> option A
  | None : option A.
```

To be able to use the environment, it needs to be added to the transitions. In the book, transitions for the language **Proc** are defined as $env_p \vdash \langle S, s\rangle \rightarrow s'$. We need the new transitions for all the natural semantics rules, but not for the rules with the transition $\rightarrow_D$ defined for **Block**. To update the environment when a new procedure is encountered we can define the following function:

$$\begin{aligned} upd_P(\varepsilon, env_P) &= env_P \\ upd_P(\text{proc } p \text{ is } S; D_P, env_P) &= upd_P(D_P, env_P[p \mapsto S]) \end{aligned}$$

Where $env_p[p \mapsto S]$ indicates that we update p in env_P with statement S. We can represent this in Coq as follows:

```
Fixpoint UpdP (dp : Dp) (envp : Env) : Env :=
  match dp with
  | dec_p p s dp => UpdP dp (t_update envp p s)
  | empty_p => envp
  end.
```

We can now define the natural semantics rules for **Proc**. Of course, we actually need to redefine all the rules because they now use a different transition that includes the environment. However, this is this a very small change so we will only show the new rules for [block$_{\text{ns}}$] and [call$_{\text{ns}}^{\text{rec}}$].

[block$_{\text{ns}}$]	$$\frac{upd_P(D_V, env_p) \vdash \langle D_V, s\rangle \to_D s' \quad env_p \vdash \langle S, s'\rangle \to s''}{\langle \texttt{begin}\ D_V\ D_P\ S\ \texttt{end}, s\rangle \to s''[DV(D_V) \mapsto s]}$$
[call$_{\text{ns}}^{\text{rec}}$]	$$\frac{env_p \vdash \langle S, s\rangle \to s'}{env_p \vdash \langle \texttt{call}\ p, s\rangle \to s'}$$ where $env_p\ p = S$

Table 12: The natural semantics rules for **Proc**

We can add these rules with the new notation using env_p in Coq as follows:

```
  | block_ns : forall envp (dv:Dv) (dp:Dp) S st st' st'' x,
      SDecV dv st st' ->
      Seval (UpdP dp envp) st' (Some S) st'' ->
      Seval envp st (Some (BEGIN dv, dp, S END))
        (var_update st st'' (DVeval dv) x)
  | call_rec_ns : forall envp st st' p,
      Seval envp st (envp p) st' ->
      Seval envp st (Some (CALL p)) st'.
Notation "envp |- << s , st >>-->st'" := (Seval envp st s st')
```

The `Some` is included to let Coq know that this is actually a statement and not something of type `None`.

3.9.2 Static Procedures

For static procedures we execute the original definition of the procedure at the time of declaration. This means we will need a different definition of the environment which also includes the environment at the time of declaration. The mathematical definition of this new environment looks as follows:

$$\mathbf{Env} = \mathbf{Pname} \hookrightarrow \mathbf{Stm} \times \mathbf{Env}$$

In Coq, there already exists a special type called `prod` [6]. We can use this to represent the **Stm** $\times$ **Env** part. However, Coq is unable to define definitions that use their own definition. This means we cannot use the partial map definition like in dynamic scopes. We need to define a new inductive type that looks as follows:

```
Inductive Envp : Type :=
  | static_env (f : Pname -> option (Stm * Envp)).
```

The function `Envp` is similar to the partial map defined for dynamic scopes but now the partial function can have the `Envp` type in the mapping. We still need to define the actual mapping. In the previous partial maps this is done by comparing the strings, we can do the same for this function.

```
Definition extend (e : Envp) (p : Pname) (s : Stm) : Envp :=
  match e with
  | static_env f => static_env (fun p' : Pname
    => if eqb_string p p' then (Some (s,e)) else f p')
  end.
```

We also need to define the empty environment.

```
Definition empty_envp := static_env (fun _ => None).
```

Lastly, a new updating rule is needed because the environment changed. For the given mathematical definition that would be as follows:

$$upd_P(\varepsilon, env_P) = env_P$$
$$upd_P(\text{proc } p \text{ is } S; D_P, env_P) = upd_P(D_P, env_P[p \mapsto (S, env_P])$$

The only real difference is that **Pname** maps to **(Stm** $\times$ **Env)** instead of only **Stm**. In Coq this update function looks as follows:

```
Fixpoint UpdP2 (dp : Dp) (envp : Envp) : Envp :=
  match dp with
  | dec_p p s dp => UpdP2 dp (extend envp p s)
  | empty_p => envp
  end.
```

With this environment and update function we can define the natural semantics rules for **Proc**. We can define both the recursive call statement and a call statement where the statement goes to the previous environment.

$[\text{call}_{\text{ns}}]$	$\dfrac{env'_p \vdash \langle S, s\rangle \to s'}{env_p \vdash \langle \texttt{call } p, s\rangle \to s'}$
$[\text{call}^{\text{rec}}_{\text{ns}}]$	$\dfrac{env'_p[p \mapsto (S, env'_p)] \vdash \langle S, s\rangle \to s'}{env_p \vdash \langle \texttt{call } p, s\rangle \to s'}$ where $env_p\ p = (S, env'_p)$

Table 13: The natural semantics rules for Proc with mixed scope rules

As can be seen we need to be able to extract both the statement and the environment from the environment. We based the new functions on the functions `fst` and `snd` from the `prod` type [6]. These functions take the first (`fst`) or second (`snd`) argument from the `prod` type.

```
Definition fst2 (p: option (Stm * Envp)) :=
  match p with
  | Some (x, y) => x
  | None => SKIP
end.
Definition first (e : Envp) (p : Pname) :=
  match e with
  | static_env f => fst2 (f p)
  end.

Definition snd2 (p: option (Stm * Envp)) :=
  match p with
  | Some (x, y) => y
  | None => empty_envp
end.
Definition second (e : Envp) (p : Pname) :=
  match e with
  | static_env f => snd2 (f p)
  end.
```

Because the definition for this environment works slightly different we do not have to add `Some` in the definition of the rules. Again, we will only show the rules with the most changes.

```
  | call_ns : forall (envp:Envp) st st' p,
      Seval (second envp p) st (first envp p) st' ->
      Seval envp st (CALL p) st'
  | call_rec_ns : forall envp st st' p,
      Seval envp st (first envp p) st' ->
      Seval envp st (CALL p) st'
```

We can now use these rules to make a proof with static or dynamic procedure scopes. However, every separate proof one of the rules should be chosen. They cannot be mixed.

Chapter 4

Application of the framework

In the application of the framework we will provide some Coq proofs of theorems, examples and exercises presented in the book [10] and slides of the course Semantics and Correctness using the framework described in the previous chapter.

4.1 Semantics of Expressions

Example 1.5 Suppose that `s x = 3` and we want to calculate $\mathcal{A}[\![x{+}1]\!]s$. The Coq proof looks like this:

```
Example Example1_5 :
  (A[[x+1]] (x!->3)) = 4.
Proof.
reflexivity.
Qed.
```

First, we tell Coq that variable `x` starts with value 3, using `(x !-> 3)`. Then, we state that we want to calculate `(x + 1)` with the given state. We also already give the expected output, `4`. However, we can also easily ask Coq to compute what the output will be using the `Compute` command. Coq will give `= 4 : Z` as output.

```
Compute (A[[x+1]] (x!->3)).
```

The Coq command `reflexivity` is already able to completely reduce it to 4 so the proof is very simple.

4.2 Natural Semantics

Example 2.1 We will build a proof for the statement `(z := x; x := y); y := z` with starting values `s x = 5`, `s y = 7` and all other variables are 0. In Coq we represent this as:

```
Example Example2_1 :
  << (z::= x; x::= y); y::= z , y!->7, x!->5 >>
 -->(y!->5, x!->7, z!->5, y!->7, x!->5).
```

This means that we execute the given statement from a state with the given values for the variables and we end up in the final state where `s x = 7`, `s y = 5` and `s z = 5`. The proof looks like this:

```
Proof.
  apply comp with (x!->7, z!->5, y!->7, x!->5).
  - apply comp with (z!->5, y!->7, x!->5).
    + apply ass. reflexivity.
    + apply ass. reflexivity.
  - apply ass. reflexivity.
Qed.
```

It is very similar to the proof trees used in the book. Every level in the proof represents a level in the tree.

$$\dfrac{\dfrac{\langle \texttt{z:=x}, s_{5,7,\bot}\rangle \to s_{5,7,5}\ [\text{ass}_{\text{ns}}] \qquad \langle \texttt{x:=y}, s_{5,7,5}\rangle \to s_{7,7,5}\ [\text{ass}_{\text{ns}}]}{\langle \texttt{z:=x;x:=y}, s_{5,7,\bot}\rangle \to s_{7,7,5}}\,[\text{comp}_{\text{ns}}] \qquad \langle \texttt{y:=z}, s_{7,7,5}\rangle \to s_{7,5,5}\ [\text{ass}_{\text{ns}}]}{\langle \texttt{(z:=x;x:=y);y:=z}, s_{5,7,\bot}\rangle \to s_{7,5,5}}\,[\text{comp}_{\text{ns}}]$$

In the book the final state is written more compactly, not all the intermediate states are in the conclusion. We can solve this problem by introducing a theorem and a new tactic. However, first we will try to get rid of having to specify the intermediate states that need to be given for certain rules. It is possible to omit the intermediate states using the `eapply` tactic instead of `apply`. A proof using `eapply` looks like this:

```
Example Example2_1b :
  << (z::= x; x::= y); y::= z ,  z!->0,y!->7, x!->5 >>
 -->(y!->5,x!->7,z!->5,z!->0,y!->7,x!->5).
Proof.
  eapply comp.
  - eapply comp.
    + apply ass. reflexivity.
    + apply ass. reflexivity.
  - apply ass. reflexivity.
Qed.
```

We can add the following theorem that states that if execution of statement S in state s leads to state s' and $s' = s''$, then the execution of statement S in state s must also lead to state s'':

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

If we `eapply` this theorem at the start of a proof, we get a state `?s` that we can `eapply` all the other rules to without any problems and in the end we only have to prove that this $?s$ is equal to the given final state. To prove two states are equal, we need the concept of **functional extensionality**. Using functional extensionality, two functions are said to be equal if they produce the same outputs when given the same inputs. It is not part of Coq's standard library, but it is easy to import. Functional extensionality for states means that two states are equal if all variables evaluate to the same value. The `t_update` function checks if two variables are equal, we can `unfold` this function and simplify it using `destruct` to get to a state where every variable only has one value, this is the final state that would usually also be in the proof tree. Using this we can make the following proof:

```
Example Example2_1c :
  << (z::= x; x::= y); y::= z  , y!->7, x!->5 >>
 -->(y!->5, x!->7, z!->5).
Proof.
  eapply stm_eq.
  eapply comp_ns.
  - eapply comp_ns.
    + apply ass_ns. reflexivity.
    + apply ass_ns. reflexivity.
  - apply ass_ns. reflexivity.
  - apply functional_extensionality.
    intros x0.
    unfold t_update.
    simpl.
    destruct (eqb_string y x0). reflexivity.
    destruct (eqb_string x x0). reflexivity.
    reflexivity.
Qed.
```

We can make this easier by introducing a tactic to check if the two states in the end are equal because it has a systematic method of applying `functional extensionality`, unfolding `t_update`, `intros`, `simpl` and then using `destruct` and `reflexivity` until we find that the two states are equal. The tactic can be defined like this:

```
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.
```

The final proof then looks as follows:

```
Example Example2_1d :
  << (z::= x; x::= y); y::= z , y!->7, x!->5 >>
 -->(y!->5, x!->7, z!->5).
Proof.
  eapply stm_eq.
  eapply comp_ns.
  - eapply comp_ns.
    + apply ass_ns. reflexivity.
    + apply ass_ns. reflexivity.
  - apply ass_ns. reflexivity.
  - eq_states.
Qed.
```

Using `eapply` and `stm_eq` is useful when a student is trying to build the tree, while using `apply` is useful to check if the tree created on paper is correct. We will give proofs using `apply` or `eapply` in the remaining part of this thesis, depending on the size and complexity of the proof.

Example 2.2 We can also prove that the factorial statement `y := 1; while ¬(x=1) do (y := y ⋆ x; x := x-1)` starting with `s x = 3`, ends with `s x = 1` and `s y = 6`. We can prove this in Coq as the following:

```
Theorem Example2_2 :
  << y::= 1 ; WHILE ~(x=1) DO ( y ::= y * x ; x ::= x-1 ),
   x!->3 >>-->(x!->1, y!->6, x!->2, y!->3, y!->1, x!->3).
Proof.
  apply comp_ns with ( y!->1, x!->3).
  - apply ass_ns. reflexivity.
  - apply while_tt_ns with ( x!->2 , y!->3, y!->1, x!->3).
    + reflexivity.
    + apply comp_ns with ( y!->3, y!->1, x!->3).
      * apply ass_ns. reflexivity.
      * apply ass_ns. reflexivity.
    + apply while_tt_ns with
        ( x!->1, y!->6, x!->2, y!->3, y!->1, x!->3).
      * reflexivity.
      * apply comp_ns with
        ( y!->6, x!->2, y!->3, y!->1, x!->3).
        { apply ass_ns. reflexivity. }
        { apply ass_ns. reflexivity. }
      * apply while_ff_ns.
```

```
        reflexivity.
Qed.
```

4.3 Semantic Equivalence

Lemma 2.5 The statement ‘`while b do S`’ is semantically equivalent to ‘`if b then (S; while b do S) else skip`’. The lemma is represented like this:

```
Lemma Lemma2_5 : forall b S,
  Sequiv
    ( WHILE b DO S )
    ( IF_ b THEN (S ; WHILE b DO S) ELSE SKIP ).
```

The proof looks like this:

```
  intros b S st st''.
  symmetry.
  split; intros Hce.
  - (* -> while b do S, s -> s''*)
    inversion Hce; subst.
    + (* loop runs *)
      inversion H5; subst.
      apply while_tt_ns with (st' := st').
      apply H4.
      apply H3.
      apply H6.
    + (* loop doesn't run *)
      inversion H5; subst.
      apply while_ff_ns.
      apply H4.
  -  (* <- <if b then (S; while b do S) else skip, s> -> s''*)
    inversion Hce; subst.
    + (* loop runs *)
      apply if_tt_ns.
      apply H2.
      apply comp_ns with (st' := st').
      apply H4.
      apply H5.
    + (* loop doesn't run *)
      apply if_ff_ns.
      apply H3.
      apply skip_ns.
Qed.
```

In the book we first prove → and then ←. To get the same order in Coq, we will have to do an extra `symmetry` command before splitting the equivalence. The proofs are also combined in one proof. Here, `inversion Hce; subst.` will try to derive as much information as possible from the hypothesis and then simplify the available hypotheses.

Induction on the shape of the derivation tree In the slides [4], it is proven that S; `while` $\neg b$ `do` S is semantically equivalent to `repeat` S `until` b. This is proven by induction on the shape of the derivation tree. In induction on the shape of the derivation tree, we first prove that a certain property holds for all the simple derivation trees by showing that it holds for the axioms of the transition system. Then, we prove that the property holds for all composite derivation trees. This means that we assume that the property holds for all premises of each rule and then we prove that it also holds for the conclusion of the rule if the conditions of the rule are satisfied. To prove that S; `while` $\neg b$ `do` S is semantically equivalent to `repeat` S `until` b, we first have to specify the property and prove that it holds for all axioms. The property in the slides is specified as:
P(T): For all states s and s' we have that if the conclusion of T is

$$\langle \texttt{ repeat } S \texttt{ until } b, s\rangle \rightarrow s'$$

then there exists a derivation tree T' with the conclusion

$$\langle\ S;\ \texttt{while } \neg b \texttt{ do } S, s\rangle \rightarrow s'$$

Before we start the actual proof we will prove a lemma to help with the direction $\langle S$; `while` $\neg b$ `do` S, $s\rangle \rightarrow s'$ implies $\langle$ `repeat` S `until` b, $s\rangle \rightarrow s'$. This was also given in the assignment where the semantic equivalence of the two statements in that direction needed to be proven. The lemma uses different states in certain places which makes it easier to prove. This lemma can be formalized and proven in Coq as follows:

```
Lemma Lem_Assignment:
  forall b S s s' s'',
    << WHILE ~b DO S , s'' >>-->s'
  ->
    << S , s >>-->s''
  ->
    << REPEAT S UNTIL b , s >>-->s'.
Proof.
  intros b S.
  intros st st' st'' H0.
  remember (WHILE ~b DO S) as Swhile eqn:HeqSwhile.
  generalize dependent st.
  induction H0.
  - (* ass *)
    inversion HeqSwhile.
  - (* skip *)
    inversion HeqSwhile.
  - (* comp *)
    inversion HeqSwhile.
  - (* if tt *)
    inversion HeqSwhile.
  - (* if ff *)
    inversion HeqSwhile.
  - (* while tt *)
     inversion HeqSwhile; subst.
     intro st'''.
     intro H1.
     apply repeat_until_ff_ns with (st':=st).
     + assumption.
     + simpl in H.
       apply negb_true_iff in H.
       assumption.
     + apply IHSeval2.
       reflexivity.
       assumption.
  - (* while ff *)
    inversion HeqSwhile; subst.
    intro s.
    intro H1.
    apply repeat_until_tt_ns.
    + apply H1.
    + simpl in H.
      apply negb_false_iff in H.
      assumption.
  - (* repeat until tt *)
    inversion HeqSwhile.
  - (* repeat until ff *)
    inversion HeqSwhile.
Qed.
```

Here, it is important to use `remember (WHILE ~b DO S)`. This will give a hypothesis `HeqSwhile : Swhile = (WHILE ~b DO S)` in the context. From the second `intros` we obtain the hypothesis `H0 : << Swhile, st'' >>--> st'`. By executing `generalize dependent st` and `induction H0`, we will get nine subgoals, one for each natural semantics rule. And for proving every subgoal we get a variation on `HeqSwhile` in the context. For example, when proving the assignment rule we get the following hypothesis `HeqSwhile : x ::= a1 = (WHILE ~ b DO S)`. Of course, the assignment rule is not equal to the while rule, so we can easily prove this by using the `inversion` tactic. This means, we can prove most rules simply because this cannot have been the last rule that has been applied, the rule vacuously holds. However, if we leave out the `remember (WHILE ~b DO S)` line, we do not get any hypothesis in the context that can be used to prove the subgoal because the last rule cannot have been applied. In the rules where the last rule that has been applied does match `(WHILE ~b DO S)` the `inversion` does not automatically solve the subgoal, some more work is needed.

We can now formalize the semantic equivalence proof in Coq. For the direction $\langle S;\ \texttt{while}\ \neg b\ \texttt{do}\ S, s\rangle \to s'$ implies $\langle \texttt{repeat}\ S\ \texttt{until}\ b, s\rangle \to s'$ we need to use the lemma from above. The proof looks as follows:

```
Theorem repeat_until_slides : forall b S,
  Sequiv
    ( REPEAT S UNTIL b )
    ( S ; WHILE ~b DO S ).
Proof.
  intros b S.
  split; intro HSe.
  - (* -> *)
    remember (REPEAT S UNTIL b ) as Srepunt eqn:HeqSrepunt.
    induction HSe.
    + (* ass *)
      inversion HeqSrepunt.
    + (* skip *)
      inversion HeqSrepunt.
    + (* comp *)
      inversion HeqSrepunt.
    + (* if tt *)
      inversion HeqSrepunt.
    + (* if ff *)
      inversion HeqSrepunt.
    + (* while tt *)
      inversion HeqSrepunt.
    + (* while ff *)
      inversion HeqSrepunt.
    + (* repeat until tt *)
       inversion HeqSrepunt; subst.
       apply comp_ns with (st' := st').
       assumption.
       apply while_ff_ns.
       simpl.
       apply negb_false_iff.
       assumption.
    + (* repeat until ff *)
       inversion HeqSrepunt; subst.
       assert (<< S; WHILE ~ b DO S, st' >>--> st'').
       { apply IHHSe2.
         reflexivity. }
       inversion H0; subst.
       apply comp_ns with (st' := st').
       * assumption.
       * apply while_tt_ns with (st' := st'0).
         simpl.
```

```
        apply negb_true_iff.
        assumption.
        assumption.
        assumption.
  - (* <- *)
    inversion HSe; subst.
    apply Lem_Assignment with (s'':=st'0).
    assumption.
    assumption.
Qed.
```

4.4 Determinism

The following proof is based on the determinism proof in chapter *Imp* of Software Foundations [13].

Theorem 2.9 We can prove that the natural semantics of **While** are deterministic by performing a case distinction on all the rules. The Coq proof looks like this:

```
Theorem Seval_deterministic: forall S s s' s'',
     << S, s >>-->s'  ->
     << S, s >>-->s'' ->
      s' = s''.
Proof.
  intros S s s' s'' H1 H2.
  generalize dependent s''.
  induction H1; intros s'' H2; inversion H2; subst.
  - (* ass *) reflexivity.
  - (* skip *) reflexivity.
  - (* comp *)
    assert (st' = st'0) as H3.
    { apply IHSeval1; assumption. }
    subst st'0.
    apply IHSeval2. assumption.
  - (* if tt, b1 evaluates to true *)
      apply IHSeval. assumption.
  - (* if tt,  b1 evaluates to false (contradiction) *)
      rewrite H in H7. discriminate H7.
  - (* if ff, b1 evaluates to true (contradiction) *)
      rewrite H in H7. discriminate H7.
  - (* if ff, b1 evaluates to false *)
      apply IHSeval. assumption.
  - (* while tt, b1 evaluates to true *)
      assert (st' = st'0) as H3.
      { apply IHSeval1; assumption. }
      subst st'0.
      apply IHSeval2. assumption.
  - (* while tt, b1 evaluates to false (contradiction) *)
    rewrite H in H5. discriminate H5.
  - (* while ff, b1 evaluates to true (contradiction) *)
    rewrite H in H4. discriminate H4.
  - (* while ff, b1 evaluates to false *)
    reflexivity.
Qed.
```

For this proof we use induction to get all the possible rules. This will also give us an induction hypothesis based on the rule we currently want to prove. For example, during the subproof of the assignment rule we have the following hypothesis in the context `<< x ::= a1, st >>-->(x!->A [[a1]] st, st)`. It looks like a long proof, but the proof for each rule is actually pretty short. The proof can be made even shorter by using automation, more about this can be found in the *Automation* chapter of Software Foundations [13].

4.5 Hoare logic

Example 6.8 We can prove `while true do skip` using the Hoare triple
{ true } `while true do skip` { true }:

```
Theorem Example6_8a :
  {{ B[[BTrue]] }} WHILE BTrue DO SKIP {{ B[[BTrue]] }}.
Proof.
  intros P Q.
  apply consp with
    (P':= B[[BTrue]]) (Q' := ~B[[BTrue]] /\pB[[BTrue]]).
  apply whilep.
  - apply hoare_post_true.
    intros st.
    unfold bassn.
    simpl.
    reflexivity.
  - (* Precondition implies invariant *)
    intros st H.
    constructor.
  - (* Loop invariant and negated guard imply postcondition *)
    simpl.
    unfold assert_implies.
    unfold and_sub.
    intros st [H0 H1].
    apply H1.
Qed.
```

However, we can also prove { true } `while true do skip` { false }

```
Theorem Example6_8b :
  {{ B[[BTrue]] }} WHILE BTrue DO SKIP {{ ~B[[BTrue]] }}.
Proof.
  intros P Q.
  apply consp with
    (P':= B[[BTrue]]) (Q' := ~B[[BTrue]] /\pB[[BTrue]]).
  apply whilep.
  - apply hoare_post_true.
    intros st.
    unfold bassn.
    simpl.
    reflexivity.
  - (* Precondition implies invariant *)
    intros st H.
    constructor.
  - (* Loop invariant and negated guard imply postcondition *)
    simpl.
    unfold assert_implies.
    unfold and_sub.
    intros st [H0 H1].
    apply H0.
Qed.
```

This is possible because `while true do skip` never terminates.

Example 6.9 We can also prove the assertion

{ `x = n` } `y := 1; while ¬(x=1) do (y := y⋆x; x := x-1)` { `y = n! ∧ n > 0` }

In this case, the `!` represents the factorial function. To use this in the assertion, we first need to define the factorial function. The following factorial function was taken from the chapter *Hoare logic, part I* of Software Foundations [13].

```
Fixpoint real_fact (n : nat) : nat :=
  match n with
  | O => 1
  | S n' => n * (real_fact n')
  end.
```

After this, we can formalize the proof as follows:

```
Theorem Example6_9:
  forall (m:nat),
  {{ fun st => st x = m }}
  y ::= 1; WHILE ~(x=1)
  DO (y ::= AMult y x; x ::= AMinus x 1)
  {{ fun st => st y = (real_fact m) /\ m > 0}}.
```

```
Proof.
  intros m.
  apply consp_pre with (fun (st:State) => (st x > 0 ->
        real_fact (st x) = real_fact m /\ m >= st x)).
  apply compp with (fun (st:State) => (st x > 0 ->
        st y * real_fact (st x) = real_fact m /\ m >= st x)).
  eapply consp_post.
  apply whilep.
  apply consp_pre with (fun (st:State) => st x - 1 > 0 ->
        st y * st x * real_fact (st x - 1) =
        real_fact m /\ m >= (st x - 1)).
  apply compp with (fun (st:State) => st x - 1 > 0 ->
        st y * real_fact (st x - 1) =
        real_fact m /\ m >= (st x - 1)).
  eapply consp_pre.
  apply assp.
  - intros st P Q.
    unfold bassn, assn_sub, assert_implies, t_update. simpl.
    unfold bassn, assn_sub, assert_implies, t_update in Q.
    simpl in Q.
    apply P in Q.
    apply Q.
  - eapply consp_pre.
    apply assp.
    intros st P Q.
    unfold bassn, assn_sub, assert_implies, t_update. simpl.
    unfold bassn, assn_sub, assert_implies, t_update in Q.
    simpl in Q.
    apply P in Q.
    apply Q.
  - unfold and_sub.
    intros st P Q.
    destruct P.
    unfold bassn in H.
    simpl in H.
    apply negb_true_iff in H.
    apply beq_nat_false in H.
    assert (forall n, n <> 0 -> n * real_fact (n - 1) =
        real_fact n).
    + intros. destruct n.
      * destruct H1. reflexivity.
      * simpl. rewrite <- minus_n_O. reflexivity.
    + rewrite <- mult_assoc. rewrite H1. omega. omega.
  - unfold and_sub.
    intros st P. destruct P.
    unfold bassn2 in H.
    apply not_true_iff_false in H.
    simpl in H.
    apply negb_false_iff in H.
    apply beq_nat_true in H.
    assert (st x > 0).
      { omega. }
    apply H0 in H1.
    assert (real_fact (st x) = 1).
      { rewrite H. reflexivity. }
    rewrite H2 in H1. omega.
  - eapply consp_pre.
    apply assp.
    intros st P Q.
    unfold bassn, assn_sub, assert_implies, t_update. simpl.
    unfold bassn, assn_sub, assert_implies, t_update in Q.
    simpl in Q.
    apply P in Q.
    rewrite <- plus_n_O.
    apply Q.
  - intros st P H.
    rewrite P.
    omega.
Qed.
```

The proof follows the rules described in the book by Nielson and Nielson [10]. Moreover, to prove certain steps we need to unfold, rewrite or simplify parts in the context or the subgoal. When the [cons_p] rules are used, it is common that we need to assert that the current state and a state in a hypothesis are equal. This can be done by using the `assert` tactic and mathematical lemmas.

4.6 Annotated programs

Since annotated programs are not described in the book by Nielson and Nielson [10], we will use the examples of a pdf about annotated programs that was also available in the Semantics and Correctness course [2].

Example of an annotated program In the annotated program document the first annotated program is of { `true` } `while true do skip` { `false` }. The annotated program can be represented in Coq as follows:

```
Example annotatedprogram : Annotated := (
  {{ B[[BTrue]] }}
  WHILE BTrue
  DO
    {{B[[BTrue]] /\pB[[BTrue]]}} =>>
    {{B[[BTrue]]}}
    SKIP
    {{ B[[BTrue]] }}
  {{ ~B[[BTrue]]/\pB[[BTrue]]}}=>>
  {{ B[[BFalse]] }}
)
.
```

We can prove the annotated program is correct using the `verify` tactic that we defined in the formalization.

```
Theorem annotatedprogram_correct :
  ann_correct (annotatedprogram).
Proof.
  verify.
  unfold and_sub in H.
  destruct H.
  unfold not in H.
  destruct H.
  reflexivity.
Qed.
```

The only thing that the `verify` tactic does not solve yet is the last use of the $[\text{cons}_\text{p}]$ rule { ¬`true`∧`true` } ⇒ { `false` }. This can easily be proven manually by unfolding the ∧p notation and proving that ¬`true ∧ true = false`.

Example 6.9 revisited We have already seen the proof for the following assertion { `x=n` } `y := 1; while ¬(x=1) do (y := x⋆y; x := x-1)` { `y=n! ∧ n>0` } using the axiomatic semantics rules. We can also translate this into an annotated program and prove the annotated program. We use the previously defined `real_fact` function. The annotated program follows the proof in the book:

```
Example my_fact (m:nat) : Annotated := (
    {{ fun st => st x = m }}=>>
    {{ fun st => st x > 0 -> real_fact (st x) =
         real_fact m /\ m >= st x }}
  y ::= ANum 1
    {{ fun st => st x > 0 -> st y * real_fact (st x) =
         real_fact m /\ m >= st x }};
  WHILE BNot (BEq (AId x) (ANum 1))
  DO   {{ fun st => (st x > 0 ->
         st y * real_fact (st x) = real_fact m /\ m >= st x)
       /\ st x <> 1 }}=>>
       {{ fun st => st x - 1 > 0 ->
         st y * st x * real_fact (st x - 1) =
         real_fact m /\ m >= (st x - 1) }}
     y ::= AMult (AId y) (AId x)
       {{ fun st => st x - 1 > 0 ->
         st y * real_fact (st x - 1) =
        real_fact m/\ m >= (st x - 1) }};
     x ::= AMinus (AId x) (ANum 1)
       {{ fun st => st x > 0 -> st y * real_fact (st x)
        = real_fact m/\ m >= st x }}
    {{ fun st => (st x > 0 -> st y * real_fact (st x) =
        real_fact m/\ m >= st x) /\ st x = 1 }}=>>
    {{ fun st => st y = real_fact m/\ m > 0}}
).
```

After using the `verify` tactic we have some subgoals that still need to be proven. These subgoals are usually parts in the proof where the [cons_p] rule has been applied. This means that we have to prove that a state in the subgoal and hypothesis are equal. This usually requires mathematical lemmas that are not in the `verify` tactic. The proof looks like this:

```
Theorem my_factcorrect : forall m,
  ann_correct (my_fact m).
Proof.
  intros.
  verify.
  - rewrite <- plus_n_O.
    assert ((y!->1, st) x > 0 ->
        real_fact (st x) = real_fact m/\ m >= st x).
    { apply H. }
    apply H1 in H0.
    apply H0.
  - assert (forall n, n <> 0 -> n * real_fact (n - 1)
       = real_fact n).
    + intros.
      destruct n.
      * destruct H1.
        reflexivity.
      * simpl.
        rewrite <- minus_n_O.
        reflexivity.
    + unfold not.
      intros.
      rewrite H2 in H.
      discriminate H.
  - apply beq_nat_true in H.
    apply H.
  - assert (forall n, n <> 0 -> n * real_fact (n - 1) =
        real_fact n).
    + intros.
      destruct n.
      * destruct H2.
```

```
        reflexivity.
      * simpl.
        rewrite <- minus_n_O.
        reflexivity.
    + rewrite <- mult_assoc.
      rewrite H2.
      omega.
      omega.
  - assert ((y!->st y * st x, st) x - 1 > 0 ->
        st y * st x * real_fact (st x - 1)
        = real_fact m /\ m >= st x - 1).
    { apply H. }
    apply H1 in H0.
    apply H0.
  - assert ((x!->st x - 1, st) x > 0 ->
        st y * real_fact (st x - 1)
        = real_fact m /\ m >= st x - 1).
    { apply H. }
    apply H1 in H0.
    apply H0.
  - assert (1 > 0).
    { omega. }
    apply H in H0.
    assert (real_fact 1 = 1).
    { reflexivity. }
    rewrite H1 in H0.
    assert (st y * 1 = st y).
    { omega. }
    rewrite H2 in H0.
    apply H0.
Qed.
```

4.7 Soundness and Completeness

As mentioned in the formalization of the framework, we can prove the inference system for axiomatic semantics is sound and complete.

6.17 The inference system for axiomatic semantics is sound.

```
Theorem hoare_proof_sound : forall P S Q,
  hoare_proof P S Q -> {{P}} S {{Q}}.
Proof.
  intros.
  induction X.
  - unfold hoare_triple.
    intros.
    inversion H; subst.
    apply H0.
  - unfold hoare_triple.
    intros.
    inversion H; subst.
    apply H0.
  - unfold hoare_triple.
    intros.
    inversion H; subst.
    apply (IHX2 st'0 st').
    apply H6.
    apply (IHX1 st st'0).
    apply H3.
    apply H0.
  - unfold hoare_triple.
    intros.
    inversion H; subst.
    + apply (IHX1 st st').
      apply H7.
      split.
      apply H0.
      apply H6.
    + apply (IHX2 st st').
      apply H7.
      split.
      apply H0.
      apply Bexp_eval_false.
      apply H6.
  - unfold hoare_triple.
    intros.
    remember (WHILE b DO S ) as wStm eqn:HeqwStm.
    induction H.
    + inversion HeqwStm.
    + inversion HeqwStm.
    + inversion HeqwStm.
    + inversion HeqwStm.
    + inversion HeqwStm.
    + inversion HeqwStm; subst.
      apply IHSeval2.
      reflexivity.
      apply (IHX st st').
      apply H1.
      split.
      apply H0.
      apply Bexp_eval_true.
      apply H.
    + inversion HeqwStm; subst.
      split.
      apply H0.
      apply Bexp_eval_false.
      apply H.
  - unfold hoare_triple.
    intros.
    apply q.
    apply (IHX st st').
    apply H.
    apply p.
    apply H0.
Qed.
```

6.23 The inference system for axiomatic semantics is complete.

```
Theorem hoare_proof_complete: forall P S Q,
  {{P}} S {{Q}} -> hoare_proof P S Q.
Proof.
  intros P S.
  generalize dependent P.
  induction S; intros P Q HT.
  - eapply cons_p.
    + eapply ass_p.
    + intro s.
      apply HT.
      econstructor.
      reflexivity.
    + intros.
      apply H.
  - eapply cons_p.
    + eapply skip_p.
    + intros.
      eapply H.
    + intro st.
      apply HT.
      apply skip_ns.
  - apply comp_p with (wlp S2 Q).
    + eapply IHS1.
      unfold wlp.
      intros s s' E1 H.
      intros s'' E2.
      eapply HT.
      * econstructor.
        eapply E1.
        eapply E2.
      * apply H.
    + eapply IHS2.
      intros s s' E1 H.
      apply H.
      apply E1.
  - eapply if_p.
     + apply IHS1.
       intros s s' H [H1 H2].
       eapply HT.
       * apply if_tt_ns.
         apply H2.
         apply H.
       * apply H1.
     + apply IHS2.
       intros s s' H [H1 H2].
       eapply HT.
       * apply if_ff_ns.
         unfold bassn in H2.
         apply not_true_is_false in H2.
         apply H2.
         apply H.
       * apply H1.
  - eapply cons_p with (P' := wlp (WHILE b DO S ) Q).
    + apply while_p.
      apply IHS.
      intros s s' H [H2 H3] s'' H4.
      assert(<<(WHILE b DO S ) , s >>-->s'').
      * eapply while_tt_ns.
        apply H3.
        apply H.
        apply H4.
      * eapply wlp_property1.
        { apply H0. }
        { apply H2. }
    + apply wlp_property2.
      apply HT.
    + intros s [H1 H2].
      eapply wlp_property1.
      * assert(<<(WHILE b DO S ), s>>--> s).
        { apply while_ff_ns.
           unfold bassn in H2.
           apply not_true_is_false in H2.
           apply H2.
        }
        { apply H. }
      * apply H1.
```

```
Qed.
```

4.8 Blocks

Exercise 2.37 We can prove that the following program will lead to a state where x has value 4.

```
Definition example2 :=
  BEGIN var y := 1,
    ( x ::= 1;
      (BEGIN var x := 2, y ::= x + 1 END);
      x ::= y + x )
  END.

Theorem blocksproof2 :
  << example2, y!->0, x!->0 >>
 -->(x!->4, y!->0).
Proof.
  unfold example2.
  eapply stm_eq.
  eapply block_ns with (x := y).
  - eapply var_ns.
    + reflexivity.
    + eapply none_ns.
  - eapply comp_ns.
    + eapply ass_ns.
      reflexivity.
    + eapply comp_ns.
      * eapply block_ns with (x := x).
        { eapply var_ns.
          - reflexivity.
          - eapply none_ns.
        }
        { eapply ass_ns.
          reflexivity.
        }
      * eapply ass_ns.
        reflexivity.
  - simpl.
    unfold var_update.
    simpl.
    apply functional_extensionality.
    intros.
    destruct (string_dec x0 y).
    + rewrite e.
```

```
      reflexivity.
    + destruct (string_dec x0 x).
      * subst. reflexivity.
      * rewrite t_update_neq.
        { destruct (string_dec x0 x).
          - apply eqb_string_false_iff in n0.
            apply eqb_string_true_iff in e.
            rewrite n0 in e.
            discriminate e.
          - repeat (try (rewrite t_update_neq);
            try reflexivity;
            try (apply not_sym; assumption)).
          }
        apply not_sym.
        assumption.
Qed.
```

4.9 Procedures

4.9.1 Dynamic Procedures

Running example We can prove that using dynamic procedure scope the following program will lead to a state when y has value 6.

```
Definition procprogram :=
  BEGIN var x := 0,
    (proc p is ( x ::= x * 2 );
    proc q is ( CALL p )),
    BEGIN var x := 5,
      proc p is ( x ::= x + 1 ),
      ( (CALL q) ; y ::= x )
    END
  END.

Theorem dynamic_proof :
  empty |- << Some procprogram, x!->0 >>
 -->
  (y!->6, x!->0).
Proof.
  eapply stm_eq_env.
  - unfold procprogram.
    eapply block_ns with (x := x).
    + eapply var_ns.
      * reflexivity.
      * eapply none_ns.
    + eapply block_ns with (x := x).
```

```
      * eapply var_ns.
        { reflexivity. }
        { eapply none_ns. }
      * eapply comp_ns.
        { simpl. eapply call_rec_ns.
          eapply call_rec_ns.
          eapply ass_ns.
          reflexivity. }
        { eapply ass_ns.
          reflexivity. }
  - simpl.
    unfold var_update.
    simpl.
    apply functional_extensionality.
    intros.
    destruct (string_dec x0 x).
    + rewrite e.
      reflexivity.
    + destruct (string_dec x0 y).
      * subst. reflexivity.
      * rewrite t_update_neq.
        { repeat (try (rewrite t_update_neq);
          try reflexivity;
          try (apply not_sym; assumption)). }
        { apply not_sym. assumption. }
Qed.
```

4.9.2 Static Procedures

Running example We can prove that using static procedure scope the following program will lead to a state when y has value 10.

```
Theorem static_proof :
  empty_envp |- << procprogram, x!->0 >>
 -->
  (y!->10, x!->0).
Proof.
  eapply stm_eq_env.
  - unfold procprogram.
    eapply block_ns with (x:=x).
    + eapply var_ns.
      * reflexivity.
      * eapply none_ns.
    + simpl. eapply block_ns with (x := x).
      * eapply var_ns.
        { reflexivity. }
        { eapply none_ns. }
      * eapply comp_ns.
        { simpl.
          eapply call_ns.
          simpl.
          eapply call_ns.
          simpl.
          apply ass_ns.
          reflexivity.
        }
          eapply ass_ns.
          reflexivity.
  - simpl.
    unfold var_update.
    simpl.
    apply functional_extensionality.
    intros.
    destruct (string_dec x0 x).
    + rewrite e.
      reflexivity.
    + destruct (string_dec x0 y).
      * subst. reflexivity.
      * rewrite t_update_neq.
        { repeat (try (rewrite t_update_neq);
          try reflexivity;
          try (apply not_sym; assumption)). }
        { apply not_sym. assumption. }
Qed.
```

We can also prove that the dynamic procedure scope will lead to y = 6 with this definition of the environment. However, this is very similar to the proof above so we will omit the proof here.

Chapter 5

Extensions

The formalization of the framework and application of the framework chapters concerned topics discussed in the course Semantics and Correctness. There are many extensions of **While** that can be formalized. In this chapter we will discuss two extensions. First, we will discuss non-determinism. Non-determinism is also a topic in the book by Nielson and Nielson [10] and it was briefly mentioned during the Semantics and Correctness course. Secondly, we will discuss an extensions that contains break and continue statements. Popular imperative programming language like Java also have a break and continue statements [12]. In the course Semantics and Correctness these statements are not discussed. However, in the follow-up course Semantics and Rewriting they are discussed. This makes it interesting to also try to formalize these statements. We will discuss the formalization and the application of the topic in the same subsection.

5.1 Non-determinism

Previously, we already defined that semantics are deterministic if executing the same statement from the same initial state, should give the same final state. In the next chapter we proved that the natural semantics for **While** are deterministic. However, we can extend **While** in a way such that it is not deterministic anymore. For example, with a non-deterministic `or` statement which can execute the first or the second statement. We extend the natural semantics with the following rules.

$$[\mathrm{or}^1_{\mathrm{ns}}] \qquad \frac{\langle S_1, s\rangle \to s'}{\langle S_1 \ \texttt{or}\ S_2, s\rangle \to s'}$$

$$[\mathrm{or}^2_{\mathrm{ns}}] \qquad \frac{\langle S_2, s\rangle \to s'}{\langle S_1 \ \texttt{or}\ S_2, s\rangle \to s'}$$

Table 14: The natural semantics for the `or` rules

The syntax of the extended version of **While** in Coq now looks like this:

```
Inductive Stm : Type :=
  | ass (x : string) (a : Aexp)
  | skip
  | comp (s1 s2 : Stm)
  | if_ (b : Bexp) (s1 s2 : Stm)
  | while (b : Bexp) (s : Stm)
  | or (S1 S2 : Stm).
```

Moreover, the `or` rules are added to the `Seval` function as follows:

```
  | or_1_ns : forall st st' s1 s2,
      Seval st s1 st' ->
      Seval st (or s1 s2) st'
  | or_2_ns : forall st st' s1 s2,
      Seval st s2 st' ->
      Seval st (or s1 s2) st'.
```

Now the formalization is finished and we can prove that the same statement and initial state can lead to different final states. The program `x := 1 or (x := 2; x := x+2)` starting in a state where `x = 0` can lead to a final state where `x = 1` or `x = 4`. We can represent this in Coq as follows:

```
Example nondet1 :
  << x ::= 1 OR (x ::= 2; x ::= x + 2), x!->0 >>-->(x!->1).
Proof.
  eapply stm_eq.
  - eapply or_1_ns.
    eapply ass_ns.
    reflexivity.
  - eq_states.
Qed.

Example nondet2 :
  << x ::= 1 OR (x ::= 2; x ::= x + 2), x!->0 >>-->(x!->4).
Proof.
  eapply stm_eq.
  - eapply or_2_ns.
    eapply comp_ns.
    + eapply ass_ns.
      reflexivity.
    + eapply ass_ns.
      reflexivity.
  - eq_states.
Qed.
```

In this example, we get two different proof trees. However, it is also possible to get only one proof tree. This is because if a statement and state combination does not terminate in natural semantics, then it is not possible to make a derivation tree. For example, if we change `x := 1` into `while true do skip` in the previous example we only get one proof tree. We can try to represent this in Coq but its quite difficult to show that no derivation tree exists because the program has an infinite loop.

```
Example nondet3 :
 << (WHILE true DO SKIP) OR (x ::= 2; x ::= x + 2), x!->0 >>
-->(x!->4).
Proof.
  eapply stm_eq.
  - eapply or_1_ns.
    eapply while_tt_ns.
    + reflexivity.
    + eapply skip_ns.
    + eapply while_tt_ns.
(* There is no derivation tree for this because it loops forever.
Therefore, the statement is not provable using or_1_ns. *)
    Admitted.

Example nondet4 :
 << (WHILE true DO SKIP) OR (x ::= 2; x ::= x + 2), x!->0 >>
-->(x!->4).
Proof.
  eapply stm_eq.
  - eapply or_2_ns.
    eapply comp_ns.
    + eapply ass_ns.
      reflexivity.
    + eapply ass_ns.
      reflexivity.
  - eq_states.
Qed.
```

5.2 Break and Continue

5.2.1 Break

The break statement causes the loop around the break statement to terminate. This means that if a break occurs in the body of a while statement, every statement after the break is not executed. The break statement was partly formalized in the chapter *Imp* of Software Foundations [13], the actual natural semantics rules were left as exercise. We can extend the syntax of **While** to include the break statement

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

We can define this in Coq as follows:

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

To be able to skip all the statements after a break we need some break status that indicates whether a break has occurred or not. This means we need to modify the transition to the following form:

$$\langle S, s\rangle \to (s', B)$$

Here B indicates the meta-variable for the break status. We can indicate that a break has occurred with $\bullet$ and indicate that a break has not occurred with $\circ$. We can also define this in Coq with the following inductive type:

```
Inductive Bstatus : Type :=
  | b
  | no_b.
```

Using the definition of what a break statement should do we can make the following natural semantics rule for this extensions of **While**:

$[\text{break}_{\text{ns}}]$	$\langle \texttt{break}, s\rangle \to (s, \bullet)$	
$[\text{ass}_{\text{ns}}]$	$\langle x := a, s\rangle \to (s[x \mapsto \mathcal{A}[\![a]\!]s, \circ)$	
$[\text{skip}_{\text{ns}}]$	$\langle \texttt{skip}, s\rangle \to (s, \circ)$	
$[\text{comp}^{\bullet}_{\text{ns}}]$	$\dfrac{\langle S_1, s\rangle \to (s', \bullet)}{\langle S_1; S_2, s\rangle \to (s', \bullet)}$	
$[\text{comp}^{\circ}_{\text{ns}}]$	$\dfrac{\langle S_1, s\rangle \to (s', \circ) \quad \langle S_2, s'\rangle \to (s'', B)}{\langle S_1; S_2, s\rangle \to (s'', B)}$	
$[\text{if}^{\text{tt}}_{\text{ns}}]$	$\dfrac{\langle S_1, s\rangle \to (s', B)}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s\rangle \to (s', B)}$	if $\mathcal{B}[\![b]\!]s = \mathbf{tt}$
$[\text{if}^{\text{ff}}_{\text{ns}}]$	$\dfrac{\langle S_2, s\rangle \to (s', B)}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s\rangle \to (s', B)}$	if $\mathcal{B}[\![b]\!]s = \mathbf{ff}$

$[\text{while}^{\text{tt}\bullet}_{\text{ns}}]$	$\dfrac{\langle S, s\rangle \rightarrow (s', \bullet)}{\langle \texttt{while}\ b\ \texttt{do}\ S, s\rangle \rightarrow (s', \circ)}$	if $\mathcal{B}[\![b]\!]s = \mathbf{tt}$
$[\text{while}^{\text{tt}\circ}_{\text{ns}}]$	$\dfrac{\langle S, s\rangle \rightarrow (s', \circ) \quad \langle \texttt{while}\ b\ \texttt{do}\ S, s'\rangle \rightarrow (s'', B)}{\langle \texttt{while}\ b\ \texttt{do}\ S, s\rangle \rightarrow (s'', \circ)}$	if $\mathcal{B}[\![b]\!]s = \mathbf{tt}$
$[\text{while}^{\text{ff}}_{\text{ns}}]$	$\langle \texttt{while}\ b\ \texttt{do}\ S, s\rangle \rightarrow (s, \circ)$ if $\mathcal{B}[\![b]\!]s = \mathbf{ff}$	

Table 15: The natural semantics for While with the break statement

We can define the rules in Coq as follows:

```
Inductive Seval : State -> Stm -> State -> Bstatus -> Prop :=
  | ass_ns  : forall st a1 n x,
      Aeval st a1 = n ->
      Seval st (x ::= a1) (t_update st x n) no_b
  | skip_ns : forall st,
      Seval st SKIP st no_b
  | comp_b_ns : forall s1 s2 st st',
      Seval st s1 st' b ->
      Seval st (s1 ; s2) st' b
  | comp_no_b_ns : forall s1 s2 st st' st'' B,
      Seval st s1 st' no_b ->
      Seval st' s2 st'' B ->
      Seval st (s1 ; s2) st'' B
  | if_tt_ns : forall st st' b1 s1 s2 B,
      Beval st b1 = true ->
      Seval st s1 st' B ->
      Seval st (IF_ b1 THEN s1 ELSE s2) st' B
  | if_ff_ns : forall st st' b1 s1 s2 B,
      Beval st b1 = false ->
      Seval st s2 st' B ->
      Seval st (IF_ b1 THEN s1 ELSE s2) st' B
  | while_tt_b_ns : forall st st' b1 s1,
      Beval st b1 = true ->
      Seval st s1 st' b ->
      Seval st (WHILE b1 DO s1) st' no_b
  | while_tt_no_b_ns : forall st st' st'' b1 s1 B,
      Beval st b1 = true ->
      Seval st s1 st' no_b ->
      Seval st' (WHILE b1 DO s1) st'' B ->
      Seval st (WHILE b1 DO s1) st'' no_b
  | while_ff_ns : forall b1 st s1,
      Beval st b1 = false ->
      Seval st (WHILE b1 DO s1) st no_b
  | break_ns : forall st,
      Seval st (BREAK) st b
where "'<<' s ',' st '>>' '-->' '(' st' ',' br ')'" :=
      (Seval st s st' br).
```

As example we will take the program from the Semantic and Rewriting lecture [5]

```
Definition program1 :=
  x ::= 5;
  y ::= 3;
  WHILE (x <= 7) DO
    x ::= x + 2;
    (IF_ ~(x <= 8) THEN BREAK ELSE SKIP);
    y ::= y*3.
```

This program should break out of the second while loop and end with the values `x = 9`, `y = 9`.

```
Example proof1 :
  << program1, x!->0 >>-->((x!->9, y!->9), no_b).
Proof.
  unfold program1.
  eapply stm_eq.
  - eapply comp_no_b_ns.
    + apply ass_ns.
      reflexivity.
    + eapply comp_no_b_ns.
      * apply ass_ns.
        reflexivity.
      * eapply while_tt_no_b_ns.
        { reflexivity. }
        { eapply comp_no_b_ns.
          - apply ass_ns.
            reflexivity.
          - eapply comp_no_b_ns.
            + eapply if_ff_ns.
              * reflexivity.
              * apply skip_ns.
            + apply ass_ns.
                reflexivity.
        }
        { eapply while_tt_b_ns.
          - reflexivity.
          - eapply comp_no_b_ns.
            + apply ass_ns.
```

```
              reflexivity.
            + eapply comp_b_ns.
              eapply if_tt_ns.
              * reflexivity.
              * apply break_ns.
        }
  - eq_states.
Qed.
```

Similar to this, we can prove that if the break statement does not occur in a while loop then the program should end with a 'break has occurred'-status. For this we use almost the same program. Due to the break occurring early in the program, all the other statements are skipped and the proof is very short.

```
Definition program2 :=
  x ::= 5;
  BREAK;
  y ::= 3;
  WHILE (x <= 7) DO
    x ::= x + 2;
    (IF_ ~(x <= 8) THEN BREAK ELSE SKIP);
    y ::= y*3.

Example proof2 :
  << program2, x!->0 >>-->((x!->5, x!->0), b).
Proof.
  unfold program2.
  apply comp_no_b_ns with (x!->5, x!->0).
  - apply ass_ns.
    reflexivity.
  - apply comp_b_ns.
    apply break_ns.
Qed.
```

5.2.2 Continue

If a continue occurs inside a while loop then the statements after the continue are not executed but then loop condition is evaluated again. The continue extension of **While** is very similar to the break extension. The syntax looks as follows

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

In Coq the syntax is defined as follows:

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

The new transition uses a continue status instead of a break status. This status is indicated by the meta-variable C. For continue we will use the symbol $\blacksquare$ to indicate a continue has occurred and $\square$ to indicate no continue has occurred. The transitions look like this:

$$\langle S, s\rangle \to (s', C)$$

In Coq the we can define an inductive type for the continue status:

```
Inductive Cstatus : Type :=
  | c
  | no_c.
```

Now we can define the natural semantics rules for this extension of **While**:

$[\text{continue}_{\text{ns}}]$	$\langle \texttt{continue}, s\rangle \to (s, \blacksquare)$	
$[\text{ass}_{\text{ns}}]$	$\langle x := a, s\rangle \to (s[x \mapsto \mathcal{A}[\![a]\!]s, \square)$	
$[\text{skip}_{\text{ns}}]$	$\langle \texttt{skip}, s\rangle \to (s, \square)$	
$[\text{comp}_{\text{ns}}^{\blacksquare}]$	$\dfrac{\langle S_1, s\rangle \to (s', \blacksquare)}{\langle S_1; S_2, s\rangle \to (s', \blacksquare)}$	
$[\text{comp}_{\text{ns}}^{\square}]$	$\dfrac{\langle S_1, s\rangle \to (s', \square) \quad \langle S_2, s'\rangle \to (s'', C)}{\langle S_1; S_2, s\rangle \to (s'', C)}$	
$[\text{if}_{\text{ns}}^{\text{tt}}]$	$\dfrac{\langle S_1, s\rangle \to (s', C)}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s\rangle \to (s', C)}$	if $\mathcal{B}[\![b]\!]s = \mathbf{tt}$
$[\text{if}_{\text{ns}}^{\text{ff}}]$	$\dfrac{\langle S_2, s\rangle \to (s', C)}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s\rangle \to (s', C)}$	if $\mathcal{B}[\![b]\!]s = \mathbf{ff}$
$[\text{while}_{\text{ns}}^{\text{tt}\blacksquare}]$	$\dfrac{\langle S, s\rangle \to (s', \blacksquare) \quad \langle \texttt{while}\ b\ \texttt{do}\ S, s'\rangle \to (s'', C)}{\langle \texttt{while}\ b\ \texttt{do}\ S, s\rangle \to (s'', \square)}$	if $\mathcal{B}[\![b]\!]s = \mathbf{tt}$
$[\text{while}_{\text{ns}}^{\text{tt}\square}]$	$\dfrac{\langle S, s\rangle \to (s', \square) \quad \langle \texttt{while}\ b\ \texttt{do}\ S, s'\rangle \to (s'', C)}{\langle \texttt{while}\ b\ \texttt{do}\ S, s\rangle \to (s'', \square)}$	if $\mathcal{B}[\![b]\!]s = \mathbf{tt}$
$[\text{while}_{\text{ns}}^{\text{ff}}]$	$\langle \texttt{while}\ b\ \texttt{do}\ S, s\rangle \to (s, \square)$ if $\mathcal{B}[\![b]\!]s = \mathbf{ff}$	

Table 16: The natural semantics for While with the continue statement

We can define this in Coq as follows:

```
Inductive Seval : State -> Stm -> State -> Cstatus -> Prop :=
  | ass_ns  : forall st a1 n x,
      Aeval st a1 = n ->
      Seval st (x ::= a1) (t_update st x n) no_c
  | skip_ns : forall st,
      Seval st SKIP st no_c
  | comp_c_ns : forall s1 s2 st st',
      Seval st s1 st' c ->
      Seval st (s1 ; s2) st' c
  | comp_no_c_ns : forall s1 s2 st st' st'' C,
      Seval st s1 st' no_c ->
      Seval st' s2 st'' C ->
      Seval st (s1 ; s2) st'' C
  | if_tt_ns : forall st st' b1 s1 s2 C,
      Beval st b1 = true ->
      Seval st s1 st' C ->
      Seval st (IF_ b1 THEN s1 ELSE s2) st' C
  | if_ff_ns : forall st st' b1 s1 s2 C,
      Beval st b1 = false ->
      Seval st s2 st' C ->
      Seval st (IF_ b1 THEN s1 ELSE s2) st' C
  | while_tt_c_ns : forall st st' st'' b1 s1 C,
      Beval st b1 = true ->
      Seval st s1 st' c ->
      Seval st' (WHILE b1 DO s1) st'' C ->
      Seval st (WHILE b1 DO s1) st' no_c
  | while_tt_no_c_ns : forall st st' st'' b1 s1 C,
      Beval st b1 = true ->
      Seval st s1 st' no_c ->
      Seval st' (WHILE b1 DO s1) st'' C ->
      Seval st (WHILE b1 DO s1) st'' no_c
  | while_ff_ns : forall b1 st s1,
      Beval st b1 = false ->
      Seval st (WHILE b1 DO s1) st no_c
  | continue_ns : forall st,
      Seval st (CONTINUE) st c
where "'<<' s ',' st '>>' '-->' '(' st' ',' con ')'" :=
      (Seval st s st' con).
```

We can reuse the program for break to show that the continue statement works correctly. We need to change all the break statements into continue statements. After this, we can prove that we will end in a final state where x = 9, y = 9.

```
Definition program1 :=
  x ::= 5;
  y ::= 3;
  WHILE (x <= 7) DO
    x ::= x + 2;
    (IF_ ~(x <= 8) THEN CONTINUE ELSE SKIP);
    y ::= y*3.

Example proof1 :
  << program1, x!->0 >>-->((x!->9, y!->9), no_c).
Proof.
  unfold program1.
  eapply stm_eq.
  - eapply comp_no_c_ns.
    + apply ass_ns.
      reflexivity.
    + eapply comp_no_c_ns.
      * apply ass_ns.
        reflexivity.
      * eapply while_tt_no_c_ns.
        { reflexivity. }
        { eapply comp_no_c_ns.
          - apply ass_ns.
            reflexivity.
          - eapply comp_no_c_ns.
            + eapply if_ff_ns.
              * reflexivity.
              * apply skip_ns.
            + apply ass_ns.
                reflexivity.
        }
        { eapply while_tt_c_ns.
          - reflexivity.
          - eapply comp_no_c_ns.
            + apply ass_ns.
              reflexivity.
            + eapply comp_c_ns.
              eapply if_tt_ns.
              * reflexivity.
              * apply continue_ns.
          - eapply while_ff_ns.
            reflexivity.
        }
  - eq_states.
Qed.
```

We can also prove that if the continue statement does not occur in a while loop then the program should end with a 'continue has occurred'-status.

```
Definition program2 :=
  x ::= 5;
  CONTINUE;
  y ::= 3;
  WHILE (x <= 7) DO
    x ::= x + 2;
    (IF_ ~(x <= 8) THEN CONTINUE ELSE SKIP);
    y ::= y*3.

Example proof2 :
 << program2, x!->0 >>-->((x!->5, x!->0), c).
Proof.
  unfold program2.
  apply comp_no_c_ns with (x!->5, x!->0).
  - apply ass_ns.
    reflexivity.
  - apply comp_c_ns.
    apply continue_ns.
Qed.
```

Chapter 6

Experiment

This formalization of natural semantics, as well as the one of structural operational semantics written in a parallel thesis by Elitsa Bahovska [1], is primary meant to serve as a studying aid in the course Semantics and Correctness. Therefore, we tested whether it is indeed helpful using an experiment.

6.1 Research question

The research question of this experiment is: Does the formalization of natural semantics and structural operational semantics in Coq help Semantics and Correctness students to make proofs on paper?

6.2 Experiment

For this experiment we want to learn if having a Coq formalization of proof can help Semantics and Correctness students to make the same proof on paper. For this, we used a comparative study. We asked a series of questions on the difficultly level of certain Semantics and Correctness topics before and after having seen the formalization. Because of the limited amount of time we had for the experiment, we only tested if the formalization helped to make proofs on paper for two topics, induction on the shape of the derivation tree and induction on the length of the derivation tree. We think these topics were considered difficult by many students of the course, so they will give an accurate representation of whether the formalization helps students or not. Additionally, we asked some questions about their experience, the explanation and if they would like to see it incorporated into the course after they have seen the formalization. We included these questions because the experiment will be used to decide whether it is beneficial to include Coq proofs in the course.

6.2.1 Participants

The formalization covers all the topics that are also covered in the course Semantics and Correctness. This means that to test whether the formalization can help students, we need participants that are following, or have followed the course Semantics and Correctness. The formalization should typically be used during the learning phase. If the participants already passed the course, they understand all topics to a passable level. In that case, explaining the formalization and testing whether it helps them make proofs on paper is not useful. Therefore, we need participants who are still in the learning phase of the course Semantics and Correctness.

6.2.2 Variables

For our analysis, we use just one binary independent variable - the participants have had no contact with the formalization versus the formalization 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 - induction proofs for natural semantics and structural operational semantics - on a scale from one to ten.

6.2.3 Research hypothesis

We hypothesize that there will be a decrease in the perceived difficulty of the two Semantics and Correctness topics after introduction to the formalization in comparison to before.

6.3 Setup

The experiment is setup to be a within subject experiment. This means we will use the same participants for the control group as the test group. We can afford to do that, as we are comparing whether spending time on the formalization is better than not doing so. We are not testing whether spending time on the formalization is better than spending the same amount of time on studying the book, slides or homework.

6.3.1 Baseline - before

The participants are first asked to fill in a questionnaire, to establish how they perceive the difficulty of induction proofs in natural semantics and structural operational semantics. Later on, their average updated opinions will be compared to this baseline.

6.3.2 Task

A meeting of one hour and forty five minutes is conducted with the participants. During the meeting the formalization 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 up too much time for the participants, we considered a meeting to be the better option. A text would take longer to read and it does not showcase exactly what is happening in Coq. Moreover, regardless of whether it is text or a lecture, the topic would be the same, an explanation of the basic formalization and some example proofs that use the formalization. The used Coq file for natural semantics can be found in Appendix B. The file uses the formalization and examples from previous chapters, but with some more comments on how this relates to the proofs on paper.

Both the natural semantics and structural operational semantics files contain a proof that will be discussed in detail and a proof that is similar that will not be discussed. For natural semantics a proof of induction on the shape of the derivation tree will be discussed. That first half of that proof was also covered in the lecture slides of the course. They have to make the other half of the proof on paper. They are allowed to spend as much time as they would like, but are asked to deliver the proof within a few days from the meeting. That is the actual task, where they can try out how the formalization works and decide whether using it helps them to understand the proofs on paper.

6.3.3 Measuring the depending variables - after

After the proofs on paper are submitted, the participants are asked to fill in a second questionnaire with their updated perception of the difficulty of the two topics. A comparison can then be made to see if there is a change in their opinion. This comparison will only be made for the average opinion of the whole group. We decided to make the questionnaires anonymous and the first and second questionnaire are not connected, thus we cannot compare individual results.

Finally, a second meeting is conducted to provide feedback on the submitted proofs on paper. Moreover, we will also ask the participants to share their experiences and give feedback on what can be improved to make the explanation or formalization better. Most importantly, if they would like to see the formalization in the course and in what form.

6.3.4 Avoiding confounding variables and possible modification

In this section 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 discuss some of the choices that were made for the experiment.

Different study ways per participant

We only measured the overall change in difficulty to see if it helps the groups as a whole to better understand the proofs. Naturally, the participants would have different levels and capacity of understanding the material. This means that for some participants the change in difficulty might be different than for others. This indicates that it might be interesting to compare each of the Semantics and Correctness students to their own baseline. However, this was not done in this experiment due to a lack of time. It might also be that some students had previous experiences with Coq that influence how well they are able to understand or are willing to understand the Coq formalization. For that purpose, we added a question in the first questionnaire about their previous experiences with Coq.

Varying amount of time invested

The experiment was conducted just before an exam period, which is typically a busy time for students. This might indicate that students will have a varying amount of time to spend on the formalization and making the proof on paper. We do not ask how much time they spend on making the proof. This is because students also spend varying time on a course. Furthermore, we want to give the participants the freedom to experiment with the formalization as long as they want and not give a maximum amount of time. However, the time invested in making the proof on paper might effect other variables. For example, if a student needs more time to understand the formalization it will affect their response of perceived difficulty. If a lot of time is spend on trying to understand the formalization, they will 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 time to make the proof, but not as much. A way to avoid both effects is to make a between-subject designed experiment, where a control group spends a fixed amount of time studying the already provided material, and a test group spends the same amount of time studying the formalization. However, in the current situation, that is not possible due to a lack of participants and time is needed to understand and get familiar with the formalization. In future research, a between-subject experiment could be tested with more carefully selected participants, who are familiar with both the Coq formalization and the material of Semantics and Correctness. In 6.4 we will show another reason why such a design is not well applicable in this experiment.

Finally, our participants have an incentive to spend time on the course because it is a mandatory course for certain programs. Thus they are assumed to have spent some time on studying the conventional materials. Moreover, we assume they already have an opinion on how difficult they find the topics and whether spending a few more hours on them might improve their understanding of the material.

Using grades as measuring the effect, instead of perceived difficulty

If we were to use testing and grades to measure our depending variable, the three confounding variables mentioned above would have much bigger effects. Additionally, we would need to account for student's 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 formalization.

6.4 Execution

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

The main challenge in this experiment is the timing. This thesis is written in the third and fourth quarter of the study year. However, the course Semantics and Correctness is run during the first quarter this year. Therefore, the only students that fulfill the requirements we discussed above for participants, are students who did not pass the final exam. Out of this already very limited set of possible participants, only five agreed to participate in the experiment.

6.4.1 Pros

The small sample size makes a nice personal relation with the participants and all their questions to be answered. Because of this, it is possible to give feedback on all submitted proofs, as well as receive much needed feedback. As this is the first version of this formalization, feedback from students on how the studying aid can be improved to better suit their needs is very much appreciated.

Additionally, most of these participants have spent at least some time on trying to understand the material. Therefore, the few hours they can spend on the formalization are just a small percentage of what the total time spent on the course and the confounding variable of any time spent on the subjects being an improvement is limited.

6.4.2 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. This means that we will not be able to test if the formalization significantly helps the students. Therefore it is highly recommended that the experiment, with the same design, is repeated while the course Semantics and Correctness is running.

6.5 Results

For the experiment we gathered data in different ways. We used questionnaires to get quantitative data which we could compare to before the Coq tutorial. We also gathered qualitative data in the form of feedback that was given during the tutorial, in the open questions from the questionnaire and in the final meeting. As mentioned above, the average difficulty score before and after the meeting will be compared to see if the formalization helps the participants understand the proofs better.
It should be mentioned that the first questionnaire was completed by five student but only three students completed the second questionnaire. Because the experiment was done in a short time and with a small sample size, we assumed that everybody would be able to complete the whole experiment. This also means that we made the questionnaires anonymous, it is not possible to see who filled in which answer. More

importantly, it is not possible to compare individual participant results before and after the formalization. Unfortunately, one person had to drop out due to personal reasons and another person did complete all other steps but did not fill in the second questionnaire. Because the questionnaire was anonymous we cannot correct for the people who dropped out since we do not know which answers were theirs in the first questionnaire. Because it is not possible to only consider the answers of the participants that filled in both questionnaires, we decided to use all the available results. This makes the average scores in the next part less reliable. Therefore, we will put more emphasis on the feedback part on the experiment than the average difficulty scores before and after having seen the formalization.

6.5.1 Questionnaires

Before the tutorial

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.

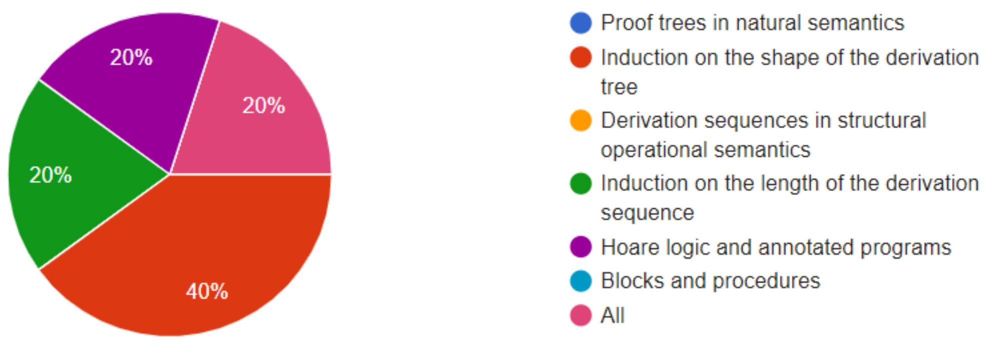

Figure 6.1: Which Semantics and Correctness topic did you find the most difficult?

As can be seen in Figure 6.1, induction proofs were chosen by 3 out of 5 students. Another participant chose the option 'All', this includes induction proofs. 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 proofs. Induction on the shape of the derivation tree has an average difficulty of 6.8/10. This was because it was difficult to understand the concept, apply the induction step correctly and difficult names were used in the proof. Induction on the length of the derivation sequence has an average difficulty of 6.2/10. Some people found this more difficult because there is no visual representation. Other people found it easier because it uses numbers instead of trees.

After this, we asked how much they liked making proofs in Coq. This question was asked because the participants might have previously used Coq. The previous experiences might influence their reception to the Coq formalization. The average score for this question was 5.4/10. This indicates that the average opinion about Coq was neutral.

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

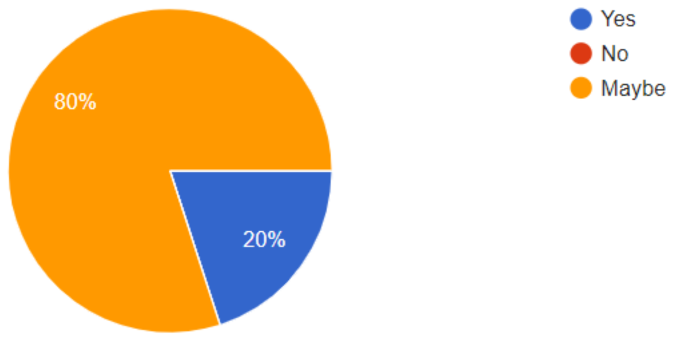

Figure 6.2: Do you think seeing natural semantics and operational structural semantics proofs in Coq would help you to make proofs on paper?

As can be seen in Figure 6.2, most participants think a formalization in Coq might help them. They think it might help because it easier to correct mistakes in Coq. Moreover, Coq can give real-time feedback on if you are taking a step that is impossible. It is easier to directly see the results of your actions. However, there might also be downsides because it is necessary to learn to work with the formalization while learning how the actual proofs work. Because you are learning multiple things at once, it might get confusing.

After the tutorial

The first question asked if they found the tutorial helpful. The average score for this question was 3.67/10. This is very low and there are multiple possible explanations for this. For example, the time between the tutorial and making the proofs was quite long. It might be that parts of the explanation were already forgotten. It might also be because, as can be seen in one of the next questions, some participants did not actually use the formalization to make the proof on paper. It is important to note that one participant did score the usefulness of the tutorial with a 7/10. This leads us to believe that the usefulness of the tutorial and the Coq formalization may also depend on the participants themselves.
In the next question, we asked if the participants were able to finish the proofs and which materials they used to make their proof.

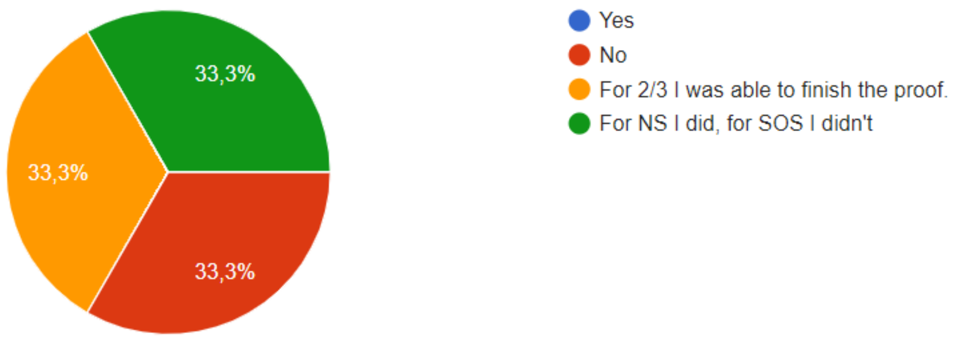

Figure 6.3: Did you manage to finish the proofs on paper?

As can be seen from Figure 6.5, none of the participants actually managed to complete the both proofs fully. However, some participants did finish quite a big part of the proofs.

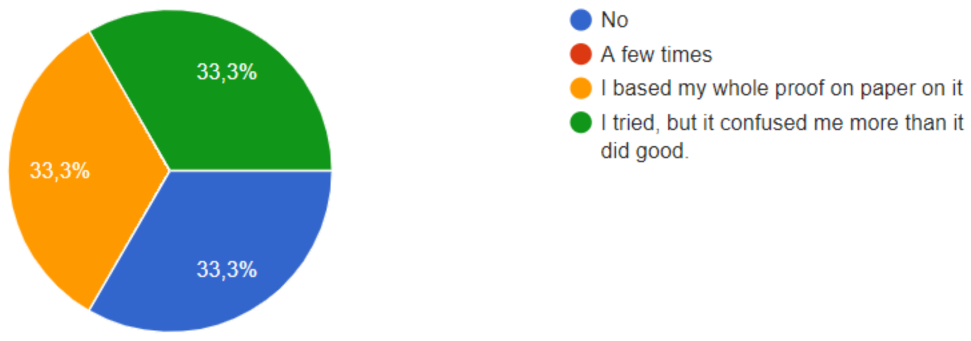

Figure 6.4: Did you use the proofs in Coq while making the proof on paper?

Most participants used the slides but some did not use the Coq formalization at all. Since the experiment was set up to use the Coq formalization to make proofs on paper, this can affect the other results.
After that, they were asked to rate how difficult induction on the shape of the derivation tree and induction on the length of the derivation sequence was. This was done to compare them to the results from before the tutorial. The average score for induction on the shape of the derivation tree went from 6.8/10 to 5.7/10. This means the average difficulty decreased. The average score for induction on the length of the derivation sequence went from 6.2/10 to 6.7/10. This indicates that the difficulty of induction on the length of the derivation sequence increased after having seen the formalization. We decided to report the average difficulty scores of the induction proofs because that was the main goal of the experiment. However, due to participants dropping out and using a not corrected average score, these values do not mean anything. It can be that the participants that found natural semantics induction proofs difficult dropped out, which decrease the average score. It can also be that the participants that found the

structural operational semantics induction proofs easy dropped out. There is no way to know this for sure, thereby making the average scores very unreliable. The only thing we can conclude is that the formalization did not make the average difficulty scores dramatically lower of higher.

Next, we asked if the participants would like to see some Coq proofs in the course Semantics and Correctness.

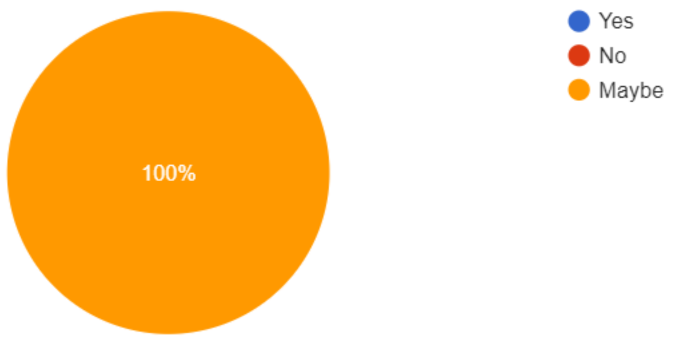

Figure 6.5: Would you like to see Coq proofs in the course Semantics and Correctness?

All participants answered 'Maybe'. When asked why they would like or would not like to see Coq in the course there were a couple of reasons. The formalization needs to be better documented to be able to use it. It increases the workload but it might help other students. Another participant said that it would be beneficial for natural semantics proof trees if they could use Coq to visually see what is happening in the tree when applying a certain tactic. The participants mentioned that some possible improvements for the formalization include: better documentation, a visual representation of the proof and better explanation of the tactics in relation to the proofs.

6.5.2 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 feedback points.

Visualization and interface 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 derivation tree that was displayed next to the proof. 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 formalization more understandable.

Many learning topics Moreover, the participants 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 formalization, the used notation, induction proofs in Coq and induction on the shape/length of the derivation tree/sequence.

Meaning of Coq rules There also needs to be a better introduction and more explanation before the formalization can be used in the course or in a next 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 and tactics work themselves.

Workload Furthermore, the participants showed concern for the increase of workload that might occur if Coq proofs are incorporated in the course. The course is already quite time consuming and has several difficult topics. Adding Coq proofs might make the workload too high. On the other hand, it could not hurt to have a working formalization available for students that are interested and understand Coq better.

Derivation trees 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 derivation tree proofs that were discussed during the tutorial would be useful. Most participants said that they would find that very helpful if there was some visual representation. This would be very useful because making trees in Coq is easier. It is easier to correct mistakes and check if certain rules can be applied. Moreover, we asked if there were other parts of the course where a Coq formalization 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.

Coq as solution Moreover, one participant mentioned that Coq might not be helpful for most students because most of the work is performed by `inversion` and that tactic was difficult to understand. According to that participant, looking at the Coq proof is equal to looking at the solution for the proof on paper. It was also mentioned that looking at solutions does not increase understanding. The Coq proof does not teach students how the proof should be made, it only shows a solution.

Benefit Next to all the concerns and possible improvements, the participants did mention that they were glad that somebody was looking into ways to make the course easier. Moreover, they mentioned that they might not be the best group of participants for the experiment because they already found the topics difficult. Other students might find the formalization more useful.

6.6 Experiment Discussion

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 participants 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 and the usefulness of the tutorial, as some information from the Coq tutorial might have been forgotten.
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 eight months ago. The results for the experiment are less reliable for these reasons. This 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, this should include Semantics and Correctness students during a run of the course and with a stricter deadline between the tutorial and proof on paper deadline. It would also be better to link the first and second questionnaire of a single participant. This would make the results more usable when certain participants drop out. It would also allow us to get results per participant. Finally, this group is too small to generalize the findings to all the Semantics and Correctness students. Even if none of the participants dropped out, the sample size was still too small to make any solid conclusions on if the experiment actually helps the students or not.

6.7 Experiment Conclusion

Currently, it is unclear whether the formalization helps students to understand the proofs better. However, there seem to be enough possible improvements that might make the formalization better. For example, a visual representation of the trees, exercises with the Coq formalization during the learning phase of the formalization and better documentation might make the formalization more usable. However, the workload must be taken into account if the Coq formalization is incorporated in the course.

Chapter 7

Conclusion

7.1 Research Question

To conclude, it is possible to make a Coq formalization of natural semantics that follows the theory and notation used in the course Semantics and Correctness. Our hypothesis that is was possible was correct. However, it is not as helpful as we hoped. The hypothesis for the experiment was correct, the difficulty score did for natural semantics decreased. However, it is unclear whether this is because the tutorial helped the students or because of other reasons. Finally, it seems that there are many possibilities to improve the formalization.

7.2 Challenges

During the process of making the formalization, some difficulties occurred. For example, formalizing **Num** in Coq turned out to be quite hard because many coercions were needed. The Hoare logic proof for the factorial took quite some time to finish because it used a factorial function for natural numbers instead of numerals like in the book. Moreover, the **Block** and **Proc** extensions needed to be made from scratch. There was no formalization of these concepts in Coq yet. Finally, the notations used were adapted many times to find the notation that matches the notation used in the course the best.

The experiment was also challenging because there was a limited time period for the experiment and some participants did not finish the complete experiment. This lead to inconclusive results. This means that we had to rely more on the feedback that was given by the students instead of quantitative results that would say more about how useful the formalization actually is.

7.3 Comparison to related work

In chapter 1 it was mentioned that quite a few formalizations already exist but they do not match the course. This formalization does match the course and the book by Nielson and Nielson better for the following reasons:

- Software Foundations uses natural numbers and not numerals, our formalization does include numerals.
- In Software Foundations induction on the shape of the derivation tree is not explained explicitly. This type of proof is a central topic in Semantics and Correctness. This formalization covers the topic using examples.
- This formalization has notation that matches the notation in the book better. For example, the transitions in Software Foundations have the form $st = [c] => st'$. In the book by Nielson and Nielson $\langle S, s\rangle \to\ s'$ is used. In the formalization we used the following notation: `<< S , s >>--> s '.`
- This formalization has many examples directly taken from the book or the slides. This helps to understand the formalization, a direct comparison between proofs in the book and in the formalization can be made.
- The formalization is in Coq, which is already familiar to most of the students. Other formalization were made in Isabelle or other proof assistants, which are not familiar to the students.

This list indicates that this formalization matches the book by Nielson and Nielson and the course Semantics and Correctness better.

7.4 Future research

The formalization could be improved in certain ways. For example, the Hoare logic chapter uses natural numbers instead of numerals. This is because the Hoare proofs needed many lemmas about numbers that were available in the Coq standard library for natural numbers, but would have to be made manually for the numerals. In future research, these lemmas can be formalized such that the Hoare logic chapter also uses numerals.

Logically, 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 the formalization that can be used to improve the formalization or its explanation.

The most obvious focus of future work should be on making the current formalization a better studying aid. To accomplish this we could use the feedback from the students to already improve it.

One of the conclusions from the experiment was that a visualization of the current state of the tree next to the proofs would be very useful. This is possible in the Coq server used for Logic and Applications, however this Coq server is outdated. It would be interesting to investigate if we can get the natural semantics formalization on the Coq server and implement the Coq server such that the trees are available next to the proof.

If there is more time, it would be interesting to find alternatives for the mapping of the states. Currently, the formalization uses the definition for total and partial maps from Software Foundations [13]. However, this definition does not match nicely with the notation used in the course. In the course we use $S_{5,7}$ and in Coq that would be `x !-> 5 , y !-> 7`.

Lastly, the course Semantics and Correctness has a follow-up course Semantics and Rewriting. The first lecture covers the Break and Continue statements. These are added in the extensions of **While** in the thesis. The next few lectures dive more into term rewrite systems and lambda calculus. Software Foundations has some chapters on simply typed lambda calculus. It would be interesting to research if some parts of this course can also be formalized in Coq. If it turns out that the formalization is useful in Semantics and Correctness, it would certainly be interesting to research this.

Acknowledgments

Firstly, I would like to thank my supervisors, Engelbert Hubbers and Freek Wiedijk. I would like to thank them for offering advice and feedback during the thesis process. Secondly, I would like to thank Elitsa Bahovska, who did a parallel thesis in structural operational semantics. Because I was always able to ask her questions whenever I got stuck. Lastly, I would like to thank the participants of our experiment. They gave very helpful feedback on the formalization and the explanation.

Bibliography

[1] Elitsa Bahovska. Structural operational semantics of imperative programming languages in Coq. 2020.

[2] Engelbert Hubbers. Annotated Programs, 2018. From the Brightspace page for Semantics and Correctness.

[3] Engelbert Hubbers. Logic and Applications 12: Mathematical induction and possibly modal logic, 2019. From the Brightspace page for Logic and Applications.

[4] Engelbert Hubbers. Semantics and Correctness 3" Semantic equivalence, 2019. From the Brightspace page for Semantics and Correctness.

[5] Engelbert Hubbers. Natural semantics of abrupt completion, 2020. From the Brightspace page for Semantics and Rewriting.

[6] CNRS Inria and contributors. Library Coq.Init.Datatypes, 1999-2020.

[7] CNRS Inria and contributors. Library Coq.Lists.ListSet, 1999-2020.

[8] CNRS Inria and contributors. Library Coq.Numbers.BinNums, 1999-2020.

[9] CNRS Inria and contributors. Tactic Index, 1999-2020.

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

[11] Tobias Nipkow and Gerwin Klein. Concrete Semantics with Isabelle/HOL, 2014.

[12] Oracle. Branching Statements.

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

[14] Norbert Schirmer. A Verification Environment for Sequential Imperative Programs in Isabelle/HOL, 2005.

Appendices

Appendix A: Questions

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: Coq File for the Experiment

Below is the code for the natural semantics part of the experiment. This includes the example of a simple proof tree and a proof by induction on the shape of the derivation tree.

```
(* 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.
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.

(*  Natural Semantics for While  *)
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').

(* The rules can be used to proof that a statement executed in
   an intial state will lead to the given final state. Just like
   the proofs trees on paper. Lets start with an easy example,
                       [ass_ns]                       [ass_ns]
 <z:=x, s_5_7_0> -> s_5_7_5  <x:=y, s_5_7_5> -> s_7_7_5
 ---------------------------------------------------[comp_ns]                        [ass_ns]
       < z:=x; x:=y, s_5_7_0 > -> s_7_7_5                < y:=z, s_7_7_5 > -> s_7_5_5
    -------------------------------------------------------------------------------[comp_ns]
                       < (z:=x; x:=y); y:=z, s_5_7_0 > -> s_7_5_5)
*)
Example Example1_NS :
  << (z::= x; x::= y); y::= z  , y!->7, x!->5 >>
 -->(y!->5, x!->7, z!->5, y!->7, x!->5).
Proof.
  apply comp_ns with (x!->7, z!->5, y!->7, x!->5).
  - apply comp_ns with (z!->5, y!->7, x!->5).
    + apply ass_ns. reflexivity.
    + apply ass_ns. reflexivity.
  - apply ass_ns. reflexivity.
Qed.

(* The final state is very big because every update adds a new
   mapping to the left. We can make this faster by using eapply.
   eapply applies the rule and uses a new variable that doesn't
   have a known value yet. Later in the proof it always becomes
   clear what the state should be, this is then filled in
   automatically by Coq. We also need some new state to eapply all
   these rules to and then we can later check if the final state
   given matches the state we calculated. This requires a theorem
   and a tactic. It is not needed to understand them, but it is
   useful to know that stm_eq should be the first thing in the proof.
   eq_states just checks if the two states are equal, it should be
   the last applied rule.*)

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

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.

Example Example2_1d :
  << (z::= x; x::= y); y::= z  , y!->7, x!->5 >>
 -->(y!->5, x!->7, z!->5).
Proof.
  eapply stm_eq.
  eapply comp_ns.
  - eapply comp_ns.
    + apply ass_ns. reflexivity.
    + apply ass_ns. reflexivity.
  - apply ass_ns. reflexivity.
  - eq_states.
Qed.


(* Induction on the shape of the derivation tree.
   We will redefine the syntax and semantics of
   While to include the repeat rules:

repeat_until_tt:          < S, s >-->s'
                 ------------------------------- if B[[b]]s'=tt
                  < repeat S until b, s >-->s'

repeat_until_ff: < S, s >-->s',  < repeat S until b, s' >-->s''
                 --------------------------------------------------- if B[[b]]s'=ff
                                < repeat S until b, s >-->s''
*)
```

```
Module Repeat.
Declare Scope rep.
Open Scope rep.

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

Bind Scope rep with StmRepeat.
Notation "x '::=' a" :=
  (ass x a) (at level 60) : rep.
Notation "'SKIP'" :=
   skip : rep.
Notation "s1 ; s2" :=
  (comp s1 s2) (at level 80, right associativity) : rep.
Notation "'WHILE' b 'DO' s " :=
  (while b s) (at level 80, right associativity) : rep.
Notation "'IF_' b 'THEN' s1 'ELSE' s2 " :=
  (if_ b s1 s2) (at level 80, right associativity) : rep.
Notation "'REPEAT' s1 'UNTIL' b1 " :=
  (repeat s1 b1) (at level 80, right associativity) : rep.

Inductive Seval : State -> StmRepeat -> State -> Prop :=
  | ass_ns  : forall st a1 n X,
      Aeval st a1 = n ->
      Seval st (X ::= a1) (t_update st X n)
  | skip_ns : forall st,
      Seval st SKIP st
  | comp_ns : forall s1 s2 st st' st'',
      Seval st s1 st' ->
      Seval st' s2 st'' ->
      Seval st (s1 ; s2) st''
  | if_tt_ns : forall st st' b1 s1 s2,
      Beval st b1 = true ->
      Seval st s1 st' ->
      Seval st (IF_ b1 THEN s1 ELSE s2) st'
  | if_ff_ns : forall st st' b1 s1 s2,
      Beval st b1 = false ->
      Seval st s2 st' ->
      Seval st (IF_ b1 THEN s1 ELSE s2) st'
  | while_tt_ns : forall st st' st'' b1 s1,
      Beval st b1 = true ->
      Seval st s1 st' ->
      Seval st' (WHILE b1 DO s1) st'' ->
      Seval st (WHILE b1 DO s1) st''
  | while_ff_ns : forall b1 st s1,
      Beval st b1 = false ->
      Seval st (WHILE b1 DO s1) st
  | repeat_until_tt_ns : forall st st' b1 s1,
      Seval st s1 st' ->
      Beval st' b1 = true ->
      Seval st (repeat s1 b1) st'
  | repeat_until_ff_ns : forall st st' st'' b1 s1,
      Seval st s1 st' ->
      Beval st' b1 = false ->
      Seval st' (repeat s1 b1) st'' ->
      Seval st (repeat s1 b1) st''.

Notation "<< s , st >>-->st'" := (Seval st s st')
                                  (at level 40).

(* Definitions of how the equivalences work *)
Definition Aequiv (a1 a2 : Aexp) : Prop :=
  forall (st : State),
    Aeval st a1 = Aeval st a2.
```

```
Definition Bequiv (b1 b2 : Bexp) : Prop :=
  forall (st : State),
    Beval st b1 = Beval st b2.

Definition Sequiv (S1 S2 : StmRepeat) : Prop :=
  forall (st st' : State),
    Seval st S1 st'<->Seval st S2 st'.


(* Proof from the slides *)

(*
Induction on the shape of derivation trees:
1. Prove that the property holds for all the simple derivation trees by
showing that it holds for the axioms of the transition system.

2. Prove that the property holds for all composite derivation trees: For
each rule assume that the property holds for its premises (this is
called the induction hypothesis) and prove that it also holds for the
conclusion of the rule provided that the conditions of the rule are
satisfied.

Property P(T):
For all states s and s' we have that if the conclusion of T is
 < repeat S until b, s > -> s'
then there exists a derivation tree T' with the conclusion
 < S; while ~b do S, s > -> s
*)

Theorem proof_slides :
  forall b S s s',
  << REPEAT S UNTIL b, s >>-->s'
  ->
  << S; WHILE ~b DO S, s >>-->s'.
Proof.
  (* Introduces the variables/hypothesis we will be talking about *)
  intros b S st st' HSe.
  remember (REPEAT S UNTIL b ) as Srepunt eqn:HeqSrepunt.
  induction HSe.
    - (* ass_ns / conclusion is not of the requested form*)
      inversion HeqSrepunt.
    - (* skip_ns / conclusion is not of the requested form *)
      inversion HeqSrepunt.
    - (* comp_ns / conclusion is not of the requested form *)
      inversion HeqSrepunt.
    - (* if_tt_ns / conclusion is not of the requested form *)
      inversion HeqSrepunt.
    - (* if_ff_ns / conclusion is not of the requested form *)
      inversion HeqSrepunt.
    - (* while_ff_ns / conclusion is not of the requested form *)
      inversion HeqSrepunt.
    - (* while_tt_ns / conclusion is not of the requested form *)
      inversion HeqSrepunt.
    - (* repeat_until_tt_ns *)
      (* inversion will try to find out as much as possible from the given
         hypothesis. For example, we know B[[b1]]st'=tt so we know the
         last applied rule must have been repeat_until_tt and using the IH
         a tree T1 must exist such that the tree looks like
                    T1
               ---------------------[..]
                  < S, st > -> st'
            -------------------------------[repeat_until_tt_ns]
             < repeat S until b, st > -> st' *)
       inversion HeqSrepunt; subst.
       (* We can use this to make the following tree
                  T1
          -----------------[..]                               [while_ff_ns]
           < S, st > -> st'       < while ~b do S, st'> -> s'
        ---------------------------------------------------[comp_ns]
                  < S; while ~b do S, st > -> st'
           We know [while_ff_ns] needs to be applied because B[[b1]]st=tt *)
       apply comp_ns with (st' := st').
       assumption.
       apply while_ff_ns.
       (* Proof B[[~b]] st'=ff from assumption B[[b]]st'=tt *)
       simpl.
       apply negb_false_iff.
       assumption.
    - (* repeat_until_ff_ns *)
      (* Again we use inversion but now we know B[[b1]]st'=ff. This
        means the last applied rule must be repeat_until_ff_ns.
        We know that trees T1 and T2 must exist such that the tree looks
        like
               T1                                T2
      ------------------[..]   -------------------------------[..]
       < S, st > -> st'        < repeat S until b, st' > -> st''
      ------------------------------------------------------[repeat_until_ff_ns]
           < repeat S until b, st > -> st''
       Note that the states are not in the normal order *)
       inversion HeqSrepunt; subst.
       (* Because we know tree T2 exists we can apply IH to get tree T2'
                           T2'
           --------------------------------[..]
           < S; while ~b do S, st' > -> st''
       *)
       assert (<< S; WHILE ~ b DO S, st' >>-->st'').
       + apply IHHSe2.
        reflexivity.
       (* Now we use inversion on < S; while ~b do S, st'> -> st''.
           This indicates that there must be some state st'0 and trees T3 and T4
                  T3                                T4
       ------------------[..]    ------------------------------[..]
       < S, st' > -> st'0        < while ~b do S, st'0 > -> st''
       -------------------------------------------------------[comp_ns]
                  <S; while ~b do S, st' > -> st''
       This means we can make the complete tree!
                                        T3                               T4
                            ------------------[..]    ------------------------------[..]
      T1                      < S, st' > -> st'0        < while ~b do S, st'0 > -> st''
-----------------[..]       -----------------------------------------------------------[while_tt_ns]
< S, st > -> st'                     <while ~b do S, st' > -> st''
---------------------------------------------------[comp_ns]
  <S; while ~b do S, st > -> st''
      *)
       + inversion H0; subst.
        apply comp_ns with (st' := st').
        * assumption.
        * apply while_tt_ns with (st' := st'0).
          (* B[[~b]]st'=tt is the same as B[[b]]st'=ff *)
          simpl.
          apply negb_true_iff.
          assumption.
          assumption.
          assumption.
Qed.


(* Now we still have to the other side.
   This was done in the assignment with a the following lemma.
   This lemma was needed to make the states match better *)
Lemma indirect_induction:
  forall b S s s' s'',
    << WHILE ~b DO S, s''>>-->s'
  ->
    <<S, s>>-->s''
  ->
    <<REPEAT S UNTIL b, s>>-->s'.
```

```
Proof.
  intros b S st st' st'' H0.
  remember (WHILE ~b DO S) as Swhile eqn:HeqSwhile.
  (* This next step is to get the states to match better *)
  generalize dependent st.
  induction H0.
  - (* ass_ns / conclusion is not of the requested form *)
    inversion HeqSwhile.
  - (* skip_ns / conclusion is not of the requested form *)
    inversion HeqSwhile.
  - (* comp_ns / conclusion is not of the requested form *)
    inversion HeqSwhile.
  - (* if_tt_ns / conclusion is not of the requested form *)
    inversion HeqSwhile.
  - (* if_ff_ns / conclusion is not of the requested form *)
    inversion HeqSwhile.
  - (* while_tt_ns, this is of the proper form *)
    (* Get information from B[[b1]]st=tt and while rule *)
     inversion HeqSwhile; subst.
    (* Get state and hypothesis in assumptions *)
     intros st''' H1.
     apply repeat_until_ff_ns with (st':=st).
     + assumption.
     + (* B[[~b]]st=tt is the same as B[[b]]st=ff *)
       simpl in H.
       apply negb_true_iff in H.
       assumption.
     + (* apply IH *)
       apply IHSeval2.
       reflexivity.
       assumption.
  - (* while_ff_ns, this is of the proper form *)
    (* Get information from B[[b1]]st=ff and while rule *)
    inversion HeqSwhile; subst.
    (* Get state and hypothesis in assumptions *)
    intros s H1.
    apply repeat_until_tt_ns.
    + apply H1.
    + (* B[[~b]]st=ff is the same as B[[b]]st=tt *)
      simpl in H.
      apply negb_false_iff in H.
      assumption.
  - (* repeat_until_tt_ns / conclusion is not of the requested form *)
    inversion HeqSwhile.
  - (* repeat_until_ff_ns / conclusion is not of the requested form *)
    inversion HeqSwhile.
Qed.

Theorem proof_assignment :
  forall b S s s',
   << S; WHILE ~b DO S, s >>-->s'
  ->
   << REPEAT S UNTIL b, s >>-->s'.
Proof.
  intros b S st st' HSe.
  inversion HSe; subst.
  (* Now we can apply the lemma with the proper state *)
  apply indirect_induction with (s'':=st'0).
  assumption.
  assumption.
Qed.

(* We can easily combine the two proofs to prove semantic equivalence,<->*)
Theorem repeat_until_slides : forall b S,
  Sequiv
    (REPEAT S UNTIL b)
    (S ; WHILE ~b DO S ).
Proof.
  split.
  apply proof_slides.
  apply proof_assignment.
Qed.
```

Appendix C: Coq definitions, lemmas and theorems

```
(*-------------------Framework_common-----------------------------*)
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.

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.

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

Definition t_empty {A : Type} (v : A) : total_map A :=
  (fun _ => v).

 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'.

Definition State := total_map Z.
Definition empty_State := (_ !-> 0).

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

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

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

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

(*--------------------------FrameworkNS---------------------------*)
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' ).

Example Example1_5 :
  (A[[x+1]] (x!->3)) = 4.

Theorem stm_eq :
  forall S s s' s'',
  << S, s >>-->s' ->
  s' = s'' ->
  << S, s >>-->s''.

Example Example2_1 :
  << ( z::= x ; x::= y ) ; y::= z , y!->7, x!->5 >>
```

```
 -->(y !-> 5, x !-> 7, z !-> 5, y !-> 7, x !-> 5).

Theorem Example2_2 :
  << y ::= 1 ; WHILE ~(x=1) DO ( y ::= y * x ; x ::= x-1 ),
  x !-> 3 >>-->(x !-> 1, y !-> 6, x !-> 2, y !-> 3, y !-> 1, x !-> 3).

Theorem Exercise2_3 :
  << z ::= 0; WHILE y <= x DO (z ::= z + 1 ; x ::= x - y),
  y !-> 5, x !-> 17 >>
 -->(x !-> 2, z !-> 3, x !-> 7, z !-> 2, x !-> 12,
       z !-> 1, z !-> 0, y !-> 5, x !-> 17).

(*--------------------Semantic_Equivalence----------------------*)

Definition Aequiv (a1 a2 : Aexp) : Prop :=
  forall (st : State),
    Aeval st a1 = Aeval st a2.

Definition Bequiv (b1 b2 : Bexp) : Prop :=
  forall (st : State),
    Beval st b1 = Beval st b2.

Definition Sequiv (S1 S2 : Stm) : Prop :=
  forall (st st' : State),
   << S1 , st >>--> st' <-> << S2 , st >>--> st'.

Theorem Lemma2_5 : forall b S,
  Sequiv
    ( WHILE b DO S )
    ( IF_ b THEN (S ; WHILE b DO S) ELSE SKIP ).

Theorem Exercise2_6: forall S1 S2 S3,
  Sequiv ( (S1 ; S2) ; S3) ( S1 ; (S2 ; S3)).

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

Inductive Seval : State -> Stm -> State -> Prop :=
  | ass_ns  : forall st a1 n x,
      Aeval st a1 = n ->
      Seval st (x ::= a1) (t_update st x n)
  | skip_ns : forall st,
      Seval st SKIP st
  | comp_ns : forall s1 s2 st st' st'',
      Seval st s1 st' ->
      Seval st' s2 st'' ->
      Seval st (s1 ; s2) st''
  | if_tt_ns : forall st st' b1 s1 s2,
      Beval st b1 = true ->
      Seval st s1 st' ->
      Seval st (IF_ b1 THEN s1 ELSE s2) st'
  | if_ff_ns : forall st st' b1 s1 s2,
      Beval st b1 = false ->
      Seval st s2 st' ->
      Seval st (IF_ b1 THEN s1 ELSE s2) st'
  | while_tt_ns : forall st st' st'' b1 s1,
      Beval st b1 = true ->
      Seval st s1 st' ->
      Seval st' (WHILE b1 DO s1) st'' ->
      Seval st (WHILE b1 DO s1) st''
  | while_ff_ns : forall b1 st s1,
      Beval st b1 = false ->
      Seval st (WHILE b1 DO s1) st
  | repeat_until_tt_ns : forall st st' b1 s1,
      Seval st s1 st' ->
      Beval st' b1 = true ->
      Seval st (repeat s1 b1) st'
  | repeat_until_ff_ns : forall st st' st'' b1 s1,
      Seval st s1 st' ->
      Beval st' b1 = false ->
      Seval st' (repeat s1 b1) st'' ->
      Seval st (repeat s1 b1) st''.

Lemma Lem_Assignment:
  forall b S s s' s'',
    << WHILE ~b DO S , s'' >>-->s'
  ->
    << S , s >>-->s''
  ->
    << REPEAT S UNTIL b , s >>-->s'.

Theorem repeat_until_slides : forall b S,
  Sequiv
    ( REPEAT S UNTIL b )
    ( S ; WHILE ~b DO S ).

(*--------------------------Determinism---------------------------*)
Theorem Seval_deterministic: forall S st st1 st2,
     << S, st >>--> st1  ->
     << S, st >>--> st2 ->
     st1 = st2.

(*--------------------------FrameworkAS---------------------------*)
Definition Assertion := State -> Prop.

Definition hoare_triple
  (P : Assertion) (S : Stm) (Q : Assertion) : Prop :=
  forall st st',
     << S , st >>--> st'  ->
     P st  ->
     Q st'.

Theorem hoare_post_true : forall (P Q : Assertion) c,
  (forall st, Q st) ->
  {{P}} c {{Q}}.

Theorem hoare_pre_false : forall (P Q : Assertion) c,
  (forall st, ~ (P st)) ->
  {{P}} c {{Q}}.

Definition assert_implies (P Q : Assertion) : Prop :=
  forall st, P st -> Q st.

Definition assn_sub x a P : Assertion :=
  fun (st : State) => P (x !-> Aeval st a , st).

Definition bassn b : Assertion :=
  fun st => (Beval st b = true).

Definition bassn2 b : Assertion :=
  fun st => ~(Beval st b = true).

Definition and_sub (P1 P2 : Assertion) : Assertion :=
  fun (st : State) => P1 st /\ P2 st.

Definition or_sub (P1 P2 : Assertion) : Assertion :=
  fun (st : State) => P1 st \/ P2 st.

Theorem assp : forall Q x a,
  {{Q [x |-> a]}} x ::= a {{Q}}.

Theorem consp_pre : forall (P P' Q : Assertion) S,
  {{P'}} S {{Q}} ->
  P =>> P' ->
  {{P}} S {{Q}}.

Theorem consp_post : forall (P Q Q' : Assertion) c,
  {{P}} c {{Q'}} ->
  Q' =>> Q ->
  {{P}} c {{Q}}.

Theorem consp : forall (P P' Q Q' : Assertion) S,
  {{P'}} S {{Q'}} ->
  P =>> P' ->
  Q' =>> Q ->
  {{P}} S {{Q}}.

Theorem skipp : forall P,
     {{P}} SKIP {{P}}.

Theorem compp : forall P Q R S1 S2,
     {{Q}} S2 {{R}} ->
     {{P}} S1 {{Q}} ->
     {{P}} S1; S2 {{R}}.

Lemma Bexp_eval_true : forall b st,
  Beval st b = true -> (bassn b) st.

Lemma Bexp_eval_false : forall b st,
  Beval st b = false -> ~(bassn b) st.

Theorem ifp : forall P Q b S1 S2,
  {{B[[b]] /\p P}} S1 {{Q}} ->
  {{~B[[b]] /\p P}} S2 {{Q}} ->
  {{P}} IF_ b THEN S1 ELSE S2 {{Q}}.

Theorem whilep : forall P b c,
  {{B[[b]] /\p P}} c {{P}} ->
  {{P}} WHILE b DO c {{~B[[b]] /\p P}}.

Theorem Example6_8a :
  {{ B[[BTrue]] }} WHILE BTrue DO SKIP {{ B[[BTrue]] }}.

Theorem Example6_8b :
  {{ B[[BTrue]] }} WHILE BTrue DO SKIP {{ ~B[[BTrue]] }}.

Fixpoint real_fact (n : nat) : nat :=
  match n with
  | O => 1
  | S n' => n * (real_fact n')
  end.

Theorem Example6_9:
  forall (m:nat),
  {{ fun st => st x = m }}
  y ::= 1; WHILE ~(x = 1)
  DO (y ::= y * x; x ::= x - 1)
  {{ fun st => st y = (real_fact m) /\ m > 0}}.

(*-----------------------Annotated_Programs-----------------------*)
Inductive AStm : Type :=
  | Aass : string -> Aexp ->  Assertion -> AStm
  | Askip :  Assertion -> AStm
  | Acomp : AStm -> AStm -> AStm
  | Aif : Bexp ->  Assertion -> AStm ->  Assertion -> AStm
          -> Assertion-> AStm
  | Awhile : Bexp -> Assertion -> AStm -> Assertion -> AStm
  | Apre : Assertion -> AStm -> AStm
  | Apost : AStm -> Assertion -> AStm.

Inductive Annotated : Type :=
  | annotated : Assertion -> AStm -> Annotated.
```

```
Fixpoint extract (a : AStm) : Stm :=
  match a with
  | Aass x y _        => x ::= y
  | Askip _           => SKIP
  | Acomp a1 a2       => (extract a1 ; extract a2)
  | Aif b _ a1 _ a2 _ => IF_ b THEN extract a1 ELSE extract a2
  | Awhile b _ a _    => WHILE b DO extract a
  | Apre _ a          => extract a
  | Apost a _         => extract a
  end.

Definition extract_ann (ann : Annotated) : Stm :=
  match ann with
  | annotated P a => extract a
  end.

Fixpoint post (a : AStm) : Assertion :=
  match a with
  | Aass x y Q              => Q
  | Askip P                 => P
  | Acomp a1 a2             => post a2
  | Aif  _ _ a1 _ a2 Q      => Q
  | Awhile b Pbody c Ppost => Ppost
  | Apre _ a                => post a
  | Apost c Q               => Q
  end.

Definition pre_ann (ann : Annotated) : Assertion :=
  match ann with
  | annotated P a => P
  end.

Definition post_ann (ann : Annotated) : Assertion :=
  match ann with
  | annotated P a => post a
  end.

Definition ann_correct (ann : Annotated) :=
  {{pre_ann ann}} (extract_ann ann) {{post_ann ann}}.

Fixpoint verification_cond (P : Assertion) (a : AStm) : Prop :=
  match a with
  | Aass x y Q =>
      (P =>> Q [x |-> y])
  | Askip Q =>
      (P =>> Q)
  | Acomp a1 a2 =>
      verification_cond P a1
     /\ verification_cond (post a1) a2
  | Aif b P1 a1 P2 a2 Q =>
      ((fun st => bassn b st /\ P st) =>> P1)
     /\ ((fun st => (bassn2 b st) /\ P st) =>> P2)
     /\ (post a1 =>> Q) /\ (post a2 =>> Q)
     /\ verification_cond P1 a1
     /\ verification_cond P2 a2
  | Awhile b Pbody a Ppost =>
      (P =>> post a)
     /\ ((fun st => bassn b st /\ post a st ) =>> Pbody)
     /\ ((fun st => (bassn2 b st) /\ post a st) =>> Ppost)
     /\ verification_cond Pbody a
  | Apre P' a =>
      (P =>> P') /\ verification_cond P' a
  | Apost a Q =>
      verification_cond P a /\ (post a =>> Q)
  end.

Lemma Bev : forall (b:Bexp) (st:State),
  (~B[[b]] st) = (~ (B[[b]]) st).
```

```
Theorem verification_correct : forall a P,
  verification_cond P a -> {{P}} (extract a) {{post a}}.

Definition verification_cond_ann (ann : Annotated) : Prop :=
  match ann with
  | annotated P a => verification_cond P a
  end.

Lemma verification_correct_ann : forall ann,
  verification_cond_ann ann -> ann_correct ann.

Example annotatedprogram : Annotated := (
  {{ B[[BTrue]] }}
  WHILE BTrue
  DO
    {{B[[BTrue]] /\p B[[BTrue]]}} =>>
    {{B[[BTrue]]}}
    SKIP
    {{ B[[BTrue]] }}
  {{ ~B[[BTrue]] /\p B[[BTrue]]}} =>>
  {{ B[[BFalse]] }}
).

Theorem annotatedprogram_correct :
  ann_correct (annotatedprogram).

Example my_fact (m:nat) : Annotated := (
    {{ fun st => st x = m }} =>>
    {{ fun st => st x > 0 -> real_fact (st x) =
        real_fact m /\ m >= st x }}
  y ::= ANum 1
    {{ fun st => st x > 0 -> st y * real_fact (st x) =
        real_fact m /\ m >= st x }};
  WHILE BNot (BEq (AId x) (ANum 1))
  DO   {{ fun st => (st x > 0 ->
        st y * real_fact (st x) = real_fact m /\ m >= st x)
       /\ st x <> 1 }} =>>
       {{ fun st => st x - 1 > 0 ->
        st y * st x * real_fact (st x - 1) =
        real_fact m /\ m >= (st x - 1) }}
     y ::= AMult (AId y) (AId x)
       {{ fun st => st x - 1 > 0 ->
        st y * real_fact (st x - 1) =
        real_fact m /\ m >= (st x - 1) }};
     x ::= AMinus (AId x) (ANum 1)
       {{ fun st => st x > 0 -> st y * real_fact (st x)
        = real_fact m /\ m >= st x }}
    {{ fun st => (st x > 0 -> st y * real_fact (st x) =
        real_fact m /\ m >= st x) /\ st x = 1 }} =>>
    {{ fun st => st y = real_fact m /\ m > 0}}
).

Theorem my_factcorrect : forall m,
  ann_correct (my_fact m).

(*---------------------------Soundness---------------------------*)
Inductive hoare_proof : Assertion -> Stm -> Assertion -> Type :=
  | ass_p : forall Q x a,
      hoare_proof (assn_sub x a Q) (x ::= a) Q
  | skip_p : forall P,
      hoare_proof P (SKIP) P
  | comp_p  : forall P S1 Q S2 R,
      hoare_proof P S1 Q ->
      hoare_proof Q S2 R ->
      hoare_proof P (S1;S2) R
  | if_p : forall P Q b S1 S2,
    hoare_proof (fun st => P st /\ bassn b st) S1 Q ->
    hoare_proof (fun st => P st /\ ~(bassn b st)) S2 Q ->
    hoare_proof P (IF_ b THEN S1 ELSE S2 ) Q
  | while_p : forall P b S,
    hoare_proof (fun st => P st /\ bassn b st) S P ->
    hoare_proof P (WHILE b DO S ) (fun st => P st /\ ~ (bassn b st))
  | cons_p  : forall (P Q P' Q' : Assertion) S,
    hoare_proof P' S Q' ->
    (forall s, P s -> P' s) ->
    (forall s, Q' s -> Q s) ->
    hoare_proof P S Q.

Theorem hoare_proof_sound : forall P S Q,
  hoare_proof P S Q -> {{P}} S {{Q}}.

(*-------------------------Completeness---------------------------*)
Definition wlp (S : Stm) (Q : Assertion) : Assertion :=
  fun s => forall s', << S , s >>-->s' -> Q s'.

Lemma wlp_property1:
  forall S Q,
  {{wlp S Q}} S {{Q}}.

Lemma wlp_property2:
  forall P S Q,
      {{P}} S {{Q}}
   ->
      forall s, P s -> wlp S Q s.

Theorem hoare_proof_complete: forall P S Q,
  {{P}} S {{Q}} -> hoare_proof P S Q.

(*-----------------------Theorems/Lemmas--------------------------*)
Theorem eqb_string_true_iff : forall x y : string,
    eqb_string x y = true <-> x = y.

Theorem eqb_string_false_iff : forall x y : string,
    eqb_string x y = false <-> x <> y.

Theorem t_update_neq : forall (m : total_map Z) x1 x2 v,
    x1 <> x2 ->
    (x1 !-> v , m) x2 = m x2.

Lemma eqb_string_sym : forall x y : string,
  eqb_string x y = eqb_string y x.

Lemma not_sym : forall (x y : string),
  x <> y <-> y <> x.

(*---------------------------Blocks-----------------------------*)
Definition DV := list string.

Definition empty_DV : DV := nil.

Fixpoint DV_add (x : string) (X : DV) : DV :=
  match X with
  | nil => x :: nil
  | x1 :: X1 =>
      match string_dec x x1 with
      | left _ => x1 :: X1
      | right _ => x1 :: DV_add x X1
      end
  end.

Fixpoint DV_mem (x : string) (X : DV) : bool :=
    match X with
    | nil => false
    | x1 :: X1 =>
        match string_dec x x1 with
        | left _ => true
        | right _ => DV_mem x X1
        end
    end.

Inductive Dv : Type :=
  | dec_v (x : string) (a : Aexp) (Dv1 : Dv)
  | empty_v.

Fixpoint DVeval (dv : Dv) : DV :=
  match dv with
  | dec_v x a Dv1 => DV_add x (DVeval Dv1)
  | empty_v => nil
  end.

Inductive Stm : Type :=
  | ass (x : string) (a : Aexp)
  | skip
  | comp (S1 S2 : Stm)
  | if_ (b : Bexp) (S1 S2 : Stm)
  | while (b : Bexp) (S : Stm)
  | block (Dv1 : Dv) (S : Stm).

Inductive SDecV : Dv -> State -> State -> Prop :=
  | none_ns : forall st,
      SDecV empty_v st st
  | var_ns : forall st st' a1 n (x : string) (Dv1 : Dv),
      Aeval st a1 = n ->
      SDecV Dv1 (t_update st x n) st' ->
      SDecV (dec_v x a1 Dv1) st st'
where "'<<' S ',' st '>>' -->D st'" := (SDecV Dv st st).

Definition var_update (s s' : State) (X : DV) (a : string) :=
  fun a => if DV_mem a X then (s a) else (s' a).

Inductive Seval : State -> Stm -> State -> Prop :=
  | ass_ns  : forall st a1 n x,
      Aeval st a1 = n ->
      Seval st (x ::= a1) (t_update st x n)
  | skip_ns : forall st,
      Seval st SKIP st
  | comp_ns : forall s1 s2 st st' st'',
      Seval st s1 st' ->
      Seval st' s2 st'' ->
      Seval st (s1 ; s2) st''
  | if_tt_ns : forall st st' b1 s1 s2,
      Beval st b1 = true ->
      Seval st s1 st' ->
      Seval st (IF_ b1 THEN s1 ELSE s2) st'
  | if_ff_ns : forall st st' b1 s1 s2,
      Beval st b1 = false ->
      Seval st s2 st' ->
      Seval st (IF_ b1 THEN s1 ELSE s2) st'
  | while_tt_ns : forall st st' st'' b1 s1,
      Beval st b1 = true ->
      Seval st s1 st' ->
      Seval st' (WHILE b1 DO s1) st'' ->
      Seval st (WHILE b1 DO s1) st''
  | while_ff_ns : forall b1 st s1,
      Beval st b1 = false ->
      Seval st (WHILE b1 DO s1) st
  | block_ns : forall Dv S st st' st'' (x:string),
      SDecV Dv st st' ->
      Seval st' S st'' ->
      Seval st (BEGIN Dv, S END)
               (var_update st st'' (DVeval Dv) x).

Definition example1 :=
  BEGIN var x := 1; var x := x + 1,
    x ::= 4
  END.
```

```
Theorem blocksproof1 :
  << example1 , x!->0 >>
 -->(x!->0).

Definition example2 :=
  BEGIN var y := 1,
    ( x ::= 1;
      (BEGIN var x := 2, y ::= x + 1 END);
      x ::= y + x )
  END.

Theorem blocksproof2 :
  << example2 , y!->0, x!->0 >>
 -->(x!->4, y!->0).

(*------------------------Dynamic_Scopes------------------------*)
Definition Pname := string.

Inductive Stm : Type :=
  | ass (x : string) (a : Aexp)
  | skip
  | comp (S1 S2 : Stm)
  | if_ (b : Bexp) (S1 S2 : Stm)
  | while (b : Bexp) (S : Stm)
  | block (Dv1 : Dv) (Dp1 : Dp) (S : Stm)
  | call (p : Pname)
with Dp : Type :=
  | dec_p (p : Pname) (S : Stm) (Dp1 : Dp)
  | empty_p.

Definition partial_map (A : Type) := total_map (option A).

Definition empty {A : Type} : partial_map A :=
  t_empty None.

Definition update {A : Type} (m : partial_map A)
           (x : string) (v : A) :=
  (x!->Some v , m).

Definition Env := partial_map Stm.

Fixpoint UpdP (dp : Dp) (envp : Env) : Env :=
  match dp with
  | dec_p p s dp => UpdP dp (update envp p s)
  | empty_p => envp
  end.

Inductive Seval : Env -> State -> option Stm -> State -> Prop :=
  | ass_ns  : forall envp st a1 n x,
      Aeval st a1 = n ->
      Seval envp st (Some (x ::= a1)) (t_update st x n)
  | skip_ns : forall envp st,
      Seval envp st (Some SKIP) st
  | comp_ns : forall envp s1 s2 st st' st'',
      Seval envp st (Some s1) st' ->
      Seval envp st' (Some s2) st'' ->
      Seval envp st (Some (s1 ; s2)) st''
  | if_tt_ns : forall envp st st' b1 s1 s2,
      Beval st b1 = true ->
      Seval envp st (Some s1) st' ->
      Seval envp st (Some (IF_ b1 THEN s1 ELSE s2)) st'
  | if_ff_ns : forall envp st st' b1 s1 s2,
      Beval st b1 = false ->
      Seval envp st (Some s2) st' ->
      Seval envp st (Some (IF_ b1 THEN s1 ELSE s2)) st'
  | while_tt_ns : forall envp st st' st'' b1 s1,
      Beval st b1 = true ->
      Seval envp st (Some s1) st' ->
      Seval envp st' (Some (WHILE b1 DO s1)) st'' ->
      Seval envp st (Some (WHILE b1 DO s1)) st''
  | while_ff_ns : forall envp b1 st s1,
      Beval st b1 = false ->
      Seval envp st (Some (WHILE b1 DO s1)) st
  | block_ns : forall envp (dv:Dv) (dp:Dp) S st st' st'' x,
      SDecV dv st st' ->
      Seval (UpdP dp envp) st' (Some S) st'' ->
      Seval envp st (Some (BEGIN dv, dp, S END))
                    (var_update st st'' (DVeval dv) x)
  | call_rec_ns : forall envp st st' p,
      Seval envp st (envp p) st' ->
      Seval envp st (Some (CALL p)) st'.
Notation "envp |- << s , st >>-->st'" := (Seval envp st s st')
                                 (at level 40).

Theorem stm_eq_env :
  forall envp S s s' s'',
  envp |- << S, s >>-->s' ->
  s' = s'' ->
  envp |- << S, s >>-->s''.

Definition procprogram :=
  BEGIN var x := 0,
    (proc p is ( x ::= x * 2 );
    proc q is ( CALL p )),
    BEGIN var x := 5,
      proc p is ( x ::= x + 1 ),
      ( (CALL q) ; y ::= x )
    END
  END.

Theorem dynamic_proof :
  empty |- << Some procprogram, x!->0 >>
 -->
  (y!->6, x!->0).

(*---------------------------Mixed_Scopes-------------------------*)
Inductive Envp : Type :=
  | static_env (f : Pname -> option (Stm * Envp)).

Definition extend (e : Envp) (p : Pname) (s : Stm) : Envp :=
  match e with
  | static_env f => static_env (fun p' : Pname
   => if eqb_string p p' then (Some (s,e)) else f p')
  end.

Definition empty_envp := static_env (fun _ => None).

Fixpoint UpdP2 (dp : Dp) (envp : Envp) : Envp :=
  match dp with
  | dec_p p s dp => UpdP2 dp (extend envp p s)
  | empty_p => envp
  end.

Definition fst2 (p: option (Stm * Envp)) :=
  match p with
  | Some (x, y) => x
  | None => SKIP
end.

Definition first (e : Envp) (p : Pname) :=
  match e with
  | static_env f => fst2 (f p)
  end.

Definition snd2 (p: option (Stm * Envp)) :=
  match p with
  | Some (x, y) => y
  | None => empty_envp
end.

Definition second (e : Envp) (p : Pname) :=
  match e with
  | static_env f => snd2 (f p)
  end.

Inductive Seval : Envp -> State -> Stm -> State -> Prop :=
  | ass_ns  : forall envp st a1 n x,
      Aeval st a1 = n ->
      Seval envp st (x ::= a1) (t_update st x n)
  | skip_ns : forall envp st,
      Seval envp st SKIP st
  | comp_ns : forall envp s1 s2 st st' st'',
      Seval envp st s1 st' ->
      Seval envp st' s2 st'' ->
      Seval envp st (s1 ; s2) st''
  | if_tt_ns : forall envp st st' b1 s1 s2,
      Beval st b1 = true ->
      Seval envp st s1 st' ->
      Seval envp st (IF_ b1 THEN s1 ELSE s2) st'
  | if_ff_ns : forall envp st st' b1 s1 s2,
      Beval st b1 = false ->
      Seval envp st s2 st' ->
      Seval envp st (IF_ b1 THEN s1 ELSE s2) st'
  | while_tt_ns : forall envp st st' st'' b1 s1,
      Beval st b1 = true ->
      Seval envp st s1 st' ->
      Seval envp st' (WHILE b1 DO s1) st'' ->
      Seval envp st (WHILE b1 DO s1) st''
  | while_ff_ns : forall envp b1 st s1,
      Beval st b1 = false ->
      Seval envp st (WHILE b1 DO s1) st
  | block_ns : forall envp (dv:Dv) (dp:Dp) S st st' st'' x,
      SDecV dv st st' ->
      Seval (UpdP2 dp envp) st' S st'' ->
      Seval envp st (BEGIN dv, dp, S END)
                    (var_update st st'' (DVeval dv) x)
  | call_ns : forall (envp:Envp) st st' p,
      Seval (second envp p) st (first envp p) st' ->
      Seval envp st (CALL p) st'
  | call_rec_ns : forall envp st st' p,
      Seval envp st (first envp p) st' ->
      Seval envp st (CALL p) st'.
Notation "envp |- << s , st >>-->st'" := (Seval envp st s st')
                                  (at level 40).

Theorem static_proof :
  empty_envp |- << procprogram, x!->0 >>
 -->
  (y!->10, x!->0).

Theorem dynamic_proof :
  empty_envp |- << procprogram, x!->0 >>
 -->
  (y!->6, x!->0).

(*-----------------------Non-determinism--------------------------*)
Inductive Stm : Type :=
  | ass (x : string) (a : Aexp)
  | skip
  | comp (s1 s2 : Stm)
  | if_ (b : Bexp) (s1 s2 : Stm)
  | while (b : Bexp) (s : Stm)
  | or (S1 S2 : Stm).

Inductive Seval : State -> Stm -> State -> Prop :=
  | ass_ns  : forall st a1 n x,
      Aeval st a1 = n ->
      Seval st (x ::= a1) (t_update st x n)
  | skip_ns : forall st,
      Seval st SKIP st
  | comp_ns : forall s1 s2 st st' st'',
      Seval st s1 st' ->
      Seval st' s2 st'' ->
      Seval st (s1 ; s2) st''
  | if_tt_ns : forall st st' b1 s1 s2,
      Beval st b1 = true ->
      Seval st s1 st' ->
      Seval st (IF_ b1 THEN s1 ELSE s2) st'
  | if_ff_ns : forall st st' b1 s1 s2,
      Beval st b1 = false ->
      Seval st s2 st' ->
      Seval st (IF_ b1 THEN s1 ELSE s2) st'
  | while_tt_ns : forall st st' st'' b1 s1,
      Beval st b1 = true ->
      Seval st s1 st' ->
      Seval st' (WHILE b1 DO s1) st'' ->
      Seval st (WHILE b1 DO s1) st''
  | while_ff_ns : forall b1 st s1,
      Beval st b1 = false ->
      Seval st (WHILE b1 DO s1) st
  | or_1_ns : forall st st' s1 s2,
      Seval st s1 st' ->
      Seval st (or s1 s2) st'
  | or_2_ns : forall st st' s1 s2,
      Seval st s2 st' ->
      Seval st (or s1 s2) st'.

Example nondet1 :
  << x ::= 1 OR (x ::= 2 ; x ::= x + 2), x !-> 0 >>-->(x !-> 1).

Example nondet2 :
  << x ::= 1 OR (x ::= 2 ; x ::= x + 2), x !-> 0 >>-->(x !-> 4).

Example nondet3 :
 << (WHILE true DO SKIP) OR (x ::= 2 ; x ::= x + 2), x !-> 0 >>
-->(x !-> 4).

Example nondet4 :
 << (WHILE true DO SKIP) OR (x ::= 2; x ::= x + 2), x !-> 0 >>
-->(x !-> 4).

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

Inductive Bstatus : Type :=
  | b
  | no_b.

Inductive Seval : State -> Stm -> State -> Bstatus -> Prop :=
  | ass_ns  : forall st a1 n x,
      Aeval st a1 = n ->
      Seval st (x ::= a1) (t_update st x n) no_b
  | skip_ns : forall st,
      Seval st SKIP st no_b
  | comp_b_ns : forall s1 s2 st st',
      Seval st s1 st' b ->
      Seval st (s1 ; s2) st' b
  | comp_no_b_ns : forall s1 s2 st st' st'' B,
      Seval st s1 st' no_b ->
      Seval st' s2 st'' B ->
      Seval st (s1 ; s2) st'' B
  | if_tt_ns : forall st st' b1 s1 s2 B,
      Beval st b1 = true ->
      Seval st s1 st' B ->
      Seval st (IF_ b1 THEN s1 ELSE s2) st' B
  | if_ff_ns : forall st st' b1 s1 s2 B,
      Beval st b1 = false ->
      Seval st s2 st' B ->
      Seval st (IF_ b1 THEN s1 ELSE s2) st' B
  | while_tt_b_ns : forall st st' b1 s1,
      Beval st b1 = true ->
      Seval st s1 st' b ->
      Seval st (WHILE b1 DO s1) st' no_b
  | while_tt_no_b_ns : forall st st' st'' b1 s1 B,
      Beval st b1 = true ->
      Seval st s1 st' no_b ->
      Seval st' (WHILE b1 DO s1) st'' B ->
      Seval st (WHILE b1 DO s1) st'' no_b
  | while_ff_ns : forall b1 st s1,
      Beval st b1 = false ->
      Seval st (WHILE b1 DO s1) st no_b
  | break_ns : forall st,
      Seval st (BREAK) st b
where "'<<' s ',' st '>>' '-->' '(' st' ',' br ')'" :=
      (Seval st s st' br).

Theorem break_ignore : forall S st st' B,
     << BREAK ; S, st >>-->( st', B ) ->
      st = st'.

Theorem while_continue : forall b S st st' B,
  << WHILE b DO S, st >>-->( st', B ) ->
  B = no_b.

Theorem while_stops_on_break : forall bo S st st',
  Beval st bo = true ->
  << S, st >>-->(st', b) ->
  << WHILE bo DO S, st >>-->(st', no_b).

Theorem stm_eq :
  forall S s s' s'' B,
  << S, s >>-->(s', B) ->
  s' = s'' ->
  << S, s >>-->(s'', B).

Definition program1 :=
  x ::= 5;
  y ::= 3;
  WHILE (x <= 7) DO
    x ::= x + 2;
    (IF_ ~(x <= 8) THEN BREAK ELSE SKIP);
    y ::= y*3.

Example proof1 :
  << program1, x!->0 >>-->( (x!->9, y!->9), no_b).

Definition program2 :=
  x ::= 5;
  BREAK;
  y ::= 3;
  WHILE (x <= 7) DO
    x ::= x + 2;
    (IF_ ~(x <= 8) THEN BREAK ELSE SKIP);
    y ::= y*3.

Example proof2 :
  << program2, x!->0 >>-->((x!->5, x!->0), b).
```

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

Inductive Cstatus : Type :=
  | c
  | no_c.

Inductive Seval : State -> Stm -> State -> Cstatus -> Prop :=
  | ass_ns  : forall st a1 n x,
      Aeval st a1 = n ->
      Seval st (x ::= a1) (t_update st x n) no_c
  | skip_ns : forall st,
      Seval st SKIP st no_c
  | comp_c_ns : forall s1 s2 st st',
      Seval st s1 st' c ->
      Seval st (s1 ; s2) st' c
  | comp_no_c_ns : forall s1 s2 st st' st'' C,
      Seval st s1 st' no_c ->
      Seval st' s2 st'' C ->
      Seval st (s1 ; s2) st'' C
  | if_tt_ns : forall st st' b1 s1 s2 C,
      Beval st b1 = true ->
      Seval st s1 st' C ->
      Seval st (IF_ b1 THEN s1 ELSE s2) st' C
  | if_ff_ns : forall st st' b1 s1 s2 C,
      Beval st b1 = false ->
      Seval st s2 st' C ->
      Seval st (IF_ b1 THEN s1 ELSE s2) st' C
  | while_tt_c_ns : forall st st' st'' b1 s1 C,
      Beval st b1 = true ->
      Seval st s1 st' c ->
      Seval st' (WHILE b1 DO s1) st'' C ->
      Seval st (WHILE b1 DO s1) st' no_c
  | while_tt_no_c_ns : forall st st' st'' b1 s1 C,
      Beval st b1 = true ->
      Seval st s1 st' no_c ->
      Seval st' (WHILE b1 DO s1) st'' C ->
      Seval st (WHILE b1 DO s1) st'' no_c
  | while_ff_ns : forall b1 st s1,
      Beval st b1 = false ->
      Seval st (WHILE b1 DO s1) st no_c
  | continue_ns : forall st,
      Seval st (CONTINUE) st c
where "'<<' s ',' st '>>' '-->' '(' st' ',' con ')'" :=
      (Seval st s st' con).

Theorem stm_eq :
  forall S s s' s'' C,
  << S, s >>-->(s', C) ->
  s' = s'' ->
  << S, s >>-->(s'', C).

Definition program1 :=
  x ::= 5;
  y ::= 3;
  WHILE (x <= 7) DO
    x ::= x + 2;
    (IF_ ~(x <= 8) THEN CONTINUE ELSE SKIP);
    y ::= y*3.

Example proof1 :
  << program1, x!->0 >>-->( (x!->9, y!->9), no_c).
```

```
Definition program2 :=
  x ::= 5;
  CONTINUE;
  y ::= 3;
  WHILE (x <= 7) DO
    x ::= x + 2;
    (IF_ ~(x <= 8) THEN CONTINUE ELSE SKIP);
    y ::= y*3.

Example proof2 :
  << program2, x!->0 >>-->( (x!->5, x!->0), c).
```