Radboud University Nijmegen

Faculty of Science

Compiling While++ into Logically Constrained Term Rewriting Systems

Thesis BSc Computing Science

Author:
Mikhail Ushakov
s1080976

Supervisor:
Dr. Cynthia Kop

Second reader:
Dr. Engelbert Hubbers

July 2024

Contents

Chapter 1

Introduction

Imperative programming is a common software development paradigm where a program consists of a specific sequence of statements that modify the program's state. It is among the oldest and most widely used paradigms in modern software development. According to the IEEE Spectrum[1], eight of the top ten programming languages in the industry support imperative programming. Considering the legacy and ubiquitous nature of imperative programming, there is a high demand for approaches that can analyze and optimize imperative code effectively.

One mathematical framework that is effective for code analysis is Term Rewriting. In addition to being a branch of theoretical computer science, Term Rewriting is a computational paradigm that combines elements of logic, algebra, and functional programming. This computational paradigm is used in various fields of computer science, such as software engineering, programming languages, program verification, and automatic theorem proving.

In this thesis project, we considered a special variation of Term Rewriting known as Logically Constrained Term Rewriting. There are two key differences between these two types of rewriting, especially in the construction of Term Rewriting Systems (TRSs). In classic term rewriting, the terms are untyped and there are no restrictions for the rules that can be applied to the terms. The only natural restriction is the form of the term that can be matched by the left-hand side of the TRS rule. In Logically Constrained Term Rewriting all the terms and variables are typed: each functional symbol and variable is equipped with a sort declaration. Also, for each rule of the Logically Constrained TRS (LCTRS), there exists a logical constraint: a logical expression that can involve variables from the left-hand side of the rule. The rule can be applied to the term only if the constraint included in the rule is satisfied.

To better understand the difference between TRSs and LCTRSs we consider Example 1. We have two term rewriting systems that describe the operation of searching the maximum element in a list implemented in a classic TRS and a LCTRS.

Example 1.

(a) **Classic TRS**

We represent numbers using the Peano axioms:

– Term 0 represents the natural number 0.
– The rest of the numbers are defined inductively: suppose a is a natural number, then the natural number $a + 1$ will be represented as $s(a)$.

So, the number 1 will be represented with term $s(0)$, number 2 with term $s(s(0))$, number 3 with term $s(s(s(0)))$ and so on.

The following set of rules defines the operation of finding the maximum between two natural numbers represented in the Peano notation:

$r1 : max2(0, a) \to a$

$r2 : max2(a, 0) \to a$

$r3 : max2(s(a), s(b)) \to s(max2(a, b))$

The list in this TRS is represented in the following way:

– The empty list is represented with the term nil.
– To show that the list contains more than one number we use the *cons*-construction:
  Suppose l is a list, and we want to add one element a to this list. It can be represented in the following way: $cons(a, l)$.

For example, the list $[2, 3, 0]$ in TRS syntax is represented as:

$$cons(s(s(0)), cons(s(s(s(0))), cons(0, nil)))$$

The following set of rules describes the procedure for finding the maximum element in the list:

$r4 : maxl(nil) \to err$

$r5 : maxl(cons(a, l)) \to maxl(pair(a, l))$

$r6 : maxl(pair(a, nil)) \to a$

$r7 : maxl(pair(a, cons(b, l))) \to maxl(pair(max2(a, b), l))$

Reduction example:

The example shows the procedure of searching the maximum element in the list of the natural numbers: $[2, 3, 0]$.

$maxl(cons(s(s(0)), cons(s(s(s(0))), cons(0, nil))))$

$\xrightarrow{r5} maxl(pair(s(s(0)), cons(s(s(s(0))), cons(0, nil))))$

$\xrightarrow{r7} maxl(pair(max2(s(s(0)), s(s(s(0)))), cons(0, nil))$

$\xrightarrow{r3} maxl(pair(s(max2(s(0), s(s(0)))), cons(0, nil))$

$\xrightarrow{r3} maxl(pair(s(s(max2(0, s(0))))), cons(0, nil))$

$\xrightarrow{r1} maxl(s(s(s(0))), cons(0, nil))$

$\xrightarrow{r7} maxl(pair(max2(s(s(s(0))), 0)), nil)$

$\xrightarrow{r2} maxl(pair(s(s(s(0))), nil))$

$\xrightarrow{r6} s(s(s(0)))$

(b) **LCTRS**

Because LCTRS is a variation of the Many-Sorted TRS, we can use integer values as terms. Here we again represent lists using *cons*-constructions. In addition, for every term, we need to define the type or function signature.

$nil : list$

$cons : [int \times list] \Rightarrow list$

$p : [int \times list] \Rightarrow pair$

$max : [list] \Rightarrow int$

$maxl : [pair] \Rightarrow int$

$r1 : max(nil) \to -1$

$r2 : max(cons(a, l)) \to maxl(p(a, l))$

$r3 : maxl(p(a, cons(b, l))) \to maxl(p(a, l))\ [a \geq b]$

$r4 : maxl(p(a, cons(b, l))) \to maxl(p(b, l))\ [a < b]$

$r5 : maxl(p(a, nil)) \to a$

Reduction example:

The example shows the procedure of searching the maximum element in the list of integers: $[2, 3, 0]$.

$max(cons(2, cons(3, cons(0, nil))))$

$\xrightarrow{r2} maxl(p(2, cons(3, cons(0, nil))))$

$\xrightarrow{r4} maxl(p(3, cons(0, nil)))$ (because $2 < 3$)

$\xrightarrow{r3} maxl(p(3, nil))$ (because $3 \geq 0$)

$\xrightarrow{r5} 3$

As we can see from Example 1: a TRS provides quite a limited functionality to describe such a basic operation as searching for the maximum element in the list. We used Peano axioms to define our representation of natural numbers. Additionally, we defined the operation of deriving the maximum of two natural numbers represented in Peano form. Regarding the Logically Constrained TRS, we do not need to "invent" extra features to support operations with natural numbers. Because LCTRS essentially is an extension of the Many-Sorted TRS, we natively can use values of different types in the rules of the Term Rewriting System. It allows us to represent numbers in a much easier way. Also, for the maximum operation, we do not need to invent extra functions that will compute a maximum of two natural numbers in a specific way. LCTRSs support logical constraints in rewriting rules, so we can add extra rules ($r3$ and $r4$) that will select the maximum number out of two values inside the term.

The LCTRS is not limited by the native representation of the numbers, boolean constants, and so on, it also supports the operations on the terms of a certain type. For example, we can define the native operations for terms of integer type. This would allow us to have the following LCTRS:

Example 2.

- **LCTRS**

$fac : [int] \Rightarrow int$

$r1 : fac(1) \to 1$
$r2 : fac(n) \to n \cdot fac(n-1)\ [n > 1]$

- **Reduction example:**

$fac(3) \xrightarrow{r2} fac(3-1) \xrightarrow{\texttt{calc}} 3 \cdot fac(2)$
$\xrightarrow{r2} 3 \cdot (2 \cdot fac(2-1)) \xrightarrow{\texttt{calc}} 3 \cdot (2 \cdot fac(1))$
$\xrightarrow{r1} 3 \cdot (2 \cdot 1) \xrightarrow{\texttt{calc}} 3 \cdot 2$
$\xrightarrow{\texttt{calc}} 6$

One of the challenges related to code analysis using LCTRSs is the development of a systematic approach for the translation of imperative code into an LCTRS. This problem is further compounded by the fact that imperative programming languages may have different features. In turn, this makes the creation of a uniform set of translation rules a massive challenge. Consequently, the goal of this paper is to create a simplified language that supports all the main features of imperative programming, which we will then use to establish a general set of translation rules for translating imperative code into an LCTRS.

To achieve our goals, we created While++: a simple imperative programming language that is derived from the While programming language[2]. The syntax of the While++ language is limited to the following constructions: variable declarations, variable assignments, if-then-else constructions, while loops, simple I/O constructions, and the composition of statements. This programming language also supports only three types: integer, boolean, and string.

This programming language was used for the design and formalization of the translation process of imperative programs into Term Rewriting Systems. The correctness of the translation was proven by showing the equivalence of the natural semantics derivation trees and the rewriting sequences.

The formalized translation process of the While++ programs into LCTRSs was implemented as a part of this thesis project in the library WPP2TRS, written in Java. The output Term Rewriting Systems were made compatible with the Cora (COnstrained Rewriting Analyzer) open-source tool [3].

Paper setup. This paper is structured as follows: in Chapter 2 there are all the necessary preliminaries that can help the reader get familiar with the research topic. In Chapter 3, there is a discussion about the related work that was done in this area: what are the differences with this thesis project, and what topics were not yet widely covered in the area of the translation of imperative programs into Term Rewriting Systems. In Chapter 4, we discuss the design of While++, including the syntax and the natural semantics of this programming language. Chapter 5 gives information about the translation process, including the notion of the translation function and the particular translation rules that allow turning While++ programs into LCTRSs. In Chapter 6, we give lemmas and theorems that give all logical arguments regarding the validity of the designed translation process. Chapter 7 covers the implementation part: it describes all the important design choices made during the development of the WPP2TRS library. Chapter 8 gives the thesis project conclusion. Consequently, in Chapter 9, we discuss

the future work that can be done in this area on the base of this project. And finally, in the appendix section, we give all the proofs that were omitted in the previous sections.

Chapter 2

Preliminaries

2.1 Many-Sorted TRS

A Many-Sorted TRS is an abstract reduction system where rewriting rules apply to the terms, and each term has a sort restriction. Terms are built from function symbols and variables. The function symbols consist of a name and zero or more arguments, where each argument is a TRS term. Function symbols with zero arguments are usually called *constants*.

A Term Rewriting System can be represented as a pair $(\Sigma \cup \mathcal{V}, R)$, where $\Sigma \cup \mathcal{V}$ is an alphabet of the TRS, and R is a set of reduction (rewrite) rules. The set of terms of the TRS is denoted as $Terms(\Sigma \cup \mathcal{V})$.

2.1.1 Alphabet of the Many-Sorted TRS

For any many-sorted TRS there exists a set of sorts $\mathcal{S}$.

> **Example 3.** $\mathcal{S} = \{int, bool, string\}$

The alphabet $\Sigma \cup \mathcal{V}$ of a Many-Sorted TRS consists of the following:

- A countably infinite set of pairs $\mathcal{V} = \{x_1 : \iota_1, x_2 : \iota_2 \dots\}$, where $x_1, x_2, \dots$ are variables, and $\{\iota_1, \iota_2, \dots\} \subseteq \mathcal{S}$ are sorts from the set of available sorts $\mathcal{S}$.
- A non-empty set of function symbols Σ, where every function symbol f is equipped with a sort declaration of the form $[\iota_1 \times \cdots \times \iota_n] \Rightarrow \kappa$, where $\{\iota_1, \dots \iota_n\} \cup \{\kappa\} \subseteq \mathcal{S}$ are sorts from the set of available sorts $\mathcal{S}$.

There exists a notion of the variable environment $\Gamma \subseteq \mathcal{V}$, which is a set of pairs of variables and their sorts.

> **Example 4.** Consider the example of the alphabet of the Many-Sorted TRS:
>
> - Set of sorts: $\mathcal{S} = \{int, bool, pair\}$.
> - Alphabet: $\Sigma \cup \mathcal{V}$.

$\{x : int, b : bool\} \subseteq \mathcal{V}$

$$\Sigma = \begin{Bmatrix} intBoolPair : [int \times bool] \Rightarrow pair, \\ first : [pair] \Rightarrow int, \\ second : [pair] \Rightarrow bool \end{Bmatrix}$$

2.1.2 Terms of the Many-Sorted TRS

The set $Terms(\Sigma \cup \mathcal{V})$ consists of all possible terms of the Many-Sorted TRS. In general, a term is an expression s that is a combination of the function symbols (and constants) from the set Σ, and variables from the set $\mathcal{V}$. This term s is built such that $\Gamma \vdash s : \iota$ can be derived for some environment Γ, and sort ι, using the inference rules:

$$\frac{}{\Gamma \cup \{x : \iota\} \vdash x : \iota}$$

$$\frac{\Gamma \vdash s_1 : \iota_1 \ \ldots \ \Gamma \vdash s_n : \iota_n \ f : [\iota_1 \times \cdots \times \iota_n] \Rightarrow \kappa \in \Sigma}{\Gamma \vdash f(s_1, \ldots, s_n) : \kappa}$$

Additionally for the term s, we define a set $Var(s)$. This set consists of all variables occurring in the term s. We call a term s ground if $Var(s) = \emptyset$.

Example 5. Consider the Many-Sorted TRS with the following alphabet:

- Set of sorts: $\mathcal{S} = \{int, bool, pair\}$.
- Alphabet: $\Sigma \cup \mathcal{V}$.

  $$\Sigma = \begin{Bmatrix} intBoolPair : [int \times bool] \Rightarrow pair, \\ first : [pair] \Rightarrow int, \\ second : [pair] \Rightarrow bool \end{Bmatrix}$$

We also consider the environment $\Gamma = \{x : int, y : bool\}$.

Using the alphabet we can construct the following term:

$$intBoolPair(x, y) : pair$$

We also can show how this term can be derived for the environment Γ, and sort $pair$:

$$\frac{\dfrac{}{\Gamma \vdash x : int} \quad \dfrac{}{\Gamma \vdash y : bool} \quad intBoolPair : [int \times bool] \Rightarrow pair \in \Sigma}{\Gamma \vdash intBoolPair(x, y) : pair}$$

2.1.3 Substitution

A substitution σ is a sort-preserving mapping: $\mathcal{V} \to \mathcal{T}erms(\Sigma \cup \mathcal{V})$ (mapping from variables to terms).

There exists an operation of applying the substitution to the term. It can be defined inductively:

- For variables: $x\sigma = \sigma(x)$

- For function symbols: $f(t_1, \ldots, t_n)\sigma = f(t_1\sigma, \ldots, t_n\sigma)$

The result of applying the substitution $\gamma = \{x_1 : \iota_1 \mapsto t_1, x_2 : \iota_2 \mapsto t_2, \ldots\}$ to the term t is the term t with all occurrences of any x_i replaced by the corresponding t_i (described in the substitution).

Example 6. Consider the alphabet:

$$\Sigma = \begin{Bmatrix} f : [int \times bool] \Rightarrow int, \\ a : [\,] \Rightarrow int, \\ b : [\,] \Rightarrow bool \end{Bmatrix}.$$

Also: $\{x : int, y : bool\} \subseteq \mathcal{V}$

Consider the substitution $\sigma : \sigma(x) = a, \sigma(y) = b$.

Application of the substitution σ to the term $f(x, y)$:

$$f(x, y)\sigma = f(x\sigma, y\sigma) = f(\sigma(x), \sigma(y)) = f(a, b)$$

2.1.4 Contexts

The context is a special term C with zero or more special variables $\Box_1, \ldots, \Box_n$ each occurring once.

If this context term C can be derived for the environment $\Gamma \cup \{\Box_1 : \iota_1, \ldots, \Box_n : \iota_n\}$ and the sort κ, and if it holds that $\forall i \in \{1, n\} : \Gamma \vdash s_i : \iota_1$, then we define term $C[s_1, \ldots, s_n]$ as a term C with each $\Box_i$ replaced by the corresponding s_i.

2.1.5 Reduction rules of the Many-Sorted TRS

A reduction rule is a pair (l, r), where $l, r \in \mathcal{T}erms(\Sigma \cup \mathcal{V})$ (notation: $l \to r$). The term l is called left-hand side and r is called right-hand side of the rule. We have the following restrictions for the terms of the reduction rule:

- Term l cannot be a variable.
- Terms l and r must have the same sort.
- $Var(r) \subseteq Var(l)$.

There exists a notion of the rewrite relation: $\to_R$ is defined as the smallest relation $\to_R \subseteq \mathcal{T}erms(\Sigma \cup \mathcal{V}) \times \mathcal{T}erms(\Sigma \cup \mathcal{V})$ that satisfies the following:

- $l\sigma \to_R r\sigma$ for every rewriting rule $l \to r \in R$, and every substitution σ.
- If $\exists i : (t_i \to_R u_i \wedge \forall j \neq i(t_j = u_j))$, then: $f(t_1, \ldots, t_n) \to_R f(u_1, \ldots, u_n)$.

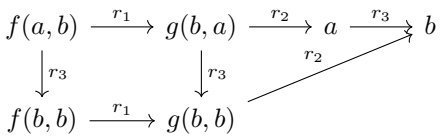
Example 7. Consider the Many-Sorted TRS with the following alphabet and set of rewriting rules:

- Alphabet: $\Sigma = \begin{cases} f : [int \times int] \Rightarrow int, \\ g : [int \times int] \Rightarrow int, \\ a : [\,] \Rightarrow int, \\ b : [\,] \Rightarrow int \end{cases}$.

  $x : int, y : int \in \mathcal{V}$

- Reduction rules:
  $r_1 : f(x, y) \to g(y, x)$
  $r_2 : g(x, y) \to y$
  $r_3 : a \to b$

  Application of the reduction rules to the term $f(a, b)$:



In this example we showed not a usual reduction chain, but the reduction graph. This way we demonstrate that with the given initial term and the set of rewriting rules there are many alternative reduction options, which in this case all lead to the same final term (usually called "normal form").

2.2 Logically Constrained TRS

A Logically Constrained TRS is a Many-Sorted Term Rewriting System, where certain terms carry an implicit meaning, and each rule is equipped with a logical constraint.

2.2.1 Alphabet of a Logically Constrained TRS

The alphabet $\Sigma \cup \mathcal{V}$ of a Logically Constrained TRS consists of the following:

- A Countably infinite set of pairs $\mathcal{V}$, where each pair $x : \iota$ is a variable equipped with a sort.
- A Non-empty set of function symbols $\Sigma = \Sigma_{terms} \cup \Sigma_{theory}$.

  This division of the function symbol set also allows us to specify the special subset of sorts: $\mathcal{S}_{theory} \subseteq \mathcal{S}$. This subset consists of all the sorts occurring in the function symbol signatures in the set Σ_{theory}.

  Also, there exists a special function $\mathcal{I}$ that assigns a set every sort $\iota \in \mathcal{S}_{theory}$.

  Example 8.
  $\mathcal{S} = \{bool\}$
  $\mathcal{I} = \{bool \mapsto \mathbb{B}\}$
  The set $\mathbb{B} = \{\top, \bot\}$ is a set of boolean values.

  Additionally, there exists a special function $\mathcal{J}$ that maps each $f : [\iota_1 \times \cdots \times \iota_n] \Rightarrow \kappa \in \Sigma_{theory}$ to a function $\mathcal{J}_f : \mathcal{I}_{\iota_1} \times \cdots \times \mathcal{I}_{\iota_n} \to \mathcal{I}_\kappa$.

Example 9.
$\mathcal{S} = \{int, bool\}$
$\mathcal{I} = \{int \mapsto \mathbb{Z}, bool \mapsto \mathbb{B}\}$
$geq : [int \times int] \Rightarrow bool \in \Sigma_{theory}$
$\mathcal{J} = \{(geq : [int \times int] \Rightarrow bool) \mapsto (\geq\colon \mathbb{Z} \times \mathbb{Z} \to \mathbb{B})\}$

For each sort $\iota \in \mathcal{S}_{theory}$ we fix a special subset $\mathcal{V}al_\iota \subseteq \Sigma_{theory}$. These subsets contain values associated with each sort available for the LCTRS.

The set of all values can be described as:

$$\mathcal{V}al = \bigcup_{\iota \in \mathcal{S}_{theory}} \mathcal{V}al_\iota$$

For each element $(a : [\,] \Rightarrow \iota) \in \mathcal{V}al_\iota$, the function $\mathcal{J}$ gives one-to-one mapping from $\mathcal{V}al_\iota$ to $\mathcal{I}_\iota$.

2.2.2 Terms of the Logically Constrained TRS

The set $Terms(\Sigma \cup \mathcal{V})$, where $\Sigma = \Sigma_{terms} \cup \Sigma_{theory}$ consists of all possible terms in the Logically Constrained TRS. Terms in LCTRSs inherit all features from Many-Sorted TRS. The exact definitions can be found in the corresponding terms' description (subsection 2.1.2). However, there are some additional features that exists specifically for the terms of the LCTRS.

Because in LCTRSs we introduced the distinction between the different types of function symbols (Σ_{terms} and Σ_{theory}), we also have this kind of distinction with the terms:

- We call a term in $Terms(\Sigma_{theory} \cup \mathcal{V})$ a logical term. These are the terms that are used to define a function or constraint in the model.
- And we call a term in $Terms(\Sigma_{terms} \cup \mathcal{V})$ a proper term. These are the terms that are used to describe objects that we want to rewrite.

As we discussed in the Alphabet subsection 2.2.1, for every function symbol from the theory set Σ_{theory}, we define the interpretation mapping $\mathcal{J}$. This mapping can be extended to an interpretation on the ground terms in $\mathcal{T}erms(\Sigma \cup \mathcal{V})$ in the following way:

$$[\![f(s_1, \ldots, s_n)]\!]_{\mathcal{J}} = \mathcal{J}_f([\![s_1]\!]_{\mathcal{J}}, \ldots, [\![s_n]\!]_{\mathcal{J}})$$

Example 10.

$\mathcal{S} = \{bool\}$
$\mathcal{I} = \{bool \mapsto \mathbb{B}\}$
$\mathcal{V}al_{bool} = \{\texttt{true}, \texttt{false}\}$
$and : [bool \times bool] \Rightarrow bool \in \Sigma_{theory}$
$[\![\texttt{true}]\!]_{\mathcal{J}} = \top$
$[\![\texttt{false}]\!]_{\mathcal{J}} = \bot$
$\mathcal{J} = \{(and : [bool \times bool] \Rightarrow bool) \mapsto (\wedge : \mathbb{B} \times \mathbb{B} \to \mathbb{B})\}$

$[\![and(a, b)]\!]_{\mathcal{J}} = [\![a]\!]_{\mathcal{J}} \wedge [\![b]\!]_{\mathcal{J}}$
$[\![and(\texttt{true}, \texttt{false})]\!]_{\mathcal{J}} = [\![\texttt{true}]\!]_{\mathcal{J}} \wedge [\![\texttt{false}]\!]_{\mathcal{J}} = \top \wedge \bot = \bot$

Additionally in the Alphabet section we defined the set of values $\mathcal{V}al \subseteq \Sigma_{theory}$. We identify a value c with the logical term $c()$.

A logical ground term s has a value t if t is a value such that $[\![s]\!] = [\![t]\!]$. Because ground terms by definition do not include any variables, we always can "evaluate" the logical ground term s and obtain the value t using appropriate interpretations. Every ground logical term has a unique value.

Finally, we introduce the notion of the logical constraint: it is a logical term of some sort $bool$, such that $\mathcal{I}_{bool} = \mathbb{B}$. And usually $Val_{bool} = \{\texttt{true}, \texttt{false}\}$.

There exists a notion of the validity of the logical constraint:

- We say that the ground logical constraint s is valid if $[\![s]\!]_{\mathcal{J}} = \top$.
- We say that the non-ground logical constraint s is valid if $\forall\gamma$, where γ is a substitution that maps the variables in $Var(s)$ to a value: $s\gamma$ is valid.

2.2.3 Reduction rules of the Logically Constrained TRS

A rule in the Logically Constrained TRS is a tuple $l \to r\ [\varphi]$, where l and r are terms and φ is a logical constraint. We have a restriction for the term l: it must have the form $f(l_1, \ldots, l_n)$, where $f \in \Sigma_{terms} \setminus \Sigma_{theory}$. Also, the terms l and r must be of the same sort. And for every $x : \iota \in Var(r) \setminus Var(l)$, it holds, that: $\iota \in \mathcal{S}_{theory}$.

As we can see, the definition of the reduction rule for LCTRSs is quite different from the reduction rule definition for Many-Sorted TRSs. In that definition, the rule was just a pair of terms where the left term was restricted from being a variable. In the LCTRS's definition, we also require the left term to be a functional symbol, but we also restrict the terms so it can have a root symbol only from the special subset: $\Sigma_{terms} \setminus \Sigma_{theory}$. The most noticeable difference between these two definitions of the reduction rule is that in Many-Sorted TRSs the rule application can only be restricted by the sort declaration of the term that should be matched with the left-hand side of the rule, however, in LCTRSs every reduction rule should be equipped with a logical constraint, which also restricts the rule application. Also, the restrictions on variables are different.

A substitution γ respects the rule $l \to r\ [\varphi]$ if:

1. $Dom(\gamma) = Var(l) \cup Var(r) \cup Var(\varphi)$.
2. $\gamma(x)$ is a value $\forall x \in Var(\varphi) \cup (Var(r) \setminus Var(l))$.

3. $\varphi\gamma$ is valid (the constraint is satisfied).

Notion of the rewrite relation $\to_{\mathcal{R}}$ for the set of rules $\mathcal{R}$: it is a relation on terms, that is defined as the union of two other relations $\to_{\texttt{rule}}$ and $\to_{\texttt{calc}}$, where:

- $l\gamma \to_{\texttt{rule}} r\gamma$ if $l \to r\ [\varphi] \in \mathcal{R}$, and γ respects $l \to r\ [\varphi]$.
- $f(s_1, \ldots, s_n) \to_{\texttt{calc}} v$ if $f \in \Sigma_{theory} \setminus \Sigma_{terms}$, all s_i are values, and v is the value of $f(s_1, \ldots, s_n)$.
- If $\exists i : (t_i \to_{\texttt{rule}} u_i) \wedge (\forall j \neq i : t_j = u_j)$, then: $f(t_1, \ldots, t_n) \to_{\texttt{rule}} f(u_1, \ldots, u_n)$.
- If $\exists i : (t_i \to_{\texttt{calc}} u_i) \wedge (\forall j \neq i : t_j = u_j)$, then: $f(t_1, \ldots, t_n) \to_{\texttt{calc}} f(u_1, \ldots, u_n)$.

A logically constrained term rewriting system (LCTRS) is defined as the pair:

$$(\mathcal{T}erms(\Sigma\ \cup\ \mathcal{V}), \to_{\mathcal{R}})$$

Example 11. Consider the LCTRS with the following alphabet and the set of rewriting rules:

- Set of sorts: $\mathcal{S} = \{int, bool\}$.
- Alphabet: $\Sigma \cup \mathcal{V}$.
  $\{x : int\} \subseteq \mathcal{V}$
  $\Sigma = \Sigma_{terms}\ \cup\ \Sigma_{theory}$

$$\Sigma_{theory} = \left\{ \begin{array}{l} true : [\,] \Rightarrow bool, \\ false : [\,] \Rightarrow bool, \\ n : [\,] \Rightarrow int\ (n \in \mathbb{Z}), \\ + : [int \times int] \Rightarrow int, \\ \geq : [int \times int] \Rightarrow bool, \\ = : [int \times int] \Rightarrow bool, \\ \wedge : [bool \times bool] \Rightarrow bool, \\ \neg : [bool] \Rightarrow bool \end{array} \right\}$$

$$\Sigma_{terms} = \left\{ \begin{array}{l} sum : [int] \Rightarrow int, \\ n : int\ (\forall n \in \mathbb{Z}) \end{array} \right\}$$

- Interpretations:
  $\mathcal{I} = \{int \mapsto \mathbb{Z}, bool \mapsto \mathbb{B}\}$

$$\mathcal{J} = \left\{ \begin{array}{l} (\texttt{true} : [\,] \Rightarrow bool) \mapsto (\top : \mathbb{B}) \\ (\texttt{false} : [\,] \Rightarrow bool) \mapsto (\bot : \mathbb{B}) \\ (n : [\,] \Rightarrow int) \mapsto (n : \mathbb{Z}) \\ (+ : [int \times int] \Rightarrow int) \mapsto (+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}) \\ (\geq : [int \times int] \to bool) \mapsto (\geq : \mathbb{Z} \times \mathbb{Z} \to \mathbb{B}) \\ (= : [int \times int] \Rightarrow bool) \mapsto (= : \mathbb{Z} \times \mathbb{Z} \to \mathbb{B}) \\ (\wedge : [bool \times bool] \Rightarrow bool) \mapsto (\wedge : \mathbb{B} \times \mathbb{B} \to \mathbb{B}) \\ (\neg : [bool] \Rightarrow bool) \mapsto (\neg : \mathbb{B} \to \mathbb{B}) \end{array} \right\}$$

- Set of rules:
  $r1 : sum(x) \to 0\ [0 \geq x]$,
  $r2 : sum(x) \to x + sum(x + -1)\ [\neg(0 \geq x)]$

Application of the reduction rules to the term $sum(3)$:

$$sum(3) \xrightarrow{r_2} 3 + sum(3 + -1) \xrightarrow{\texttt{calc}} 3 + sum(2) \xrightarrow{r2} 3 + (2 + sum(2 + -1))$$
$$\xrightarrow{\texttt{calc}} 3 + (2 + sum(1)) \xrightarrow{r_2} 3 + (2 + (1 + sum(1 + -1)))$$
$$\xrightarrow{\texttt{calc}} 3 + (2 + (1 + sum(0))) \xrightarrow{r_1} 3 + (2 + (1 + 0))$$
$$\xrightarrow{\texttt{calc}} 3 + (2 + 1) \xrightarrow{\texttt{calc}} 3 + 3 \xrightarrow{\texttt{calc}} 6$$

Chapter 3

Related work

In this section, we will go over the current state of research on program translation into TRS, as well as research on program translation into LCTRS for termination analysis and program verification.

3.1 Imperative programs translation

Kanazawa and Nishida [4] focus on transforming imperative programs with function calls and global variables into Logically Constrained TRSs. The resulting system presents the transitions of the whole execution environment including global variables and a call stack with a running frame. The paper contains the correctness proof for the presented transformation. Consider the example presented in the research paper:

Example 12. Consider the following imperative program written in the programming language SIMP$^+$:

```
int num = 0;
int sum(int x) {
    int z = 0;
    num = num + 1;
    if (x <= 0) {
        z = 0;
    } else {
        z = sum(x - 1);
        z = x + z;
    }
    return z;
}
```

```
int main() {
    int z = 3;
    z = sum(z);
    return 0;
}
```

After translating this program into the LCTRS using the call stack for function calls, we obtain the following:

$$\mathcal{R} = \left\{ \begin{array}{l} sum(x) \to u_1(x, 0), \\ env(num, stack(u_1(x, z), s)) \to env(num + 1, stack(u_2(x, z), s)), \\ u_2(x, z) \to u_3(x, z)\ [x \leq 0], \\ u_2(x, z) \to u_5(x, z)\ [\neg(x \leq 0)], \\ u_3(x, z) \to u_4(x, 0), \\ u_4(x, z) \to u_9(x, z) \\ stack(u_5(x, z), s) \to stack(sum(x - 1), stack(u_6(x, z), s)), \\ stack(return(y), stack(u_6(x, z), s)) \to stack(u_7(x, y), s), \\ u_7(x, z) \to u_8(x, x + z), \\ u_8(x, z) \to u_9(x, z), \\ u_9(x, z) \to return(z), \\ \\ main() \to u_{10}(3), \\ stack(u_{10}(z), s) \to stack(sum(z), stack(u_{11}(z), s)), \\ stack(return(y), stack(u_{11}(z))) \to stack(u_{12}(y), s), \\ u_{12}(z) \to return(0) \end{array} \right\}$$

A new program translation method is presented in the more recent research of Nishida, Kojima, and Kato [5]. The article presents a new approach for translating imperative programming languages into LCTRSs using injective functions from configurations to terms. As an example programming language, the authors use SIMP$^+$ - a simple imperative programming language. In the paper, there is given the structural operational semantics (or small-step semantics) of SIMP$^+$. Because derivation sequences in small-step semantics are similar to the reduction chains in term rewriting, the translation process described in the article is based on transforming instances of inference rules in small-step semantics into the LCTRS rules. In this case, the injective functions help transform the configurations used in the inference rules instances into LCTRS terms. The authors introduce a special translation function $\mathfrak{T}$ which can be applied to the SIMP$^+$ program P:

$$\mathfrak{T}(P) = \{\mathfrak{T}_{\text{rule}}((\textbf{rule})[\rho])|(\textbf{rule})[\rho] \in Rules(P)\}$$

Where $\mathfrak{T}_{\text{rule}}$ is a function that allows transforming instances of small-step semantics inference rules into LCTRS rules. And $Rules(P)$ is the set of the intermediately instantiated inference rules for the program P.

The application of the transformation function to the semantic rule works in the following way:

$$\mathfrak{T}((\textbf{rule})[\rho]) = \xi(cnfg) \to \xi(cnfg')\ [\varphi \wedge v_1 = e_1 \wedge \cdots \wedge v_{n'} = e_{n'}]$$

Where function ξ translates the small-step semantics configurations into the LCTRS terms.

3.2 Termination analysis

Falke and Kapur [6] present an approach based on term rewriting for automated termination analysis of imperative programs operating on integers. The paper considers

transforming imperative programs into constrained term rewrite systems operating on integers. And, it presents the techniques for showing termination of $\mathcal{PA}$-based TRSs[1]. Consider the example translation presented in the research paper:

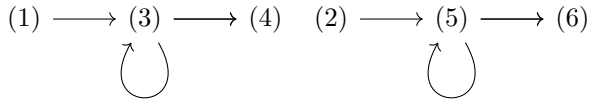
Example 13. Consider the following imperative program:

```
while (x > 0 && y > 0) {
    if (x > y) {
        while (x > 0) {
            x--;
            y++
        }
    } else {
        while (y > 0) {
            y--;
            x++
        }
    }
}
```

After translating this program into the LCTRS we obtain the following:

$eval_1(x, y) \to eval_2(x, y)\ [x > 0 \wedge y > 0 \wedge x > y]$ (1)
$eval_1(x, y) \to eval_3(x, y)\ [x > 0 \wedge y > 0 \wedge x \not> y]$ (2)
$eval_2(x, y) \to eval_2(x - 1, y + 1)\ [x > 0]$ (3)
$eval_2(x, y) \to eval_1(x, y)\ [x \not> 0]$ (4)
$eval_3(x, y) \to eval_3(x + 1, y - 1)\ [y > 0]$ (5)
$eval_3(x, y) \to eval_1(x, y)\ [y \not> 0]$ (6)

The obtained LCTRS can be further transformed into the termination graph:



The termination graph can be used in the termination processor, which can transform the given LCTRS into simpler systems that can be further used in the termination analysis.

In another work by Falke, Kapur, and Sinz [7] the authors presented the tool (KITTel), which allows the termination analysis of C programs using the intermediate language. The method used in the presented tool uses the intermediate language generated by the C compiler frontend to derive the term rewrite system. The termination of the obtained TRS is shown using the term rewriting techniques.

Consider the example translation of the C program into the LCTRS using the intermediate language representation:

Example 14.

[1] $\mathcal{PA}$ stands for Presburger arithmetic.

Original C code listing (Ackermann's function computation):

```
int ack(int m, int n) {
    if (m <= 0)
        return n + 1;
    else if (n <= 0)
        return ack(m - 1, 1);
    else
        return ack(m - 1, ack(m, n - 1));
}
```

Intermediate representation generated for the program (LLVM-IR):

```
define i32 @ack(i32 %m, i32 %n) {
entry:
    %0 = icmp sle i32 %m, 0
    br i1 %0, label %bb, label %bb1

bb:
    %1 = add nsw i32 %n, 1
    ret i32 %1

bb1:
    %2 = icmp sle i32 %n, 0
    br i1 %2, label %bb2, label %bb3

bb2:
    %3 = sub nsw i32 %m, 1
    %4 = call i32 @ack(i32 %3, i32 1)
    ret i32 %4

bb3:
    %5 = sub nsw i32 %n, 1
    %6 = call i32 @ack(i32 %m, i32 %5)
    %7 = sub nsw i32 %m, 1
    %8 = call i32 @ack(i32 %7, i32 %6)
    ret i32 %8
}
```

`int`-based TRS generated from the intermediate representation:

$state_{start}(v_m, v_n) \to state_{entry_{in}}(v_m, v_n)$
$state_{entry_{in}}(v_m, v_n) \to state_{bb_{in}}\ [v_m \leq 0]$
$state_{entry_{in}}(v_m, v_n) \to state_{bb1_{in}}(v_m, v_n)\ [v_m > 0]$
$state_{bb_{in}}(v_m, v_n) \to state_{stop}(v_m, v_n)$
$state_{bb1_{in}}(v_m, v_n) \to state_{bb2_{in}}(v_m, v_n)\ [v_n \leq 0]$

$state_{bb1_{in}}(v_m, v_n) \to state_{bb3_{in}}\ [v_n > 0]$
$state_{bb2_{in}}(v_m, v_n) \to state_{start}(v_m - 1, 1)$
$state_{bb2_{in}}(v_m, v_n) \to state_{stop}(v_m, v_n)$
$state_{bb3_{in}}(v_m, v_n) \to state_{start}(v_m - 1, z)$
$state_{bb3_{in}}(v_m, v_n) \to state_{stop}(v_m, v_n)$

The existing TRS techniques can be used to prove the termination of the generated system.

Finally, Giesl et al. [8] presented a technique for proving termination of algorithms on cyclic data structures (such as cyclic lists or graphs) in Java. The technique is based on automatically transforming Java programs into Term Rewriting Systems. It works as follows: first, the input Java program is transformed into Java byte code. After that, a JBC program is transformed into a termination graph. Afterward, a TRS is generated from the termination graph. The generated TRS is used to prove the program termination.

3.3 Program verification

Fuhs, Kop, and Nishida [9] present the new verification method for procedural programs. This method is based on a program transformation into a logically constrained term rewriting system. The authors extend the transformation methods based on integer TRS to support arbitrary data types. Additionally, the article where this verification method is presented contains the adaptation of existing induction methods to the LCTRS. Finally, the authors show the possibility of automatically verifying memory safety and proving the correctness of realistic functions. Consider the example from the research paper:

Example 15. Original C function:

```
int strlen(char *s) {
    for (int i = 0;;i++) {
        if (s[i] == 0)
            return i;
    }
}
```

LCTRS translation of the function:

$$\Sigma_{theory} = \left\{ \begin{array}{l} n : int\ (n \in \mathbb{Z}), \\ = :\ [int \times int] \Rightarrow bool, \\ < :\ [int \times int] \Rightarrow bool, \\ \leq :\ [int \times int] \Rightarrow bool, \\ \neq :\ [int \times int] \Rightarrow bool, \\ \wedge\ :\ [bool \times bool] \Rightarrow bool, \\ \vee\ :\ [bool \times bool] \Rightarrow bool, \\ size : [array] \Rightarrow int, \\ select : [array \times int] \Rightarrow int \end{array} \right\}$$

(1) $strlen(x) \to u(x, 0)$

(2) $u(x, i) \to error\ [i < 0 \vee i \geq size(x)]$

(3) $u(x, i) \to return(i)\ [0 \leq i \wedge i < size(x) \wedge select(x, i) = 0]$

(4) $u(x, i) \to u(x, i + 1)\ [0 \leq i \wedge i < size(x) \wedge select(x, i) \neq 0]$

Using the notion of rewriting induction, and in particular the inductive theorem of form $s \approx t\ [\varphi]$, it is possible to perform the check of the correctness of the translated function:

$$strlen(x) \approx return(n)$$
$$[0 \leq n < size(x) \wedge \forall i \in \{0, \ldots, n-1\}(select(x, i) \neq 0) \wedge (select(x, n) = 0)]$$

Chapter 4

Design of the While++ programming language

4.1 Syntax of While++ programming language

The syntactic notation description of While++ is based on the Backus-Naur form (BNF).
Firstly, we will describe the various syntactic categories of this programming language. For each category, there will be given a meta variable that will range over the constructs of each category. These meta-variables will later be used in the syntax description.
The list of meta-variables and categories:

- $n \in \mathbf{Num}$ - numerals.
- $x_{int} \in \mathbf{Var}_{int}$ - integer variables.
- $x_{bool} \in \mathbf{Var}_{bool}$ - boolean variables.
- $x_{string} \in \mathbf{Var}_{string}$ - string variables.
- $aexp \in \mathbf{Aexp}$ - arithmetic expressions.
- $bexp \in \mathbf{Bexp}$ - boolean expressions.
- $strexp \in \mathbf{Strexp}$ - string expressions.
- $S \in \mathcal{S}tm$ - program statements.

Type-dependent categories can be organized in the following way:

- Expressions in general:
  $expr \in \mathbf{Expr}$
  $\mathbf{Expr} = \mathbf{Aexp} \cup \mathbf{Bexp} \cup \mathbf{Strexp}$
- Variables in general:
  $x \in \mathbf{Var}$
  $\mathbf{Var} = \mathbf{Var}_{int} \cup \mathbf{Var}_{bool} \cup \mathbf{Var}_{string}$.

The given meta-variables can be primed and subscripted. For example: x'_{int}, $aexp_1$, or $bexp_2$. These constructions make the grammar notation more readable and expressive.

The syntax of the While++ programming language is defined by using the following grammar:

4.1.1 Syntax of the expressions

Arithmetic expressions	**Boolean expressions**	**String expressions**[1]
$aexp ::= \texttt{n}$	$bexp ::= \texttt{not}\ bexp$	$strexp ::= \texttt{".*"}$
$\mid x_{int}$	$\mid bexp_1\ \texttt{and}\ bexp_2$	$\mid x_{string}$
$\mid aexp_1\ \texttt{+}\ aexp_2$	$\mid bexp_1\ \texttt{or}\ bexp_2$	(the first option is a regular expression for a string of arbitrary length)
$\mid aexp_1\ \texttt{-}\ aexp_2$	$\mid bexp_1\ \texttt{=}\ bexp_2$	
$\mid aexp_1\ \texttt{*}\ aexp_2$	$\mid aexp_1\ \texttt{=}\ aexp_2$	
$\mid aexp_1\ \texttt{/}\ aexp_2$	$\mid aexp_1\ \texttt{>}\ aexp_2$	
$\mid aexp_1\ \texttt{\%}\ aexp_2$	$\mid aexp_1\ \texttt{<}\ aexp_2$	
$\mid \texttt{(}aexp\texttt{)}$	$\mid aexp_1\ \texttt{>=}\ aexp_2$	
	$\mid aexp_1\ \texttt{<=}\ aexp_2$	
	$\mid strexp_1\ \texttt{=}\ strexp_2$	
	$\mid \texttt{true}$	
	$\mid \texttt{false}$	
	$\mid x_{bool}$	
	$\mid \texttt{(}bexp\texttt{)}$	

We can unite these three types of expressions in the syntactic category **Expr**:

$$expr ::= aexp \mid bexp \mid strexp$$

Example 16. Some arithmetic expressions that can be constructed using the While++ grammar.

- 32
- 1 + 2 + 3
- x % 2
- x + y
- b * b - 4 * a * c
- (a + b) / (a - b)

[1] In the current version of While++ string expressions are quite limited: we do not include such operations as concatenation because the current version of Cora [3] doesn't support it. However, such functionality could easily be added in a future version.

Example 17. Some boolean expressions that can be constructed using the While++ grammar.

- `true`
- `true or false`
- a `or` b `and` c
- (`true = false`) `or` (1 `<` 2)
- `not` (`"Apple"` = `"Orange"`) `and` (`true or` a)

Example 18. Some string expressions that can be constructed using the While++ grammar.

- *str1*
- `"Hello World!"`

4.1.2 Syntax of the program statements

Program statements
S ::= `skip`
| *int* x_{int} `:=` *aexp*
| *bool* x_{bool} `:=` *bexp*
| *string* x_{string} `:=` *strexp*
| x_{int} `:=` *aexp*
| x_{bool} `:=` *bexp*
| x_{string} `:=` *strexp*
| `if` *bexp* `then` S_1 `else` S_2
| `while` *bexp* `do` S
| `printInt(`*aexp*`)`
| `printString(`*strexp*`)`
| `printBool(`*bexp*`)`
| `readInt(`x_{int}`)`
| `readString(`x_{string}`)`
| `readBool(`x_{bool}`)`
| `(` S `)`
| S_1`;` S_2

Example 19. Some simple programs that can be constructed using the While++ grammar:

- Program #1

```
int a := 0;
```

```
int b := 1;
int y := a - b
```

- Program #2

```
if not(1 = 1) then
    skip
else
    printInt(1)
```

- Program #3

```
int y := 0;
while (y <= 5) do (
    int x := 2;
    bool b := true;
    y := y + 1;
    printString("Done!")
)
```

Example 20. A more complex program that can be constructed using the While++ grammar:

- Euclidean algorithm

```
int a := 0;
int b := 0;
printString("Enter a:\n");
readInt(a);
printString("Enter b:\n");
readInt(b);
if b <= a then (
    int tmp := a;
    a := b;
    b := tmp
) else ( skip );
int aCopy := a;
int bCopy := b;
while not((a % b) <= 0) do (
    int tmp := a % b;
    a := b;
    b := tmp
);
printString("The gcd value for integers ");
printInt(aCopy); printString(" "); printInt(bCopy);
printString(" is: \n");
printInt(b); printString("\n")
```

The next section will give the natural semantics of the While++ program statements. Some type-dependent statements have identical semantics defined for different types. Because of this, for convenience, we introduce generalizations:

- Variable declaration:

  $type\ x_{type}$ `:=` $expr$

  Where:

  $type$ ::= `int` | `bool` | `string`

  > **Example 21.** The generalization $type\ x_{type}$ `:=` $expr$ refers to the following syntactic constructions:
  >
  > – $int\ x_{int}$ `:=` $aexp$
  > – $bool\ x_{bool}$ `:=` $bexp$
  > – $string\ x_{string}$ `:=` $strexp$

- Variable assignment:

  x_{type} `:=` $expr$

- Print statements:

  `print`$Type$`(`$expr$`)`

  Where:

  $Type$ ::= `Int` | `Bool` | `String`

  > **Example 22.** The generalization `print`$Type$`(`$expr$`)` refers to the following syntactic constructions:
  >
  > – `printInt(`$aexp$`)`
  > – `printBool(`$bexp$`)`
  > – `printString(`$strexp$`)`

- Read statements:

  `read`$Type$`(`x_{type}`)`

4.2 Semantics of While++ programming language

4.2.1 Program state

To describe the configuration of program variables at each step of the execution we use the concept of the *state*. To each variable the state will associate its current value. The state is represented as a function from variables to values:

$$\mathcal{S}tate = \mathbf{Var} \cup \mathbf{Var}_{list} \to \mathbf{Val} \cup \mathbb{L}$$

The set of variables is defined as: $\mathbf{Var} = \mathbf{Var}_{int} \cup \mathbf{Var}_{bool} \cup \mathbf{Var}_{string}$. The set of values is defined as: $\mathbf{Val} = \mathbb{Z} \cup \mathbb{B} \cup \mathbb{S}$, where: $\mathbb{S} = \{x \mid x \text{ is a valid UTF-8 string}\}$.

The set of "variables" $\mathbf{Var}_{list} = \{\texttt{In}, \texttt{Out}\}$ contain special entities used to represent the I/O input and output lists in the While++ semantics.
The set of lists is defined as:

$$\mathbb{L} = \{x \mid x \text{ is a list which can hold elements of type: } \mathbb{Z} \cup \mathbb{B} \cup \mathbb{S}\}$$

To refer to the value of the variable x in the state s we use the following notation: $s\ x$.

Example 23. Consider the state s. Suppose in this state the variables x, y, z are defined. The variable x is of type *int*, and it is assigned the value 5, the variable y is of type *bool*, and it is assigned the value `true`. Finally, the variable z is of type *string*, and it is assigned the value `"Hello World!"`. Using the state notation we can represent the information about the variables in the following way:

$s\ x = 5, x \in \mathbf{Var}_{int}$
$s\ y = \texttt{true}, y \in \mathbf{Var}_{bool}$
$s\ z = \texttt{"Hello World!"}, z \in \mathbf{Var}_{string}$

There also exists the special notation to represent the assignment of the new value to the variable in the state:

$$s' = s[x \mapsto val]$$

4.2.2 Semantics of expressions

Like almost every modern programming language While++ supports different types of expressions. It can be arithmetic expressions used to evaluate some formula, it can be boolean expressions to evaluate the while loop condition, and it can even be string expressions used in the assignment of the string literals to variables, or used directly in print statement.

As was mentioned above, While++ supports 3 types of expressions: the set **Aexp** consists of all possible arithmetic expressions, the set **Bexp** consists of all possible boolean expressions, and finally the set **Strexp** consists of all possible string expressions.

The meaning of expression is defined by the use of the total function $\mathcal{E}val : \mathbf{Expr} \times \mathcal{S}tate \to \mathbf{Val}$. This function takes two arguments: the syntactic construct and the state. The function supports special double-bracket notation:

$$\mathcal{E}val[\![expr]\!]s = val$$

Using this function we can determine the value of the expression in the given state.

Example 24. Suppose we have a state s, such that:

$s\ x = 2, x \in \mathbf{Var}_{int}$
$s\ y = 3, y \in \mathbf{Var}_{int}$
$s\ a = \top, a \in \mathbf{Var}_{bool}$
$s\ b = \bot, b \in \mathbf{Var}_{bool}$
$s\ strOne = \texttt{"Apple"}, strOne \in \mathbf{Var}_{string}$
$s\ strTwo = \texttt{"Orange"}, strTwo \in \mathbf{Var}_{string}$

Then we will have the following values of the expressions:

$\mathcal{E}val[\![\texttt{((}x\ \texttt{*}\ y\texttt{)} + x\texttt{)} - \texttt{1}]\!]s = 7$
$\mathcal{E}val[\![\texttt{(}a\ \texttt{and (not}\ b\texttt{))}\ \texttt{and false}]\!]s = \bot$
$\mathcal{E}val[\![strOne]\!]s = \texttt{"Apple"}$
$\mathcal{E}val[\![\texttt{"Hello"}]\!]s = \texttt{"Hello"}$

The function evaluates the expressions differently depending on the type of the expression, because we have multiple expression evaluation functions for every type of expression. Namely:

- For arithmetic expressions: $\mathcal{A} : \mathbf{Aexp} \times \mathcal{S}tate \to \mathbb{Z}$
- For boolean expressions: $\mathcal{B} : \mathbf{Bexp} \times \mathcal{S}tate \to \mathbb{B}$
- For string expressions: $\mathcal{S}tr : \mathbf{Strexp} \times \mathcal{S}tate \to \mathbb{S}$

Like the $\mathcal{E}val$ function, the functions given above also support the double-bracket notation. Using these functions, we can formally define the expression evaluation function:

$$\mathcal{E}val[\![expr]\!]s = \begin{cases} \mathcal{A}[\![expr]\!]s & \text{if } \mathcal{A}[\![expr]\!]s \downarrow \\ \mathcal{B}[\![expr]\!]s & \text{if } \mathcal{B}[\![expr]\!]s \downarrow \\ \mathcal{S}tr[\![expr]\!]s & \text{if } \mathcal{S}tr[\![expr]\!]s \downarrow \\ \mathcal{E}val[\![expr]\!]s \uparrow & \text{otherwise} \end{cases}$$

Semantics of arithmetic expressions

To define semantics of the arithmetic expressions we define the function $\mathcal{A}$:

$$\mathcal{A}[\![\texttt{n}]\!]s = n, n \in \mathbb{Z}$$

$$\mathcal{A}[\![x]\!]s = \begin{cases} s\ x & \text{if } x \in \mathbf{Var}_{int} \ \wedge \ x \in Dom(s) \\ \mathcal{A}[\![x]\!]s \uparrow & \text{otherwise} \end{cases}$$

$$\mathcal{A}[\![aexp_1 \ \texttt{+} \ aexp_2]\!]s = \mathcal{A}[\![aexp_1]\!]s + \mathcal{A}[\![aexp_2]\!]s$$
$$\mathcal{A}[\![aexp_1 \ \texttt{-} \ aexp_2]\!]s = \mathcal{A}[\![aexp_1]\!]s - \mathcal{A}[\![aexp_2]\!]s$$
$$\mathcal{A}[\![aexp_1 \ \texttt{*} \ aexp_2]\!]s = \mathcal{A}[\![aexp_1]\!]s \cdot \mathcal{A}[\![aexp_2]\!]s$$

Example 25. Suppose that:

$s\ x = 2, x \in \mathbf{Var}_{int}$
$s\ y = 3, y \in \mathbf{Var}_{int}$
Then, we have:

$$\begin{aligned} \mathcal{A}[\![\texttt{((}x\ \texttt{*}\ y\texttt{)} \ \texttt{+}\ x\texttt{)} \ \texttt{-}\ \texttt{1}]\!]s &= \mathcal{A}[\![\texttt{((}x\ \texttt{*}\ y\texttt{)} \ \texttt{+}\ x\texttt{)}]\!]s - \mathcal{A}[\![1]\!]s \\ &= \mathcal{A}[\![\texttt{(}x\ \texttt{*}\ y\texttt{)}]\!]s + \mathcal{A}[\![x]\!]s - \mathcal{A}[\![1]\!]s \\ &= \mathcal{A}[\![x]\!]s * \mathcal{A}[\![y]\!]s + \mathcal{A}[\![x]\!]s - \mathcal{A}[\![1]\!]s \\ &= 2 * 3 + 2 - 1 \\ &= 7 \end{aligned}$$

To make the semantics of the arithmetic expressions be consistent with the Cora [3] implementation we will use the Euclidean definition of the integer division and modulo operations [10]. Let's introduce the following abbreviations: $a = \mathcal{A}[\![aexp_1]\!]s$ and $b = \mathcal{A}[\![aexp_2]\!]s$.

Then the definition of the integer division and modulo operations can be defined as follows:

$$\mathcal{A}[\![aexp_1 \texttt{ / } aexp_2]\!]s = \begin{cases} 0 & \text{if } b = 0 \\ sign(a \cdot b) \cdot \left\lfloor \frac{|a|}{|b|} \right\rfloor & \text{if } a \geq 0 \\ sign(a \cdot b) \cdot \left\lceil \frac{|a|}{|b|} \right\rceil & \text{otherwise} \end{cases}$$

$$\mathcal{A}[\![aexp_1 \texttt{ \% } aexp_2]\!]s = \begin{cases} 0 & \text{if } b = 0 \\ |a| - |b| \cdot \left\lfloor \frac{|a|}{|b|} \right\rfloor & \text{if } (a \geq 0) \vee (|a| - |b| \cdot \left\lfloor \frac{|a|}{|b|} \right\rfloor = 0) \\ |b| - (|a| - |b| \cdot \left\lfloor \frac{|a|}{|b|} \right\rfloor) & \text{otherwise} \end{cases}$$

Example 26. Suppose that:

$s\ x = 24, x \in \mathbf{Var}_{int}$
$s\ y = 7, y \in \mathbf{Var}_{int}$
Then, we have:

$$\begin{aligned} \mathcal{A}[\![\texttt{(}x \texttt{ / } y\texttt{) + (}x \texttt{ \% } y\texttt{)}]\!]s &= \mathcal{A}[\![\texttt{(}x \texttt{ / } y\texttt{)}]\!]s + \mathcal{A}[\![\texttt{(}x \texttt{ \% } y\texttt{)}]\!]s \\ &= (sign(24 \cdot 7) \cdot \left\lfloor \frac{|24|}{|7|} \right\rfloor) + (|24| - |7| \cdot \left\lfloor \frac{|24|}{|7|} \right\rfloor) \\ &= (1 \cdot 3) + (24 - 7 \cdot 3) \\ &= 3 + 3 \\ &= 6 \end{aligned}$$

Example 27. Suppose that:

$s\ x = -25, x \in \mathbf{Var}_{int}$
$s\ y = 7, y \in \mathbf{Var}_{int}$
Then, we have:

$$\begin{aligned} \mathcal{A}[\![\texttt{(}x \texttt{ / } y\texttt{) + (}x \texttt{ \% } y\texttt{)}]\!]s &= \mathcal{A}[\![\texttt{(}x \texttt{ / } y\texttt{)}]\!]s + \mathcal{A}[\![\texttt{(}x \texttt{ \% } y\texttt{)}]\!]s \\ &= (sign(-25 \cdot 7) \cdot \left\lceil \frac{|-25|}{|7|} \right\rceil) \\ &\quad + (|7| - (|-25| - |7| \cdot \left\lfloor \frac{|-25|}{|7|} \right\rfloor)) \\ &= (-1 \cdot 4) + (7 - (25 - 7 \cdot 3)) \\ &= -4 + (7 - 4) \\ &= -4 + 3 \\ &= -1 \end{aligned}$$

Semantics of boolean expressions

To define the semantics of boolean expressions we define the function $\mathcal{B}$:

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

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

$$\mathcal{B}[\![x]\!]s = \begin{cases} s\ x & \text{if } x \in \mathbf{Var}_{bool}\ \wedge\ x \in Dom(s) \\ \mathcal{B}[\![x]\!]s \uparrow & \text{otherwise} \end{cases}$$

$\mathcal{B}[\![\texttt{not}\ \ bexp]\!]s = \neg\mathcal{B}[\![bexp]\!]s$

$\mathcal{B}[\![bexp_1\ \texttt{and}\ bexp_2]\!]s = \mathcal{B}[\![bexp_1]\!]s\ \wedge\ \mathcal{B}[\![bexp_2]\!]s$

$\mathcal{B}[\![bexp_1\ \texttt{or}\ bexp_2]\!]s = \mathcal{B}[\![bexp_1]\!]s\ \vee\ \mathcal{B}[\![bexp_2]\!]s$

$\mathcal{B}[\![bexp_1 = bexp_2]\!]s = \mathcal{B}[\![bexp_1]\!]s = \mathcal{B}[\![bexp_2]\!]s$

$$\mathcal{B}[\![aexp_1 = aexp_2]\!]s = \begin{cases} \top \text{ if } \mathcal{A}[\![aexp_1]\!]s = \mathcal{A}[\![aexp_2]\!]s \\ \bot \text{ if } \mathcal{A}[\![aexp_1]\!]s \neq \mathcal{A}[\![aexp_2]\!]s \end{cases}$$

$$\mathcal{B}[\![aexp_1 > aexp_2]\!]s = \begin{cases} \top \text{ if } \mathcal{A}[\![aexp_1]\!]s > \mathcal{A}[\![aexp_2]\!]s \\ \bot \text{ if } \mathcal{A}[\![aexp_1]\!]s \leq \mathcal{A}[\![aexp_2]\!]s \end{cases}$$

$$\mathcal{B}[\![aexp_1 < aexp_2]\!]s = \begin{cases} \top \text{ if } \mathcal{A}[\![aexp_1]\!]s < \mathcal{A}[\![aexp_2]\!]s \\ \bot \text{ if } \mathcal{A}[\![aexp_1]\!]s \geq \mathcal{A}[\![aexp_2]\!]s \end{cases}$$

$$\mathcal{B}[\![aexp_1\ \texttt{>=}\ aexp_2]\!]s = \begin{cases} \top \text{ if } \mathcal{A}[\![aexp_1]\!]s \geq \mathcal{A}[\![aexp_2]\!]s \\ \bot \text{ if } \mathcal{A}[\![aexp_1]\!]s < \mathcal{A}[\![aexp_2]\!]s \end{cases}$$

$$\mathcal{B}[\![aexp_1\ \texttt{<=}\ aexp_2]\!]s = \begin{cases} \top \text{ if } \mathcal{A}[\![aexp_1]\!]s \leq \mathcal{A}[\![aexp_2]\!]s \\ \bot \text{ if } \mathcal{A}[\![aexp_1]\!]s > \mathcal{A}[\![aexp_2]\!]s \end{cases}$$

$$\mathcal{B}[\![strexp_1 = strexp_2]\!]s = \begin{cases} \top \text{ if } \mathcal{S}tr[\![strexp_1]\!]s = \mathcal{S}tr[\![strexp_2]\!]s \\ \bot \text{ if } \mathcal{S}tr[\![strexp_1]\!]s \neq \mathcal{S}tr[\![strexp_2]\!]s \end{cases}$$

Example 28. Suppose that:

$s\ x = 2, x \in \mathbf{Var}_{int}$

$s\ y = 3, y \in \mathbf{Var}_{int}$

$s\ a = \top, a \in \mathbf{Var}_{bool}$

$s\ b = \bot, b \in \mathbf{Var}_{bool}$

$s\ apple =$ "Apple"

Then, we have:

$\mathcal{B}[\![(x\ \texttt{>=}\ y)\ \texttt{or}\ (a\ \texttt{and}\ (\texttt{not}\ (\text{"Orange"} = apple)))]\!]s$
$= \mathcal{B}[\![(x\ \texttt{>=}\ y)]\!]s \vee \mathcal{B}[\![(a\ \texttt{and}\ (\texttt{not}\ (\text{"Orange"} = apple)))]\!]s$
$= (\mathcal{A}[\![x]\!]s \geq \mathcal{A}[\![y]\!]s) \vee (\mathcal{B}[\![a]\!]s \wedge \mathcal{B}[\![\texttt{not}\ (\text{"Orange"} = apple)]\!]s)$
$= (\mathcal{A}[\![x]\!]s \geq \mathcal{A}[\![y]\!]s) \vee (\mathcal{B}[\![a]\!]s \wedge \neg(\mathcal{S}tr[\![\text{"Orange"}]\!]s = \mathcal{S}tr[\![apple]\!]s))$
$= (2 \geq 3) \vee (\top \wedge \neg(\text{"Orange"} = \text{"Apple"}))$
$= (2 \geq 3) \vee (\top \wedge \neg\bot)$
$= (2 \geq 3) \vee (\top \wedge \top)$
$= \bot \vee \top$
$= \top$

Semantics of string expressions

To define the semantics of string expressions we define the function $\mathcal{S}tr$:

$\mathcal{S}tr[\![\textbf{"str"}]\!]s = \textbf{str}, \textbf{str} \in \mathbb{S}$

$$\mathcal{S}tr[\![x]\!]s = \begin{cases} s\ x & \text{if } x \in \textbf{Var}_{string} \ \wedge \ x \in Dom(s) \\ \mathcal{S}tr[\![x]\!]s \uparrow & \text{otherwise} \end{cases}$$

Example 29. Suppose that: $s\ x =$ "Hello!", $x \in \textbf{Var}_{string}$

Then, we have:

$\mathcal{S}tr[\![x]\!]s =$ "Hello!"

Some more examples

Example 30. Suppose that:

$s\ x = 2, x \in \textbf{Var}_{int}$
$s\ a = \top, a \in \textbf{Var}_{bool}$
$s\ apple =$ "Apple"

Then, we have:

$$\begin{aligned} \mathcal{E}val[\![x+1]\!]s &= \mathcal{A}[\![x+1]\!]s \\ &= \mathcal{A}[\![x]\!]s + \mathcal{A}[\![1]\!]s \\ &= 2+1 \\ &= 3 \end{aligned}$$

Example 31. Suppose that:

$s\ x = 2, x \in \textbf{Var}_{int}$
$s\ a = \top, a \in \textbf{Var}_{bool}$
$s\ apple =$ "Apple"

Then, we have:

$\mathcal{E}val[\![x+a]\!]s \uparrow$, because $\mathcal{A}[\![x+a]\!] \uparrow$, and $\mathcal{B}[\![x+a]\!] \uparrow$, and $\mathcal{S}tr[\![x+a]\!] \uparrow$.

4.2.3 Semantics of lists

While++ supports simple I/O operations: the language allows printing expressions and reading values into the variables. To appropriately store the output and provide the input to the programs, we define the special list construction, which will be used to define the semantics for the I/O operations.

In While++, we have only two "variables" that represent lists: `In`, `Out`. These "variables" represent the input and output streams of the program, and they are for internal

implementation only. The only interaction with these "variables" is possible via statements `print`$Type$ and `read`$Type$. We can view `In`, `Out` as "variables" for a program state; however, in reality, these entities are not the variables from the While++ syntax perspective.

The lists can be constructed and modified using the following functions:

- **consT** : $(\mathbb{Z} \cup \mathbb{B} \cup \mathbb{S}) \times \mathbb{L} \to \mathbb{L}$ - this is a general pattern of the function that allows us to add an element of the given type to the list.
  - For the integer values we will use **consI** : $\mathbb{Z} \times \mathbb{L} \to \mathbb{L}$.
  - For boolean values: **consB** : $\mathbb{B} \times \mathbb{L} \to \mathbb{L}$.
  - And for strings: **consS** : $\mathbb{S} \times \mathbb{L} \to \mathbb{L}$.

  > **Example 32.** Consider the list $l = [1, \top, \texttt{"Hello!"}]$. We can add the new element to this list in the following way:
  >
  > $$\textbf{consI}(5, l) = [5, 1, \top, \texttt{"Hello!"}]$$

- **headT** : $\mathbb{L} \to (\mathbb{Z} \cup \mathbb{B} \cup \mathbb{S})$ - this is a general pattern of the function that returns the first element of the list.
  - For the integer values we will use **headI** : $\mathbb{L} \to \mathbb{Z}$.
  - For boolean values - **headB** : $\mathbb{L} \to \mathbb{B}$.
  - And for strings: **headS** : $\mathbb{L} \to \mathbb{S}$.

  The **headT** function is considered undefined if the type of the first element of the list doesn't correspond to the type of the **headT** function.

  > **Example 33.** Consider the list $l = [1, \top, \texttt{"Hello!"}]$. If we apply the function **headI** on the list l, then we will have the following:
  >
  > $$\textbf{headI}(l) = 1$$
  >
  > Also, we can say that the functions **headS** and **headB** are undefined for the list l.

- **tail** : $\mathbb{L} \to \mathbb{L}$ - this is the function that removes the first element of the list and returns the remaining part of the list. The **tail** function is undefined if it is applied to the empty list.

  > **Example 34.** Consider the list $l = [1, \top, \texttt{"Hello!"}]$. If we apply the function **tail** on the list l, then we will have the following:
  >
  > $$\textbf{tail}(l) = [\top, \texttt{"Hello!"}]$$

4.2.4 Semantics of program statements

To describe the semantics of the While++ language, we will use the natural semantics. The application of the language statements will be described in the form of derivation trees.

In natural semantics, we are concerned about the initial and the final state of execution. So, the transition relation specifies the relationship between the initial state and the final state for each program statement. The transition is denoted as $\langle S, s\rangle \to s'$ (Basically, it means that the execution of the statement S in the state s will result in the state s'). All transitions are described with special rules. These rules have a general form:

$$\frac{\langle S_1, s_1\rangle \to s_1' \ldots \langle S_n, s_n\rangle \to s_n'}{\langle S, s\rangle \to s'} \text{ if } \varphi$$

The sub-statements $S_1, S_2, \ldots, S_n$ are the immediate constituents of the statement S. The rule consists of these constituents (basically premises) and one conclusion. Additionally, the rule can have some conditions φ which can contain boolean expressions $bexp \in \mathbf{Bexp}$ evaluated in the state s.

Natural semantics of While++

While++ supports different types of variables and expressions. Because of this, for some constructions we need to define multiple semantics rules with identical behavior, but for different types. To simplify the rule declaration, we will give only a generalized version of those rules to increase the expressiveness of the rule's description. We give the details on the generalizations' construction specifically for every type-dependent rule.

Skip statement	
Rule Name	$[\text{skip}_{\text{ns}}]$
Semantics	$\langle \texttt{skip}, s\rangle \to s \quad \text{if } \varphi$
Condition	$\varphi = \top$

Variable declaration	
Rule Name	$[\text{declare}_{\text{type}}]$
Semantics	$\langle type\ var\ \texttt{:=}\ expr, s\rangle \to s[var \mapsto \mathcal{E}val_{type}[\![expr]\!]s] \quad \text{if } \varphi$
Condition	$\varphi = (\forall x' \in Dom(s) : x' \neq var \wedge var \in \mathbf{Var}_{type} \wedge \mathcal{E}val_{type}[\![expr]\!]s\downarrow)$

The rule updates the state by adding a mapping for a variable of a certain type. In the specific instances of this rule the keyword *type* and the function $\mathcal{E}val_{type}$ need to be replaced accordingly by:

1. *int* and $\mathcal{A}$ - for integer variables
2. *bool* and $\mathcal{B}$ - for boolean variables
3. *string* and $\mathcal{S}tr$ - for string variables

Example 35. Variable declaration rule for the boolean variables:

$[\text{declare}_{\text{bool}}] \quad \langle bool\ var\ \texttt{:=}\ expr, s\rangle \to s[var \mapsto \mathcal{B}[\![expr]\!]s]$
if $(\exists x' \in Dom(s) : x' = var \wedge var \in \mathbf{Var}_{bool} \wedge \mathcal{B}[\![expr]\!]s\downarrow)$

Variable assignment	
Rule Name	$[\text{ass}_{\text{type}}]$
Semantics	$\langle var \texttt{ := } expr, s\rangle \rightarrow s[var \mapsto \mathcal{E}val_{type}[\![expr]\!]s]$ if φ
Condition	$\varphi = (var \in Dom(s) \wedge var \in \mathbf{Var}_{type} \wedge \mathcal{E}val_{type}[\![expr]\!]s\downarrow)$

The rule is updating the variable value in the state. This rule uses the similar generalization approach as for the variable declaration rule. The only difference is that here we do not mention the type of the variable.

Example 36. Variable assignment rule for the integer variables:

$[\text{ass}_{\text{int}}]$ $\langle var \texttt{ := } expr, s\rangle \rightarrow s[var \mapsto \mathcal{A}[\![expr]\!]s]$
if $(var \in Dom(s) \wedge var \in \mathbf{Var}_{int} \wedge \mathcal{A}[\![expr]\!]s\downarrow)$

Print statement	
Rule Name	$[\text{print}_{\text{type}}]$
Semantics	$\langle \texttt{printType(}expr\texttt{)}, s\rangle \rightarrow s[\texttt{Out} \mapsto \mathbf{consT}(\mathcal{E}val_{type}[\![expr]\!]s, s\ \texttt{Out})]$ if φ
Condition	$\varphi = (\texttt{Out} \in Dom(s) \wedge \mathcal{E}val_{type}[\![expr]\!]s\downarrow)$

The rule is updating the output list of the program by adding there the result of the expression *expr* evaluated in state s. In the specific instances of this rule the command name `printType` and the function $\mathcal{E}val_{type}$ need to be replaced accordingly by:

1. `printInt` and $\mathcal{A}$ - for integer expressions.
2. `printBool` and $\mathcal{B}$ - for boolean expressions.
3. `printString` and $\mathcal{S}tr$ - for string expressions.

Example 37. Print rule for the string expressions:

$[\text{print}_{\text{type}}]$ $\langle \texttt{printString(}expr\texttt{)}, s\rangle \rightarrow s[\texttt{Out} \mapsto \mathbf{consS}(\mathcal{S}tr[\![expr]\!]s, s\ \texttt{Out})]$
if $(\texttt{Out} \in Dom(s) \wedge \mathcal{S}tr[\![expr]\!]s\downarrow)$

Read statement	
Rule Name	$[\text{read}_{\text{type}}]$
Semantics	$\langle \texttt{readType(}var\texttt{)}, s\rangle \rightarrow s[var \mapsto \mathbf{headT}(s\ \texttt{In}), \texttt{In} \mapsto \mathbf{tail}(s\ \texttt{In})]$ if φ
Condition	$\varphi = (var \in Dom(s) \ \wedge\ \texttt{In} \in Dom(s) \ \wedge\ \mathbf{headT}(s\ \texttt{In})\downarrow \ \wedge \mathbf{tail}(s\ \texttt{In})\downarrow)$

The rule is taking the top element from the input list and assigns the value to the corresponding variable. After that, the rule updates the input list by removing the element that was read. In the specific instances of this rule the command name `readType` and the function **headT** need to be replaced accordingly by:

1. `readInt` and **headI** - for integer variables.
2. `readBool` and **headB** - for boolean variables.
3. `readString` and **headS** - for string variables.

Example 38. Read rule for the int variables:

$[\text{read}_{\text{int}}] \quad \langle \texttt{readInt(}var\texttt{)}, s\rangle \to s[var \mapsto \textbf{headI}(s\ \texttt{In}), \texttt{In} \mapsto \textbf{tail}(s\ \texttt{In})]$
if $(var \in Dom(s) \ \wedge\ \texttt{In} \in Dom(s) \ \wedge\ \textbf{headI}(s\ \texttt{In})\downarrow \ \wedge \textbf{tail}(s\ \texttt{In})\downarrow)$

Composition of statements	
Rule Name	$[\text{comp}_{\text{ns}}]$
Semantics	$\dfrac{\langle S_1, s\rangle \to s' \quad \langle S_2, s'\rangle \to s''}{\langle S_1\ \texttt{;}\ S_2, s\rangle \to s''}$ if φ
Condition	$\varphi = \top$

The rule $[\text{comp}_{\text{ns}}]$ describes the execution of the composition of two statements in While++. To execute $S_1\ \texttt{;}\ S_2$ in the state s, we first need to execute the statement S_1 in the state s. Assuming that this execution yields some state s', we need to execute the statement S_2 in the state s'. Finally, assuming the execution of the statement S_2 yields the state s'', we conclude that the result of the execution of the composition of the statements S_1 and S_2 is this state s''.

Example 39.
Application of the rule $[\text{comp}_{\text{ns}}]$ to the statement $S =$ `int y := x + 1; x := y + 1`, and the state s (such that $s\ \texttt{x} = 1$):

$$\dfrac{\langle \texttt{int y := x + 1}, s\rangle \to s' \ [\text{declare}_{\text{int}}] \qquad \langle \texttt{x := y + 1}, s'\rangle \to s'' \ [\text{ass}_{\text{int}}]}{\langle \texttt{int y := x + 1; x := y + 1}, s\rangle \to s''}\ [\text{comp}_{\text{ns}}]$$

$s' = s[y \mapsto 2]$, and $s'' = s'[x \mapsto 3]$.

Finally, we consider conditions and loops. Note that in branches of the `if` statement or in the body of the `while` loop, it is possible to have variable declarations, however to ensure the determinism of the program, after leaving the scope of the branch we need to remove from the final state the variables that were declared inside the branch.
We introduce the special function:

$$LeaveScope(s, s') = s''$$

Where: $Dom(s'') = Dom(s)$, $\forall x \in Dom(s) : s''\ x = s'\ x$.

If statement: "true" rule	
Rule Name	$[\text{if}^{\top}_{\text{ns}}]$
Semantics	$\dfrac{\langle S_1, s\rangle \to s'}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s\rangle \to LeaveScope(s, s')}$ if φ
Condition	$\varphi = \mathcal{B}[\![b]\!]s$

If statement: "false" rule	
Rule Name	$[\text{if}^{\bot}_{\text{ns}}]$
Semantics	$\dfrac{\langle S_2, s\rangle \to s'}{\langle \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, s\rangle \to \mathit{LeaveScope}(s, s')}$ if φ
Condition	$\varphi = \neg\mathcal{B}[\![b]\!]s$

The rules $[\text{if}^{\top}_{\text{ns}}]$ and $[\text{if}^{\bot}_{\text{ns}}]$ describe the execution of the conditional statement in While++. Each of these rules can be applied depending on whether the condition of the `if` will be evaluated to $\top$ or $\bot$. To execute `if` b `then` S_1 `else` S_2 in state s, we first need to evaluate the boolean expression b in the state s: $\mathcal{B}[\![b]\!]s$. Depending on the resulting value, we execute the statement S_1 or the statement S_2 in the state s. Assuming that this execution yields some state s', we conclude that the result of the execution of the conditional statement is the state $\mathit{LeaveScope}(s, s')$.

Example 40. Application of the rule $[\text{if}^{\top}_{\text{ns}}]$ to the statement:

$$S = \texttt{if x = 1 then int y := x + 1 else skip}$$

, and the state s (such that $s\,\texttt{x} = 1$):

$$\frac{\langle \texttt{int y := x + 1}, s\rangle \to s' \;[\text{declare}_{\text{int}}]}{\langle \texttt{if x = 1 then int y := x + 1 else skip}, s\rangle \to \mathit{LeaveScope}(s, s')}\;[\text{if}^{\top}_{\text{ns}}]$$

$s' = s[\texttt{y} \mapsto 2], \mathit{LeaveScope}(s, s') = s.$

While statement: "true" rule	
Rule Name	$[\text{while}^{\top}_{\text{ns}}]$
Semantics	$\dfrac{\langle S, s\rangle \to s' \quad \langle \texttt{while } b \texttt{ do } S, \mathit{LeaveScope}(s, s')\rangle \to s''}{\langle \texttt{while } b \texttt{ do } S, s\rangle \to \mathit{LeaveScope}(s, s'')}$ if φ
Condition	$\varphi = \mathcal{B}[\![b]\!]s$

While statement: "false" rule	
Rule Name	$[\text{while}^{\bot}_{\text{ns}}]$
Semantics	$\langle \texttt{while } b \texttt{ do } S, s\rangle \to s$ if φ
Condition	$\varphi = \neg\mathcal{B}[\![b]\!]s$

The rules $[\text{while}^{\top}_{\text{ns}}]$ and $[\text{while}^{\bot}_{\text{ns}}]$ describe the execution of the while loop statement in While++. Each rule can be applied depending on whether the condition of the `while` will be evaluated to $\top$ or $\bot$. To execute `while` b `do` S in state s, we need to evaluate the boolean constant b in the state s: $\mathcal{B}[\![b]\!]s$. If b evaluates to $\top$ in the state s, we

execute the statement S in the state s and obtain some new program state s'. This way, we can execute one loop iteration. If b evaluates to $\bot$ in the state s, we terminate the loop execution and conclude that the result of the execution of the while loop statement is the state s.

Example 41. Application of the rule $[\text{while}^{\top}_{\text{ns}}]$ to the statement:
$S =$ `while x<2 do x:=x+1`, and the state s (such that $s\,\texttt{x} = 1$):

$$\dfrac{\langle \texttt{x:=x+1}, s\rangle \to s' \;[\text{ass}_{\text{int}}] \qquad \langle \texttt{while x<2 do x:=x+1}, s''\rangle \to s'' \;[\text{while}^{\bot}_{\text{ns}}]}{\langle \texttt{while x<2 do x:=x+1}, s\rangle \to \mathit{LeaveScope}(s, s'')}\;[\text{while}^{\top}_{\text{ns}}]$$

$s' = s[x \mapsto 2], s'' = \mathit{LeaveScope}(s, s') = s'$, and $\mathit{LeaveScope}(s, s'') = s'$.

Chapter 5

Translation of While++ programs into LCTRS

5.1 Translating variables

Logically Constrained TRS uses variables different from ones defined in the While++ semantics. For the translation of While++ programs into the LCTRS, we need to define the translation of While++ variables into the LCTRS variables.

We define a special translation function that allows translating the natural semantics variables into the LCTRS variables:

$$\mathcal{VT}(\mathtt{x} \in \mathbf{Var} \cup \mathbf{Var}_{list}) = \begin{cases} x : int & \text{if } \mathtt{x} \in \mathbf{Var}_{int}, \\ x : bool & \text{if } \mathtt{x} \in \mathbf{Var}_{bool}, \\ x : string & \text{if } \mathtt{x} \in \mathbf{Var}_{string}, \\ In : list & \text{if } \mathtt{x} = \mathtt{In}, \\ Out : list & \text{if } \mathtt{x} = \mathtt{Out} \end{cases}$$

Additionally we define the translation for the vector that can include elements from $\mathbf{Var} \cup \mathbf{Var}_{list}$:

$$\mathcal{VT}(\vec{x}) = (\mathcal{VT}(x_1), \ldots, \mathcal{VT}(x_n)) \text{ if } \vec{x} = (x_1, \ldots, x_n)$$

5.2 Translating expressions into LCTRS

While++ supports operations with different types of expressions. In order to translate these expressions into LCTRS terms we define a special expression translation function:

$$\mathcal{ET} : \mathbf{Expr} \to \mathcal{T}erms(\Sigma_{theory} \cup \mathcal{V})$$

It is defined as follows:

$$\mathcal{ET}(expr) = \begin{cases} \mathcal{AT}(expr) & \text{if } \mathcal{AT}(expr) \downarrow \\ \mathcal{BT}(expr) & \text{if } \mathcal{BT}(expr) \downarrow \\ \mathcal{S}tr\mathcal{T}(expr) & \text{if } \mathcal{S}tr\mathcal{T}(expr) \downarrow \\ \mathcal{ET}(expr) \uparrow & \text{otherwise} \end{cases}$$

In this definition, *expr* is some While++ expression or list.

For the expression translation functions, we fix the following set of function symbols:

$$\Sigma_{theory} = \left\{ \begin{array}{ll} n : int\ (n \in \mathbb{Z}), & or : [bool \times bool] \Rightarrow bool, \\ + : [int \times int] \Rightarrow int, & =_{bool} : [bool \times bool] \Rightarrow bool, \\ - : [int \times int] \Rightarrow int, & =_{int} : [int \times int] \Rightarrow bool, \\ * : [int \times int] \Rightarrow int, & =_{string} : [string \times string] \Rightarrow bool, \\ / : [int \times int] \Rightarrow int, & > : [int \times int] \Rightarrow bool, \\ \% : [int \times int] \Rightarrow int, & < : [int \times int] \Rightarrow bool, \\ true : bool, & \geq : [int \times int] \Rightarrow bool, \\ false : bool, & \leq : [int \times int] \Rightarrow bool \\ not : [bool] \Rightarrow bool & s : string\ (s \in \mathbb{S}) \\ and : [bool \times bool] \Rightarrow bool, & \end{array} \right\}$$

5.2.1 Translation of the arithmetic expressions

To describe the arithmetic expressions' translation into the LCTRS terms we define the function:

$$\mathcal{AT} : \mathbf{Aexp} \rightarrow \mathcal{T}erms(\Sigma_{theory} \cup \mathcal{V})$$

$\mathcal{AT}(\texttt{n}) = n\ (n \in \mathbb{Z})$
$\mathcal{AT}(\texttt{x} \in \mathbf{Var}_{int}) = \mathcal{VT}(\texttt{x})$
$\mathcal{AT}(aexp_1\ \texttt{+}\ aexp_2) = +(\mathcal{AT}(aexp_1), \mathcal{AT}(aexp_2))$
$\mathcal{AT}(aexp_1\ \texttt{-}\ aexp_2) = -(\mathcal{AT}(aexp_1), \mathcal{AT}(aexp_2))$
$\mathcal{AT}(aexp_1\ \texttt{*}\ aexp_2) = *(\mathcal{AT}(aexp_1), \mathcal{AT}(aexp_2))$
$\mathcal{AT}(aexp_1\ \texttt{/}\ aexp_2) = /(\mathcal{AT}(aexp_1), \mathcal{AT}(aexp_2))$
$\mathcal{AT}(aexp_1\ \texttt{\%}\ aexp_2) = \%(\mathcal{AT}(aexp_1), \mathcal{AT}(aexp_2))$
$\mathcal{AT}(\texttt{(}aexp\texttt{)}) = \mathcal{AT}(aexp)$

5.2.2 Translation of the boolean expressions

To describe the boolean expressions' translation into the LCTRS terms we define the function:

$$\mathcal{BT} : \mathbf{Bexp} \rightarrow \mathcal{T}erms(\Sigma_{theory} \cup \mathcal{V})$$

$\mathcal{BT}(\texttt{true}) = true$
$\mathcal{BT}(\texttt{false}) = false$
$\mathcal{BT}(\texttt{x} \in \mathbf{Var}_{bool}) = \mathcal{VT}(\texttt{x})$
$\mathcal{BT}(\texttt{not}\ bexp) = not(\mathcal{BT}(bexp))$
$\mathcal{BT}(bexp_1\ \texttt{and}\ bexp_2) = and(\mathcal{BT}(bexp_1), \mathcal{BT}(bexp_2))$
$\mathcal{BT}(bexp_1\ \texttt{or}\ bexp_2) = or(\mathcal{BT}(bexp_1), \mathcal{BT}(bexp_2))$
$\mathcal{BT}(bexp_1\ \texttt{=}\ bexp_2) = =_{bool}(\mathcal{BT}(bexp_1), \mathcal{BT}(bexp_2))$
$\mathcal{BT}(aexp_1\ \texttt{=}\ aexp_2) = =_{int}(\mathcal{AT}(aexp_1), \mathcal{AT}(aexp_2))$
$\mathcal{BT}(strexp_1\ \texttt{=}\ strexp_2) = =_{string}(\mathcal{S}tr\mathcal{T}(strexp_1), \mathcal{S}tr\mathcal{T}(strexp_2))$
$\mathcal{BT}(aexp_1\ \texttt{>}\ aexp_2) = >(\mathcal{BT}(aexp_1), \mathcal{BT}(aexp_2))$
$\mathcal{BT}(aexp_1\ \texttt{<}\ aexp_2) = <(\mathcal{BT}(aexp_1), \mathcal{BT}(aexp_2))$

$\mathcal{BT}(aexp_1 \texttt{ >= } aexp_2) = \geq(\mathcal{BT}(aexp_1), \mathcal{BT}(aexp_2))$
$\mathcal{BT}(aexp_1 \texttt{ <= } aexp_2) = \leq(\mathcal{BT}(aexp_1), \mathcal{BT}(aexp_2))$
$\mathcal{BT}(\texttt{(}bexp\texttt{)}) = \mathcal{BT}(bexp)$

5.2.3 Translation of the string expressions

To describe the string expressions' translation into the LCTRS terms we define the function:

$$\mathcal{S}tr\mathcal{T} : \textbf{Strexp} \rightarrow \mathcal{T}erms(\Sigma_{theory} \cup \mathcal{V})$$

$\mathcal{S}tr\mathcal{T}(\texttt{"}str\texttt{"}) = "str"$ $(str \in \mathbb{S})$
$\mathcal{S}tr\mathcal{T}(\texttt{x} \in \textbf{Var}_{string}) = \mathcal{VT}(\texttt{x})$

5.3 Translating program statements into LCTRS

In order to define the translation of program statements into LCTRS, at first we need to fix some sets of function symbols, LCTRS rules and interpretation mappings that are necessary for creating the valid LCTRS:

- Σ_{theory} - the set of function symbols (already described in the previous section) which is used to represent different expressions in the While++ programming language in LCTRS.
- Σ_{list} - the set of function symbols used to represent the lists in the LCTRS for the I/O support. It is defined as follows:

$$\Sigma_{list} = \left\{ \begin{array}{l} nil : list, \\ consI : [int \times list] \Rightarrow list, \\ consB : [bool \times list] \Rightarrow list, \\ consS : [string \times list] \Rightarrow list, \\ headI : [list] \Rightarrow int, \\ headB : [list] \Rightarrow bool, \\ headS : [list] \Rightarrow string, \\ tail : [list] \Rightarrow list \end{array} \right\}$$

- $\mathcal{R}_{list}$ - the set of LCTRS rules used to define the operations on lists. It is defined as follows:

$$\mathcal{R}_{list} = \left\{ \begin{array}{l} headI(consI(i, l)) \rightarrow i \ [true], \\ headB(consB(b, l)) \rightarrow b \ [true], \\ headS(consS(str, l)) \rightarrow str \ [true], \\ tail(consI(i, l)) \rightarrow l \ [true], \\ tail(consB(b, l)) \rightarrow l \ [true], \\ tail(consS(str, l)) \rightarrow l \ [true] \end{array} \right\}$$

  Like in the semantics description for the While++ programming language, we also introduce the generalizations for the terms and rules related to the list construction in LCTRS:

  - $consT : [type \times list] \Rightarrow type$ - generalization for the terms $consI$, $consB$, and $consS$.
  - $headT : [list] \Rightarrow int$ - generalization for the terms $headI$, $headB$, and $headS$.

– $headT(consT(a, l)) \to a\ [true]$ - generalization for the rules for selecting the first element of certain type from the list.

– $tail(consT(a, l)) \to l\ [true]$ - generalization for the rules for selecting the remaining part of the list excluding the first element of certain type.

- $\mathcal{I}$ - the mapping which assigns each sort occurring in Σ_{theory} a set. It is defined as follows:

$$\mathcal{I} = \left\{ \begin{array}{l} int \mapsto \mathbb{Z}, \\ bool \mapsto \mathbb{B}, \\ string \mapsto \mathbb{S} \end{array} \right\}$$

- $\mathcal{J}$ - the interpretation mapping which maps each $f : [\iota_1, \ldots, \iota_n] \Rightarrow \kappa \in \Sigma_{theory}$ to a function $\mathcal{J}_f : \mathcal{I}_{\iota_1} \times \cdots \times \mathcal{I}_{\iota_1} \to \mathcal{I}_\kappa$.

  It is defined as follows:

$$\mathcal{J} = \left\{ \begin{array}{l} (+ : [int \times int] \Rightarrow int) \mapsto (+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}), \\ (- : [int \times int] \Rightarrow int) \mapsto (- : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}), \\ (* : [int \times int] \Rightarrow int) \mapsto (* : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}), \\ (/ : [int \times int] \Rightarrow int) \mapsto (div : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}), \\ (\% : [int \times int] \Rightarrow int) \mapsto (mod : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}), \\ (true : bool) \mapsto (\top \in \mathbb{B}), \\ (false : bool) \mapsto (\bot \in \mathbb{B}), \\ (not : [bool] \Rightarrow bool) \mapsto (\neg : \mathbb{B} \to \mathbb{B}), \\ (and : [bool \times bool] \Rightarrow bool) \mapsto (\wedge : \mathbb{B} \times \mathbb{B} \to \mathbb{B}), \\ (or : [bool \times bool] \Rightarrow bool) \mapsto (\wedge : \mathbb{B} \times \mathbb{B} \to \mathbb{B}), \\ (s : string\ s \in \mathbb{S}) \mapsto (s \in \mathbb{S}), \\ (=_{\texttt{bool}} : [bool \times bool] \Rightarrow bool) \mapsto (= : \mathbb{B} \times \mathbb{B} \to \mathbb{B}), \\ (=_{\texttt{int}} : [int \times int] \Rightarrow bool) \mapsto (= : \mathbb{Z} \times \mathbb{Z} \to \mathbb{B}), \\ (=_{\texttt{string}} : [string \times string] \Rightarrow bool) \mapsto (= : \mathbb{S} \times \mathbb{S} \to \mathbb{B}), \\ (n : int\ (n \in \mathbb{Z})) \mapsto (n \in \mathbb{Z}), \\ (> : [int \times int] \Rightarrow bool) \mapsto (> : \mathbb{Z} \times \mathbb{Z} \to \mathbb{B}), \\ (< : [int \times int] \Rightarrow bool) \mapsto (< : \mathbb{Z} \times \mathbb{Z} \to \mathbb{B}), \\ (\geq : [int \times int] \Rightarrow bool) \mapsto (\geq : \mathbb{Z} \times \mathbb{Z} \to \mathbb{B}), \\ (\leq : [int \times int] \Rightarrow bool) \mapsto (\leq : \mathbb{Z} \times \mathbb{Z} \to \mathbb{B}) \end{array} \right\}$$

  The functions div and mod are defined in the following way:

$$div(a, b) = \begin{cases} 0 & \text{if } b = 0 \\ sign(a \cdot b) \cdot \left\lfloor \frac{|a|}{|b|} \right\rfloor & \text{if } a \geq 0 \\ sign(a \cdot b) \cdot \left\lceil \frac{|a|}{|b|} \right\rceil & \text{otherwise} \end{cases}$$

$$mod(a, b) = \begin{cases} 0 & \text{if } b = 0 \\ |a| - |b| \cdot \left\lfloor \frac{|a|}{|b|} \right\rfloor & \text{if } (a \geq 0) \vee (|a| - |b| \cdot \left\lfloor \frac{|a|}{|b|} \right\rfloor = 0) \\ |b| - (|a| - |b| \cdot \left\lfloor \frac{|a|}{|b|} \right\rfloor) & \text{otherwise} \end{cases}$$

- Additionally, we define sort and interpretation mappings for the list representation in LCTRS:

$$\mathcal{I}_{list} = \{list \mapsto \mathbb{L}\}$$

$$\mathcal{J}_{list} = \begin{cases} (nil : list) \mapsto ([\,] \in \mathbb{L}) \\ (consI : [int \times list] \Rightarrow list) \mapsto (\mathbf{consI} : \mathbb{Z} \times \mathbb{L} \to \mathbb{L}), \\ (consB : [bool \times list] \Rightarrow list) \mapsto (\mathbf{consB} : \mathbb{B} \times \mathbb{L} \to \mathbb{L}), \\ (consI : [string \times list] \Rightarrow list) \mapsto (\mathbf{consS} : \mathbb{S} \times \mathbb{L} \to \mathbb{L}), \\ (headI : [list] \Rightarrow int) \mapsto (\mathbf{headI} : \mathbb{L} \to \mathbb{Z}), \\ (headB : [list] \Rightarrow int) \mapsto (\mathbf{headB} : \mathbb{L} \to \mathbb{B}), \\ (headS : [list] \Rightarrow int) \mapsto (\mathbf{headS} : \mathbb{L} \to \mathbb{S}), \\ (tail : [list] \Rightarrow list) \mapsto (\mathbf{tail} : \mathbb{L} \to \mathbb{L}) \end{cases}$$

5.3.1 Translation function $\mathcal{T}$

The translation function $\mathcal{T}$ is a specific function that allows the translation of an arbitrary While++ statement into the set of the LCTRS rules:

$$\mathcal{T}(S, \vec{x}, id) = ((\Sigma_S, \mathcal{R}_S), \vec{x'}, id')$$

The arguments of this function are:

1. $S \in \mathcal{S}tm$ - some valid While++ program statement.
2. $\vec{x} \in \overrightarrow{\mathbf{Var}}$ - some vector of variables.
3. $id \in \mathbb{N}$ - some natural number used to generate identifiers for the LCTRS terms.

And it returns the tuple, which consists of the following:

1. Σ_S - the set of all function symbols generated by translating the statement S.
2. $\mathcal{R}_S$ - the set of all LCTRS rules generated by translating the statement S.
3. $\vec{x'}$ - the resulting vectors of While++ variables which we obtain after processing the statement S. The new variables can be added because of the variable declarations in the statement S.
4. id' - the identifier of the last generated LCTRS rule term (on the right). We later will use this id' to chain the rules in one execution flow.

Also, we introduce the special double bracket notation of the function $\mathcal{T}$:

$$\mathcal{T}(S, \vec{x}, id) = [\![S]\!]_{\vec{x},id} = ((\Sigma_S, \mathcal{R}_S), \vec{x'}, id')$$

This way the translation rules will become more readable.

Consider the application of the translation function $\mathcal{T}$ on some While++ statement S, some starting identifier id and some initial vector of variables $\vec{x}$:

$$[\![S]\!]_{\vec{x},id} = ((\Sigma_S, \mathcal{R}_S), \vec{x'}, id')$$

Using the output of this function we can construct a valid LCTRS which will simulate the execution of the While++ program statement S:

$$\begin{pmatrix} \Sigma_{theory} \cup \Sigma_{list} \cup \Sigma_S, \\ \mathcal{R}_{list} \cup \mathcal{R}_S, \\ \mathcal{I}, \\ \mathcal{J} \end{pmatrix}$$

5.3.2 Special notation for the individual LCTRS rules

Because the translation function $\mathcal{T}$ is defined for arbitrary While++ statement, we don't know what kind of variables the terms of the rules will be in use. Because of this, we will introduce the following notation for the terms:

$$stm_{id}[\vec{v}]$$

This means that all the arguments of the function symbol $stm_{id}(\dots)$ are the variables from the vector $\vec{v}$. So, the typical notation of the individual LCTRS rules in the translation rules will look like:

$$stm_{id}[\vec{v}] \to_{\mathcal{R}} stm_{id'}[\vec{v'}]\ [\varphi]$$

Where:

1. id and id' are the function symbols identifiers,
2. $\vec{v}$ is a variable vector,
3. $\vec{v'}$ is a vector of terms constructed from variables and function symbols $f \in \Sigma_{theory} \cup \Sigma_{list}$.
4. φ is the boolean condition of the LCTRS rule (which may include variables from the vector $\vec{v}$).

Additionally, we define the special function *signature*. This function takes as an input a vector of terms from $\mathcal{T}erms(\Sigma \cup \mathcal{V})$ and an output sort and returns the sort declaration for the function symbol. It can be defined as:

$$signature(\vec{t}, \kappa) = ([outputSort(t_1), \dots, outputSort(t_n)] \Rightarrow \kappa) \text{ if } \vec{t} = (t_1, \dots, t_n)$$

Where the function $outputSort : \mathcal{T}erms(\Sigma \cup \mathcal{V}) \to \mathcal{S}$ is defined as:

$$outputSort(t) = \begin{cases} \kappa & \text{if } t = (f : (\dots) \Rightarrow \kappa), \\ \iota & \text{if } t = (x : \iota) \end{cases}$$

5.3.3 Definition of the translation function $\mathcal{T}$

To define the translation function $\mathcal{T}$ more effectively, we will define the function for all base syntax elements of While++ recursively: the definition for the base components (such as the variable declaration, variable introduction, printing commands, and the skip command) will be given explicitly. The rules for the complex statements (such as `if` statement, `while` statement, and composition of 2 statements) imply using the translation function as a part of the definition of the translation function for these complex statements.

Additionally, in these function definitions, we will be using the proof tree notation to specify all necessary conditions that have to hold for the arguments of the translation function, and all post-conditions that will hold for the the output of this translation function.

To simplify the translation function definition, we give only the generalized definitions for the type-dependent statements. We give the details on the generalizations' constructions specifically for every type-dependent statement.

$\mathcal{T}$ is defined as:

1. **Skip statement**

$$[\![\texttt{skip}]\!]_{\vec{x},id} = \begin{pmatrix} \begin{pmatrix} \{stm_{id+1} : signature(\mathcal{VT}(\vec{x}), list), \\ stm_{id+2} : signature(\mathcal{VT}(\vec{x}), list)\}, \\ \{stm_{id+1}[\mathcal{VT}(\vec{x})] \rightarrow stm_{id+2}[\mathcal{VT}(\vec{x})]\ [true]\} \end{pmatrix}, \\ \vec{x}, \\ id+2 \end{pmatrix}$$

2. **Variable declaration**

$$\frac{var \notin \vec{x}}{[\![type\ var\ \texttt{:=}\ expr]\!]_{\vec{x},id} = \begin{pmatrix} \begin{pmatrix} \{stm_{id+1} : signature(\mathcal{VT}(\vec{x}), list), \\ stm_{id+2} : signature(\mathcal{VT}(\vec{x}) \cdot [\mathcal{ET}(expr) : type], list)\}, \\ \{stm_{id+1}[\mathcal{VT}(\vec{x})] \rightarrow stm_{id+2}[\mathcal{VT}(\vec{x}) \cdot [\mathcal{ET}(expr) : type]]\ [true]\} \end{pmatrix}, \\ \vec{x} \cdot [var : type], \\ id+2 \end{pmatrix}}$$

In the specific instances of this generalized definition, the type *type* and the expression $expr \in$ **Expr** need to be replaced accordingly by:

(a) *int*, and $aexp \in$ **Aexp** - for integer variables

(b) *bool*, and $bexp \in$ **Bexp** - for boolean variables

(c) *string*, and $strexp \in$ **Strexp** - for string variables

3. **Variable assignment**

$$\frac{\begin{array}{l}(|\vec{x}| = |\vec{w}|) \wedge \\ \exists p \in \{0, |\vec{x}| - 1\} : \\ \quad \begin{pmatrix} ((x_p = var_{type}) \wedge (w_p = \mathcal{ET}(expr))) \quad \wedge \\ (\forall i \in \{0, |\vec{x}| - 1\} : (i \neq p) \implies (w_i = \mathcal{ET}(x_i))) \end{pmatrix} \end{array}}{[\![var_{type}\ \texttt{:=}\ expr]\!]_{\vec{x},id} = \begin{pmatrix} \begin{pmatrix} \{stm_{id+1} : signature(\mathcal{VT}(\vec{x}), list), \\ stm_{id+2} : signature(\mathcal{VT}(\vec{x}), list)\}, \\ \{stm_{id+1}[\mathcal{VT}(\vec{x})] \rightarrow stm_{id+2}[\vec{w}]\ [true]\} \end{pmatrix}, \vec{x}, id+2 \end{pmatrix}}$$

In the specific instances of this generalized definition, the type *type* needs to be replaced accordingly by:

(a) *int* - for integer variables

(b) *bool* - for boolean variables

(c) *string* - for string variables

4. **Print statement**

$$\frac{\begin{array}{l}(|\vec{x}| = |\vec{w}|) \wedge \\ \exists p \in \{0, |\vec{x}| - 1\} : \\ \quad \begin{pmatrix} (x_p = \texttt{Out}) \wedge (w_p = consT(\mathcal{ET}(expr), \mathcal{VT}(\texttt{Out}))) \wedge \\ (\forall i \in \{0, |\vec{x}| - 1\} : (i \neq p) \implies (w_i = \mathcal{ET}(x_i))) \end{pmatrix} \end{array}}{[\![\texttt{printType(}expr\texttt{)}]\!]_{\vec{x},id} = \begin{pmatrix} \begin{pmatrix} \{stm_{id+1} : signature(\mathcal{VT}(\vec{x}), list), \\ stm_{id+2} : signature(\mathcal{VT}(\vec{x}), list)\}, \\ \{stm_{id+1}[\mathcal{VT}(\vec{x})] \rightarrow stm_{id+2}[\vec{w}]\ [true]\} \end{pmatrix}, \vec{x}, id+2 \end{pmatrix}}$$

In the specific instances of this generalized definition, the expression $expr \in \mathcal{E}xpr$, the command name `printType`, and the function $consT$ need to be replaced accordingly by:

(a) $aexp \in$ **Aexp**, `printInt`, and $consI$ - for arithmetic expressions

(b) $bexp \in$ **Bexp**, `printBool`, and $consB$ - for boolean expressions

(c) $strexp \in$ **Strexp**, `printString`, and $consS$ - for string expressions

5. **Read statement**

$$\frac{\begin{array}{l}(|\vec{x}| = |\vec{w}|) \wedge \\ \exists p_1, p_2 \in \{0, |\vec{x}| - 1\} : \\ \quad \begin{pmatrix}(p_1 \neq p_2) \wedge (x_{p_1} = var) \wedge (x_{p_2} = \texttt{In}) \quad \wedge \\ (w_{p_1} = headT(\mathcal{VT}(\texttt{In}))) \wedge (w_{p_2} = tailT(\mathcal{VT}(\texttt{In}))) \quad \wedge \\ (\forall i \in \{0, |\vec{x}| - 1\} : (i \neq p_1 \wedge i \neq p_2) \implies (w_i = \mathcal{ET}(x_i)))\end{pmatrix}\end{array}}{[\![\texttt{readType(}var\texttt{)}]\!]_{\vec{x},id} = \left(\begin{pmatrix}\{stm_{id+1} : signature(\mathcal{VT}(\vec{x}), list), \\ stm_{id+2} : signature(\mathcal{VT}(\vec{x}), list)\}, \\ \{stm_{id+1}[\mathcal{VT}(\vec{x})] \to stm_{id+2}[\vec{w}]\ [true]\}\end{pmatrix}, \vec{x}, id + 2\right)}$$

In the specific instances of this generalized definition the type *type*, the command name `readType`, the function $headT$, and the function $tailT$ need to be replaced accordingly by:

(a) *int*, `readInt`, $headI$, and $tailI$ - for integer variables

(b) *bool*, `readBool`, $headB$, and $tailB$ - for boolean variables

(c) *string*, `readString`, $headS$, and $tailS$ - for string variables

6. **Composition of statements**

$$\frac{[\![S_1]\!]_{\vec{x},id} = \left(\begin{pmatrix}\Sigma_{S_1}, \\ \mathcal{R}_{S_1}\end{pmatrix}, \vec{x} \cdot \vec{y}, id'\right) \qquad [\![S_2]\!]_{\vec{x}\cdot\vec{y},id'} = \left(\begin{pmatrix}\Sigma_{S_2}, \\ \mathcal{R}_{S_2}\end{pmatrix}, \vec{x} \cdot \vec{y} \cdot \vec{z}, id''\right)}{[\![S_1\texttt{ ; } S_2]\!]_{\vec{x},id} = \left(\begin{pmatrix}\Sigma_{S_1} \cup \Sigma_{S_2}, \\ \mathcal{R}_{S_1} \cup \\ \{stm_{id'}[\mathcal{VT}(\vec{x} \cdot \vec{y})] \to stm_{id'+1}[\mathcal{VT}(\vec{x} \cdot \vec{y})]\ [true]\} \cup \\ \mathcal{R}_{S_2}\end{pmatrix}, \vec{x} \cdot \vec{y} \cdot \vec{z}, id''\right)}$$

7. **If statement**

$$\frac{[\![S_1]\!]_{\vec{x},id+1} = \left(\begin{pmatrix}\Sigma_{S_1}, \\ \mathcal{R}_{S_1}\end{pmatrix}, \vec{x} \cdot \vec{y}, id'\right) \qquad [\![S_2]\!]_{\vec{x},id'} = \left(\begin{pmatrix}\Sigma_{S_2}, \\ \mathcal{R}_{S_2}\end{pmatrix}, \vec{x} \cdot \vec{z}, id''\right)}{[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]_{\vec{x},id} = \left(\begin{pmatrix}\Sigma_{S_1} \cup \Sigma_{S_2} \cup \\ \{stm_{id+1} : signature(\mathcal{VT}(\vec{x}), list), \\ stm_{id''+1} : signature(\mathcal{VT}(\vec{x}), list)\}, \\ \{stm_{id+1}[\mathcal{VT}(\vec{x})] \to stm_{id+2}[\mathcal{VT}(\vec{x})]\ [\mathcal{BT}(b)]\} \cup \mathcal{R}_{S_1} \cup \\ \{stm_{id+1}[\mathcal{VT}(\vec{x})] \to stm_{id'+1}[\mathcal{VT}(\vec{x})]\ [not(\mathcal{BT}(b))]\} \cup \mathcal{R}_{S_2} \cup \\ \{stm_{id'}[\mathcal{VT}(\vec{x} \cdot \vec{y})] \to stm_{id''+1}[\mathcal{VT}(\vec{x})]\ [true], \\ stm_{id''}[\mathcal{VT}(\vec{x} \cdot \vec{z})] \to stm_{id''+1}[\mathcal{VT}(\vec{x})]\ [true]\}\end{pmatrix}, \vec{x}, id'' + 1\right)}$$

8. **While statement**

$$[\![S]\!]_{\vec{x},id+1} = \left(\begin{pmatrix} \Sigma_S, \\ \mathcal{R}_S \end{pmatrix}, \vec{x} \cdot \vec{y}, id' \right)$$

$$[\![\texttt{while } b \texttt{ do } S]\!]_{\vec{x},id} =$$

$$\left(\begin{pmatrix} \Sigma_S \cup \\ \{stm_{id+1} : signature(\mathcal{VT}(\vec{x}), list), \\ stm_{id'+1} : signature(\mathcal{VT}(\vec{x}), list)\}, \\ \{stm_{id+1}[\mathcal{VT}(\vec{x})] \rightarrow stm_{id+2}[\mathcal{VT}(\vec{x})]\ [\mathcal{BT}(b)]\} \cup \mathcal{R}_S \cup \\ \{stm_{id'}[\mathcal{VT}(\vec{x} \cdot \vec{y})] \rightarrow stm_{id+1}[\mathcal{VT}(\vec{x})]\ [true] \\ stm_{id+1}[\mathcal{VT}(\vec{x})] \rightarrow stm_{id'+1}[\mathcal{VT}(\vec{x})]\ [not(\mathcal{BT}(b))]\} \end{pmatrix}, \vec{x}, id' + 1 \right)$$

5.3.4 Example translations of While++ programs into LCTRS

In the following examples, we show the translations of different While++ programs into LCTRSs. At the end of each example, we show the example reduction of some initial term using the LCTRS constructed using the translation function.

Example 42.

$$\mathcal{T}\left((\texttt{skip; skip; skip}), [\texttt{In} \in \mathbf{Var}_{list}, \texttt{Out} \in \mathbf{Var}_{list}], 0\right) =$$

$$\left(\begin{Bmatrix} stm_1 : [list \times list] \Rightarrow list, \\ \dots, \\ stm_6 : [list \times list] \Rightarrow list \end{Bmatrix}, \begin{Bmatrix} stm_1(in, out) \to stm_2(in, out)\ [true], \\ stm_2(in, out) \to stm_3(in, out)\ [true], \\ stm_3(in, out) \to stm_4(in, out)\ [true], \\ stm_4(in, out) \to stm_5(in, out)\ [true], \\ stm_5(in, out) \to stm_6(in, out)\ [true] \end{Bmatrix}, [\texttt{In} \in \mathbf{Var}_{list}, \texttt{Out} \in \mathbf{Var}_{list}], 6\right)$$

Reduction example:

$$\begin{aligned} &stm_1(nil, nil) \to stm_2(nil, nil) \\ &\to stm_3(nil, nil) \to stm_4(nil, nil) \\ &\to stm_5(nil, nil) \to stm_6(nil, nil) \end{aligned}$$

Example 43.

$$\mathcal{T}\left(\begin{pmatrix}\texttt{int x := 1;}\\ \texttt{int y := 2;}\\ \texttt{bool geq := (x >= y);}\\ \texttt{printBool(geq)}\end{pmatrix}, [\texttt{In} \in \mathbf{Var}_{list}, \texttt{Out} \in \mathbf{Var}_{list}], 0\right) =$$

$$\left(\begin{Bmatrix} stm_1 : [list \times list] \Rightarrow list, stm_2 : [list \times list \times int] \Rightarrow list, \\ stm_3 : [list \times list \times int] \Rightarrow list, stm_4 : [list \times list \times int \times int] \Rightarrow list, \\ stm_5 : [list \times list \times int \times int] \Rightarrow list, \\ stm_6 : [list \times list \times int \times int \times bool] \Rightarrow list, \\ stm_7 : [list \times list \times int \times int \times bool] \Rightarrow list, \\ stm_8 : [list \times list \times int \times int \times bool] \Rightarrow list \end{Bmatrix}, \right.$$
$$\left. \begin{Bmatrix} stm_1(in, out) \to stm_2(in, out, 1)\ [true], \\ stm_2(in, out, x) \to stm_3(in, out, x)\ [true], \\ stm_3(in, out, x) \to stm_4(in, out, x, 2)\ [true], \\ stm_4(in, out, x, y) \to stm_5(in, out, x, y)\ [true], \\ stm_5(in, out, x, y) \to stm_6(in, out, x, y, \geq(x, y))\ [true], \\ stm_6(in, out, x, y, geq) \to stm_7(in, out, x, y, geq)\ [true], \\ stm_7(in, out, x, y, geq) \to stm_8(in, consB(geq, out), x, y, geq)\ [true] \end{Bmatrix}, \right.$$
$$\left. [\texttt{In} \in \mathbf{Var}_{list}, \texttt{Out} \in \mathbf{Var}_{list}, \texttt{x} \in \mathbf{Var}_{int}, \texttt{y} \in \mathbf{Var}_{int}, \texttt{geq} \in \mathbf{Var}_{bool}], 8 \right)$$

Reduction example:

$$stm_1(nil, nil) \to stm_2(nil, nil, 1) \to stm_3(nil, nil, 1) \to stm_4(nil, nil, 1, 2)$$
$$\to stm_5(nil, nil, 1, 2) \to stm_6(nil, nil, 1, 2, \geq(1, 2))$$
$$\to stm_6(nil, nil, 1, 2, false) \to stm_7(nil, nil, 1, 2, false)$$
$$\to stm_8(nil, consB(false, nil), 1, 2, false)$$

Example 44.

$$\mathcal{T}\left(\left(\begin{array}{l}\texttt{int x := 0;}\\ \texttt{printString("Enter x: ");}\\ \texttt{readInt(x);}\\ \texttt{if x > 10 then}\\ \texttt{ printString("x > 10")}\\ \texttt{else}\\ \texttt{ printString("x <= 10")}\end{array}\right), [\texttt{In} \in \mathbf{Var}_{list}, \texttt{Out} \in \mathbf{Var}_{list}], 0\right) =$$

$$\left(\left\{\begin{array}{l} stm_1 : [list \times list] \Rightarrow list,\\ stm_2 : [list \times list \times int] \Rightarrow list,\\ \ldots,\\ stm_{12} : [list \times list \times int] \Rightarrow list\end{array}\right\},\right.$$

$$\left\{\begin{array}{l} stm_1(in, out) \to stm_2(in, out, 0)\ [true],\\ stm_2(in, out, x) \to stm_3(in, out, x)\ [true],\\ stm_3(in, out, x) \to stm_4(in, consS(\text{"Enter x: "}, out), x)\ [true],\\ stm_4(in, out, x) \to stm_5(in, out, x)\ [true],\\ stm_5(in, out, x) \to stm_6(tailI(in), out, headI(in))\ [true],\\ stm_6(in, out, x) \to stm_7(in, out, x)\ [true],\\ stm_7(in, out, x) \to stm_8(in, out, x)\ [>(x, 10)],\\ stm_8(in, out, x) \to stm_9(in, consS(\text{"x >10"}, out), x)\ [true]\\ stm_7(in, out, x) \to stm_{10}(in, out, x)\ [not(>(x, 10))],\\ stm_{10}(in, out, x) \to stm_{11}(in, consS(\text{"x <= 10"}, out), x)\ [true],\\ stm_9(in, out, x) \to stm_{12}(in, out, x)\ [true],\\ stm_{11}(in, out, x) \to stm_{12}(in, out, x)\ [true]\end{array}\right\},$$

$$\left.[\texttt{In} \in \mathbf{Var}_{list}, \texttt{Out} \in \mathbf{Var}_{list}, \texttt{x} \in \mathbf{Var}_{int}], 12\right)$$

Reduction example:

$$stm_1(consI(8, nil), nil) \to stm_2(consI(8, nil), nil, 0)$$
$$\to stm_3(consI(8, nil), nil, 0) \to stm_4(consI(8, nil), consS(\text{"Enter x: "}, nil), 0)$$
$$\to stm_5(consI(8, nil), consS(\text{"Enter x: "}, nil), 0)$$
$$\to stm_6(tailI(consI(8, nil)), consS(\text{"Enter x: "}, nil), headI(consI(8, nil)))$$
$$\to stm_6(nil, consS(\text{"Enter x: "}, nil), headI(consI(8, nil)))$$
$$\to stm_6(nil, consS(\text{"Enter x: "}, nil), 8)$$
$$\to stm_7(nil, consS(\text{"Enter x: "}, nil), 8)$$
$$\xrightarrow[[\![not(>(8,10))]\!]=\top]{\text{because:}} stm_{10}(nil, consS(\text{"Enter x: "}, nil), 8)$$
$$\to stm_{11}(nil, consS(\text{"x <= 10"}, consS(\text{"Enter x: "}, nil)), 8)$$
$$\to stm_{12}(nil, consS(\text{"x <= 10"}, consS(\text{"Enter x: "}, nil)), 8)$$

Example 45.

$$\mathcal{T}\left(\left(\begin{array}{l}\texttt{int x := 0;}\\ \texttt{int i := 1;}\\ \texttt{readInt(x);}\\ \texttt{while not(x = 1) do (}\\ \texttt{ printInt(i * i);}\\ \texttt{ i := i + 1;}\\ \texttt{ readInt(x)}\\ \texttt{)}\end{array}\right), [\texttt{In} \in \mathbf{Var}_{list}, \texttt{Out} \in \mathbf{Var}_{list}], 0\right) =$$

$$\left(\left\{\begin{array}{l} stm_1 : [list \times list] \Rightarrow list,\\ stm_2 : [list \times list \times int] \Rightarrow list,\\ stm_3 : [list \times list \times int] \Rightarrow list,\\ stm_4 : [list \times list \times int \times int] \Rightarrow list,\\ \ldots,\\ stm_{14} : [list \times list \times int \times int] \Rightarrow list\end{array}\right\},\right.$$

$$\left\{\begin{array}{l} stm_1(in, out) \rightarrow stm_2(in, out, 0)\ [true],\\ stm_2(in, out, x) \rightarrow stm_3(in, out, x)\ [true],\\ stm_3(in, out, x) \rightarrow stm_4(in, out, x, 1)\ [true],\\ stm_4(in, out, x, i) \rightarrow stm_5(in, out, x, i)\ [true],\\ stm_5(in, out, x, i) \rightarrow stm_6(tailI(in), out, headI(in), i)\ [true],\\ stm_6(in, out, x, i) \rightarrow stm_7(in, out, x, i)\ [true],\\ stm_7(in, out, x, i) \rightarrow stm_8(in, out, x, i)\ [not(=_{int}(x, 1))],\\ stm_8(in, out, x, i) \rightarrow stm_9(in, consI(*(i, i), out), x, i)\ [true],\\ stm_9(in, out, x, i) \rightarrow stm_{10}(in, out, x, i)\ [true],\\ stm_{10}(in, out, x, i) \rightarrow stm_{11}(in, out, x, (i + 1));[true],\\ stm_{11}(in, out, x, i) \rightarrow stm_{12}(in, out, x, i)\ [true],\\ stm_{12}(in, out, x, i) \rightarrow stm_{13}(tailI(in), out, headI(in), i)\ [true],\\ stm_{13}(in, out, x, i) \rightarrow stm_7(in, out, x, i)\ [true],\\ stm_7(in, out, x, i) \rightarrow stm_{14}(in, out, x, i)\ [not(not(=_{int}(x, 1)))]\end{array}\right\},$$

$$\left.[\texttt{In} \in \mathbf{Var}_{list}, \texttt{Out} \in \mathbf{Var}_{list}, \texttt{x} \in \mathbf{Var}_{int}, \texttt{i} \in \mathbf{Var}_{int}], 14\right)$$

Reduction example:

$$\begin{aligned}
&stm_1(consI(4, consI(3, consI(2, consI(1, nil)))), nil)\\
&\rightarrow stm_2(consI(4, consI(3, consI(2, consI(1, nil)))), nil, 0)\\
&\rightarrow stm_3(consI(4, consI(3, consI(2, consI(1, nil)))), nil, 0)\\
&\rightarrow stm_4(consI(4, consI(3, consI(2, consI(1, nil)))), nil, 0, 1)\\
&\rightarrow stm_5(consI(4, consI(3, consI(2, consI(1, nil)))), nil, 0, 1)\\
&\rightarrow stm_6(tailI(consI(4, consI(3, consI(2, consI(1, nil))))), nil,\\
&\qquad headI(consI(4, consI(3, consI(2, consI(1, nil))))), 1)\\
&\rightarrow stm_6(consI(3, consI(2, consI(1, nil))), nil,\\
&\qquad headI(consI(4, consI(3, consI(2, consI(1, nil))))), 1)\\
&\rightarrow stm_6(consI(3, consI(2, consI(1, nil))), nil, 4, 1)\\
&\rightarrow stm_7(consI(3, consI(2, consI(1, nil))), nil, 4, 1)
\end{aligned}$$

$\xrightarrow[[\![not(=_{int}(4,1))]\!]=\top]{\text{because:}} stm_8(consI(3, consI(2, consI(1, nil))), nil, 4, 1)$

$\to stm_9(consI(3, consI(2, consI(1, nil))), consI(*(1, 1), nil), 4, 1)$

$\to stm_9(consI(3, consI(2, consI(1, nil))), consI(1, nil), 4, 1)$

$\to stm_{10}(consI(3, consI(2, consI(1, nil))), consI(1, nil), 4, 1)$

$\to stm_{11}(consI(3, consI(2, consI(1, nil))), consI(1, nil), 4, +(1, 1))$

$\to stm_{11}(consI(3, consI(2, consI(1, nil))), consI(1, nil), 4, 2)$

$\to stm_{12}(consI(3, consI(2, consI(1, nil))), consI(1, nil), 4, 2)$

$\to stm_{13}(tailI(consI(3, consI(2, consI(1, nil)))), consI(1, nil),$
$headI(consI(3, consI(2, consI(1, nil)))), 2)$

$\to stm_{13}(consI(2, consI(1, nil)), consI(1, nil),$
$headI(consI(3, consI(2, consI(1, nil)))), 2)$

$\to stm_{13}(consI(2, consI(1, nil)), consI(1, nil), 3, 2)$

$\to stm_7(consI(2, consI(1, nil)), consI(1, nil), 3, 2)$

$\xrightarrow[[\![not(=_{int}(3,1))]\!]=\top]{\text{because:}} stm_8(consI(2, consI(1, nil)), consI(1, nil), 3, 2)$

$\to^* stm_7(consI(1, nil), consI(4, consI(1, nil)), 2, 3)$

$\xrightarrow[[\![not(=_{int}(2,1))]\!]=\top]{\text{because:}} stm_8(consI(1, nil), consI(4, consI(1, nil)), 2, 3)$

$\to^* stm_7(nil, consI(9, consI(4, consI(1, nil))), 1, 4)$

$\xrightarrow[[\![not(not(=_{int}(1,1)))]\!]=\top]{\text{because:}} stm_{14}(nil, consI(9, consI(4, consI(1, nil))), 1, 4)$

Chapter 6

Correctness of the translation

In this chapter, we will analyze the developed translation function $\mathcal{T}$ by formulating and proving specific properties about the function $\mathcal{T}$ and the produced LCTRSs. We will formulate the definition of equivalence of the While++ programs and the corresponding LCTRS generated using the translation function $\mathcal{T}$. We will use this definition to formulate the **Correctness** theorem for the translation function $\mathcal{T}$. Finally, we will prove this theorem by splitting it into two parts: **soundness** and **completeness** of the translation.

6.1 Preliminaries

In this section, we will list the definitions and theorems we will use in further parts of this chapter.

Definition 6.1.1 (Termination of the statement execution)**.** We say that the execution of the statement S on a state s is terminating if and only if there exists a state s' such that $\langle S, s\rangle \to s'$.

Definition 6.1.2 (Normal form. Based on [11, Definition 2.1.1 on p. 12])**.** Consider the LCTRS $(\mathcal{T}erms(\Sigma, \mathcal{V}), \to_{\mathcal{R}})$. An element $t \in \mathcal{T}erms(\Sigma)$ is a normal form if there is no element $s \in \mathcal{T}erms(\Sigma)$ with $s \to_{\mathcal{R}} t$. The set $NF(\to_{\mathcal{R}})$ contains all normal forms of the LCTRS $(\mathcal{T}erms(\Sigma, \mathcal{V}), \to_{\mathcal{R}})$.

Definition 6.1.3 ([12, Definition 2.1.1 on p. 9])**.** y is a normal form of x iff $x \xrightarrow{*} y$ and y is a normal form. If x has a uniquely determined normal form, the latter is denoted by $x\!\downarrow$.

Definition 6.1.4 (Critical pairs. [13, Definition 1 on p. 15])**.** Given rules $\rho_1 \equiv l_1 \to r_1\ [\varphi_1]$, and $\rho_2 \equiv l_2 \to r_2\ [\varphi_2]$ with distinct variables, the critical pairs of ρ_1, ρ_2 are all tuples $\langle s, t, \varphi\rangle$ where:

- l_1 can be written as $C[l_1']$, where l_1' is not a variable, but is unifiable with l_2 (there exists a substitution γ such that $l_1'\gamma = l_2\gamma$);
- $C \neq \Box$, or not $\rho_1 = \rho_2$ modulo renaming of variables, or $Var(r_1) \not\subseteq Var(l_1)$;
- The most general unifier γ of l_1' and l_2 respects variables of both ρ_1 and ρ_2;
- $\varphi_1\gamma \wedge \varphi_2\gamma$ is satisfiable;
- $s = r_1\gamma$ and $t = (C\gamma)[r_2\gamma]$ and $\varphi = \varphi_1\gamma \wedge \varphi_2\gamma$.

The critical pair for calculations of a rule ρ are all critical pairs of ρ with any "rule" of the form $f(x_1, \ldots, x_n) \to y\ [y = f(\mathbf{x})]$ with $f \in \Sigma_{theory} \setminus \mathcal{V}al$.

Definition 6.1.5 (Trivial critical pairs. [13, Definition 2 on p. 15])**.** A critical pair $\langle s, t, \varphi \rangle$ is trivial if for every substitution γ for which it holds that $\forall x \in Var(\varphi) : \gamma(x) \in \mathcal{V}al$ and $[\![\varphi\gamma]\!] = \top$, we have $s\gamma = t\gamma$.

Definition 6.1.6 (Weak orthogonality. [13, Definition 2 on p. 15])**.**
An LCTRS $(\mathcal{T}erms(\Sigma, \mathcal{V}), \to_{\mathcal{R}})$ is weakly orthogonal if the left-hand side of each rule is linear (no variable occurs more than once), and for every pair $\rho_1, \rho_2 \in \mathcal{R}$: every critical pair between ρ_1 and a variable-renamed copy of ρ_2, and every critical pair of ρ_1 for calculations, is trivial. It is orthogonal if there are no critical pairs.

Definition 6.1.7 (Confluence property. [13, Definition on p. 14])**.** Consider the LCTRS $(\mathcal{T}erms(\Sigma, \mathcal{V}), \to_{\mathcal{R}})$. Confluence is the property that whenever $s \to_{\mathcal{R}}^* t$ and $s \to_{\mathcal{R}}^* q$ holds for some $s, t, q \in \mathcal{T}erms(\Sigma)$, then there is some $w \in \mathcal{T}erms(\Sigma)$, such that $t \to_{\mathcal{R}}^* w$ and $q \to_{\mathcal{R}}^* w$.

Definition 6.1.8 (Unique normal forms with respect to reduction property. Based on [11, Definition 2.2.1 on p. 12])**.** Consider the LCTRS $(\mathcal{T}erms(\Sigma, \mathcal{V}), \to_{\mathcal{R}})$. This LCTRS has unique normal forms with respect to reduction (or $UN^{\to}$) property if no element of $\mathcal{T}erms(\Sigma)$ reduces to different normal forms: $\forall a, b, c \in \mathcal{T}erms(\Sigma) : ((a \to_{\mathcal{R}}^* b) \wedge (a \to_{\mathcal{R}}^* c) \wedge (b, c \in NF(\to_{\mathcal{R}}))) \implies (a = b)$.

Following the [11, Proposition 2.2.2 on p. 12, and Proposition 2.2.3 on p. 13], confluence implies the uniqueness of the normal forms with respect to reduction.

Theorem 1 ([13, Theorem 4 on p. 15])**.** *A weakly orthogonal LCTRS is confluent.*

That means a weakly orthogonal LCTRS has unique normal forms with respect to reduction.

Lemma 1 (Weak orthogonality of the LCTRS constructed using the translation function $\mathcal{T}$.)**.** *Consider the translation of the While++ program statement S with the starting vector of unique variables $\vec{x}$ and the starting identifier id:*

$$\mathcal{T}(S, \vec{x}, id) = ((\Sigma_S, \mathcal{R}_S), \vec{x'}, id')$$

The LCTRS $(\Sigma_{theory} \cup \Sigma_{list} \cup \Sigma_S, \mathcal{R}_{list} \cup \mathcal{R}_S, \mathcal{I}, \mathcal{J})$ is weakly orthogonal.

Proof. See the proof in the Appendix A. □

6.2 Correctness theorem

To formulate the correctness theorem, we first need to define what it means for the While++ program to be equivalent to the LCTRS generated by the translation function $\mathcal{T}$. In section 4.2, we defined the natural semantics of the While++ programming language. We will base the equivalence of While++ programs and LCTRSs on the equivalence of the natural semantics derivation trees and the LCTRS reduction sequences.

In natural semantics, program execution is represented using program states, statements, and inference rules. We need to find counterparts for natural semantic states and inference rules in Logically Constrained TRS to effectively define the equivalence of derivation trees and the reduction sequences.

First, we start with the natural semantics states. In While++ natural semantics, we define the program state as a partial function, which is effectively a mapping from the set of While++ variables $\mathbf{Var} \cup \mathbf{Var}_{list}$ to the set of values $\mathbf{Val} \cup \mathbb{L} = \mathbb{Z} \cup \mathbb{B} \cup \mathbb{S} \cup \mathbb{L}$. The naive yet effective approach to storing variable values in the LCTRS is using the function symbols' arguments. Logically Constrained Term Rewriting is an extension of Many-Sorted Term Rewriting. That means the function symbols of the LCTRS can effectively store values of different types.

To be able to compare the representation of values in the While++ natural semantics and the representation of values and expressions in the LCTRS, we need to define some interpretation functions. Since we want LCTRS terms to be able to represent the natural semantics states of While++, we expect that the LCTRS function symbols will not only store terms from the set $\mathcal{T}erms(\Sigma_{theory})$. To fully represent the natural semantics states, the LCTRS function symbols, together with regular values, should store lists In and Out representing the standard input and output. For that we define a new sort mapping function $\mathcal{I}_{combi}$ and a new interpretation mapping $\mathcal{J}_{combi}$:

- $\mathcal{I}_{combi} = \mathcal{I} \cup \mathcal{I}_{list}$
- $\mathcal{J}_{combi} = \mathcal{J} \cup \mathcal{J}_{list}$
  This function maps each $f : [\iota_1 \times \cdots \times \iota_n] \implies \kappa \in \Sigma_{theory} \cup \Sigma_{list}$ to a function $(\mathcal{J}_{combi})_f : (\mathcal{J}_{combi})_{\iota_1} \times \ldots (\mathcal{J}_{combi})_{\iota_n} \to (\mathcal{J}_{combi})_\kappa$.

We also extend the interpretation mapping $\mathcal{J}_{combi}$ to an interpretation on the ground terms in $\mathcal{T}erms(\Sigma_{theory} \cup \Sigma_{list} \cup \mathcal{V})$:

$$[\![f(s_1, \ldots, s_n)]\!]_{combi} = (\mathcal{J}_{combi})_f([\![s_1]\!]_{combi}, \ldots, [\![s_n]\!]_{combi})$$

This extended interpretation can be used to compare representations of values in the While++ program states and values and expressions in LCTRS.

Example 46. Consider the state:

$$s = \{\texttt{In} \mapsto [1, 2, 3], \texttt{Out} \mapsto [\text{"}Hello\text{"}], \texttt{x} \mapsto 10, y \mapsto \top\}$$

We can construct an LCTRS function symbol that will store representations of all values from the state s:

$$stm_1(consI(1, consI(2, consI(3, nil))) : list,$$
$$consS(\text{"}Hello\text{"}, nil) : list, 10 : int, true : bool)$$

Using the special interpretation function $\mathcal{J}$, we can easily convert the Term-Rewriting representation of the values to the representation consistent with the natural semantics:

$$[\![consI(1, consI(2, consI(3, nil)))]\!]_{combi} = [1, 2, 3]$$
$$[\![true]\!]_{combi} = \top$$

This way, a function symbol with multiple arguments of different types can almost simulate the behavior of the natural semantics state. However, there is still a difference: the pure function symbols can't hold any information about the correspondence between the variables and values stored in the function symbol. This problem can be resolved by using a special vector of variables. This vector will work like an index: the position of a certain variable in the variable vector will define the position of the value of this variable in the variable vector.

Example 47. Consider the state:

$$s = \{\texttt{In} \mapsto [1,2,3], \texttt{Out} \mapsto [\text{"}Hello\text{"}], \texttt{a} \mapsto 10, \texttt{b} \mapsto \top\}$$

We can construct an LCTRS term which will store all values from the state s:

$$stm_1[\vec{v}]$$

Where: $\vec{v} = [consI(1, consI(2, consI(3, nil))) : list, consS(\text{"}Hello\text{"}, nil) : list, 10 : int, true : bool]$.

Additionally, we can build the vector of variables: $\vec{x} = [In, Out, a, b]$

Now, if we want to locate the value of the variable b in the function symbol stm_1, we first need to find its position in the vector $\vec{x}$: $x_3 = b$. The argument of the function symbol stm_1 on the position 3 (counting from 0), resembles the value of the variable b in state s:

$$[\![v_3]\!]_{combi} = [\![true]\!]_{combi} = \top = s\ b$$

The described approach allows us to define the equivalence of the program state and the LCTRS term with the variables vector, which simulates that state's behavior.

Suppose we have a state s, which holds values of variables from the vector $\vec{x} \in \overrightarrow{\mathbf{Var}}$. Also, consider the LCTRS term $f[\vec{v}]$, where $\vec{v}$ is a vector of values. We say that the LCTRS term effectively represents the program state s if the following conditions are met: $|\vec{x}| = |Dom(s)| = |\vec{v}|$ and $\forall x_i \in \vec{x} : s\ x_i = [\![v_i]\!]_{combi}$.

Now, let's consider the derivation trees in the natural semantics. A derivation tree is a tree-like structure constructed using inference rules. These trees allow proving facts about the execution of program statements.

Consider the execution of the program S in the state s:

- If a finite derivation tree exists with the final state s', the program is terminating in the state s'.

  $\langle S, s\rangle \to s'$

- If an infinite derivation tree exists, the program is looping.

- Finally, if the derivation tree doesn't exist, we can say there is an error during the program execution.

All listed facts about the program execution can be effectively verified in the LCTRS using reduction chains. Because the natural semantics of While++ is deterministic (if the program is terminating, then there exists only one unique derivation tree), we need to require the LCTRS simulating the program to have a unique normal form with respect to reduction. This unique normal form should be a counterpart of the final state in the derivation tree.

Consider the LCTRS obtained after translating the program S into the LCTRS:

$$\mathcal{T}(S, \vec{x}, 0) = ((\Sigma_S, \mathcal{R}_S), \vec{x'}, id')$$

The constructed LCTRS:

$$(\Sigma_{theory} \cup \Sigma_{list} \cup \Sigma_S, \mathcal{R}_{list} \cup \mathcal{R}_S, \mathcal{I}, \mathcal{J})$$

To have determinism, we need to require the constructed LCTRS to have a unique normal form with respect to reduction rules. To be precise, if we have a reduction of the term to the normal form: $stm_1[\vec{v}] \to^*_{\mathcal{R}_{list} \cup \mathcal{R}_S} stm_{id'}[\vec{v'}]$, this normal form should be a unique normal form with respect to reduction: $stm_{id'}[\vec{v'}] \in NF(\to_{\mathcal{R}_{list} \cup \mathcal{R}_S})$.

- If there exists a finite reduction chain to the unique normal form $stm_{id'}[\vec{v'}]$, then we can say that the LCTRS simulating the program is weakly normalizing, and the vector of values $\vec{v'}$ represents the final state of the program execution.
- If any reduction chain starting from the term $stm_1[\vec{v}]$ is infinite, we can say that the program that LCTRS is simulating is looping.
- And finally, if all reduction chains starting from the term $stm_1[\vec{v}]$ are finite, but none of them end in a term $stm_{id'}[\vec{v'}]$, then we can say that this is an effective representation of a program error in the LCTRS.

If we summarize the facts about the core elements of natural semantics of the While++ programming language and its counterparts in the LCTRS, we can formulate the definition of the equivalence of the While++ programs and the LCTRSs simulating these programs.

Definition 6.2.1 (Equivalence of a While++ statement and the LCTRS produced using the translation function $\mathcal{T}$)**.** Consider the While++ statement S. Also, let's consider the vector of variables $\vec{x} \in \overrightarrow{\mathbf{Var}}$ and some natural number id. Consider the application of the translation function: $\mathcal{T}(S, \vec{x}, id) = ((\Sigma_S, \mathcal{R}_S), \vec{x'}, id')$. The statement S and the LCTRS $(\Sigma_{theory} \cup \Sigma_{list} \cup \Sigma_S, \mathcal{R}_{list} \cup \mathcal{R}_S, \mathcal{I}, \mathcal{J})$ are **equivalent** if the following holds:

$$\begin{aligned}
&\forall s \in \mathcal{S}tate, stm_{id+1}[\vec{v}] \in \mathcal{T}erms(\Sigma_{theory} \cup \Sigma_{list} \cup \Sigma_S) : \\
&\quad \begin{pmatrix} |\vec{x}| = |Dom(s)| = |\vec{v}| \land \\ (\forall x_i \in \vec{x} : x_i \in Dom(s)) \land \\ (\forall i \in \{0, |\vec{v}| - 1\} : v_i \in NF(\to_{\mathcal{R}_{list}})) \land \\ (\forall i \in \{0, |\vec{x}| - 1\} : s\ x_i = [\![v_i]\!]_{combi}) \end{pmatrix} \implies \\
&\qquad \begin{pmatrix} \forall s' \in \mathcal{S}tate : \\ \langle S, s\rangle \to s' \implies \\ \quad (\forall x'_i \in \vec{x'} : x'_i \in Dom(s')) \land \\ \quad \exists v' \in \overrightarrow{\mathcal{T}erms(\Sigma_{theory} \cup \Sigma_{list})} : \\ \quad\quad \begin{pmatrix} stm_{id+1}[\vec{v}] \to^*_{\mathcal{R}_{list} \cup \mathcal{R}_S} stm_{id'}[\vec{v'}] \land \\ stm_{id'}[\vec{v'}] \in NF(\to_{\mathcal{R}_{list} \cup \mathcal{R}_S}) \land \\ |\vec{x'}| = |Dom(s')| = |\vec{v'}| \land \\ (\forall i \in \{0, |\vec{x'}| - 1\} : s'\ x'_i = [\![v'_i]\!]_{combi}) \end{pmatrix} \end{pmatrix} \land \\
&\qquad \begin{pmatrix} (\neg \exists s' \in \mathcal{S}tate : \langle S, s\rangle \to s') \implies \\ \quad \neg \exists \vec{v'} \in \overrightarrow{\mathcal{T}erms(\Sigma_{theory} \cup \Sigma_{list})} : \\ \quad\quad \begin{pmatrix} stm_{id+1}[\vec{v}] \to^*_{\mathcal{R}_{list} \cup \mathcal{R}_S} stm_{id'}[\vec{v'}] \land \\ stm_{id'}[\vec{v'}] \in NF(\to_{\mathcal{R}_{list} \cup \mathcal{R}_S}) \end{pmatrix} \end{pmatrix}
\end{aligned}$$

The formula above defines the connection between the program execution in natural semantics and the LCTRS. Assuming that the premise of the formula holds (which means that the LCTRS term $stm_{id+1}[\vec{v}]$ effectively represents the initial state s), we can conclude the following:

- If for the While++ statement S, it holds that the execution of this statement in some program state s terminates in some state s', then the formula suggests that the term $stm_{id'}[\vec{v'}]$ effectively represents the final state s' from the natural semantics representation. Also, we know that: $stm_{id+1}[\vec{v}] \to^*_{\mathcal{R}_{list} \cup \mathcal{R}_S} stm_{id'}[\vec{v'}]$ and $stm_{id'}[\vec{v'}] \in NF(\to_{\mathcal{R}_{list} \cup \mathcal{R}_S})$.

  According to the Lemma 1, any LCTRS generated by the translation function $\mathcal{T}$ is weakly orthogonal. The Theorem 1 states that the weakly orthogonal LCTRS is confluent. So, any LCTRS generated using the translation function $\mathcal{T}$ is confluent. As a consequence, the LCTRS has unique normal forms with respect to reduction.

  Because the term $stm_{id+1}[\vec{v}]$ is reducible to the term $stm_{id'}[\vec{v'}]$, $stm_{id'}[\vec{v'}]$ is a normal form, and the LCTRS has $UN^{\to}$ property, it means that $stm_{id'}[\vec{v'}]$ is the unique normal form for the term $stm_{id+1}[\vec{v}]$ ($stm_{id'}[\vec{v'}] = stm_{id+1}[\vec{v}]\downarrow$).

  If we refer to the comparison between the program execution in natural semantics and the LCTRS, we can see the following:

  - We assumed that there exists a derivation tree with the conclusion $\langle S, s\rangle \to s'$. That means we have a finite derivation tree, and the execution of a statement S in the state s terminates in the state s'.
  - The formula implies that there exists a finite reduction chain from the term $stm_{id+1}[\vec{v}]$ (represents the initial state s) to the unique normal form $stm_{id'}[\vec{v'}]$ (represents the final state s').

  So, we can conclude that if the While++ program executes without errors in natural semantics and terminates in the final state, the same happens with the equivalent LCTRS translation of the original program.

- If for the While++ statements it holds that there does not exist a state s', such that $\langle S, s\rangle \to s'$, then the derivation tree either does not exist (there is an error in the program execution) or the derivation tree is infinite (the program is looping).

  In this case, the formula suggests that there does not exist a vector of values $\vec{v'} \in \overrightarrow{\mathcal{T}erms(\Sigma_{theory} \cup \Sigma_{list})}$, such that: $stm_{id+1}[\vec{v}] \to^*_{\mathcal{R}_{list} \cup \mathcal{R}_S} stm_{id'}[\vec{v'}]$ and $stm_{id'}[\vec{v'}] \in NF(\to_{\mathcal{R}_{list} \cup \mathcal{R}_S})$.

  For the term $stm_{id+1}[\vec{v}]$ it means that it is either reducible to the normal form, which does not have a shape $stm_{id'}(\dots)$ (there is an error in the program execution) or the reduction chain is infinite (the program is looping).

  We can conclude that if there is an error in the execution of the While++ program, or this program is looping; the same happens with the equivalent LCTRS translation of the original program.

The definition of equivalence of a While++ statement and the LCTRS produced using the translation function $\mathcal{T}$ allows us to define the correctness theorem for the translation function $\mathcal{T}$.

Lemma 2 (Correctness of the translation function $\mathcal{T}$)**.** *For any While++ program, using the translation function $\mathcal{T}$, it is possible to generate an LCTRS equivalent to this program.*

Proof. Using the Definition 6.2.1 of the equivalence of While++ programs and the corresponding LCTRSs generated by the translation function $\mathcal{T}$, let's formulate the correctness theorem for the translation function $\mathcal{T}$:

$$\begin{aligned} CorrectnessTheorem(\mathcal{T}) \; := \forall S \in \mathcal{S}tm, s \in \mathcal{S}tate, \vec{x} \in \overrightarrow{\mathbf{Var}}, \\ id \in \mathbb{N}, \vec{v} \in \overrightarrow{\mathcal{T}erms(\Sigma_{theory} \cup \Sigma_{list})}, \\ (\Sigma_S, \mathcal{R}_S), \vec{x'} \in \overrightarrow{\mathbf{Var}}, id' \in \mathbb{N} : \\ Requirements(S, s, \vec{x}, id, (\Sigma_S, \mathcal{R}_S), \vec{v}, \vec{x'}, id') \implies \\ Soundness(S, s, \vec{x'}, \vec{v}, \mathcal{R}_S, id + 1, id') \wedge \\ Completeness(S, s, \Sigma_S, \mathcal{R}_S, id + 1, \vec{v}) \end{aligned}$$

Where:

$$\begin{aligned} Requirements(S, s, \vec{x}, id, (\Sigma_S, \mathcal{R}_S), \vec{v}, \vec{x'}, id') \; := \mathcal{T}(S, \vec{x}, id) = ((\Sigma_S, \mathcal{R}_S), \vec{x'}, id') \wedge \\ |\vec{x}| = |Dom(s)| = |\vec{v}| \wedge \\ (\forall x_i \in \vec{x} : x_i \in Dom(s)) \wedge \\ (\forall i \in \{0, |\vec{v}| - 1\} : v_i \in NF(\rightarrow_{\mathcal{R}_{list}})) \wedge \\ (\forall i \in \{0, |\vec{x}| - 1\} : s \; x_i = [\![v_i]\!]_{combi}) \end{aligned}$$

$$\begin{aligned} Soundness(S, s, \vec{x'}, \vec{v}, \mathcal{R}_S, id, id') \; := \forall s' \in \mathcal{S}tate : \\ \langle S, s \rangle \rightarrow s' \implies \\ (\forall x'_i \in \vec{x'} : x'_i \in Dom(s')) \wedge \\ \exists v' \in \overrightarrow{\mathcal{T}erms(\Sigma_{theory} \cup \Sigma_{list})} : \\ \begin{pmatrix} stm_{id+1}[\vec{v}] \rightarrow^*_{\mathcal{R}_{list} \cup \mathcal{R}_S} stm_{id'}[\vec{v'}] \wedge \\ stm_{id'}[\vec{v'}] \in NF(\rightarrow_{\mathcal{R}_{list} \cup \mathcal{R}_S}) \wedge \\ |\vec{x'}| = |Dom(s')| = |\vec{v'}| \wedge \\ (\forall i \in \{0, |\vec{x'}| - 1\} : s' \; x'_i = [\![v'_i]\!]_{combi}) \end{pmatrix} \end{aligned}$$

$$\begin{aligned} Completeness(S, s, \Sigma_S, \mathcal{R}_S, id, id', \vec{v}, \vec{v'}) \; := (\neg \exists s' \in \mathcal{S}tate : \langle S, s \rangle \rightarrow s') \implies \\ \neg \exists \vec{v'} \in \overrightarrow{\mathcal{T}erms(\Sigma_{theory} \cup \Sigma_{list})} : \\ \begin{pmatrix} stm_{id+1}[\vec{v}] \rightarrow^*_{\mathcal{R}_{list} \cup \mathcal{R}_S} stm_{id'}[\vec{v'}] \wedge \\ stm_{id'}[\vec{v'}] \in NF(\rightarrow_{\mathcal{R}_{list} \cup \mathcal{R}_S}) \end{pmatrix} \end{aligned}$$

To prove the correctness of the translation function $\mathcal{T}$ we need to show that:

$$CorrectnessTheorem(\mathcal{T}) = True$$

First, we assume that for the arbitrary statement $S \in \mathcal{S}tm$, initial state $s \in \mathcal{S}tate$, initial vector of While++ variables $\vec{x} \in \overrightarrow{\mathbf{Var}}$, initial identifier $id \in \mathbb{N}$, initial vector of LCTRS values $\vec{v} \in \overrightarrow{\mathcal{T}erms(\Sigma_{theory} \cup \Sigma_{list})}$, set of function symbols Σ_S, set of reduction rules $\mathcal{R}_S$, output identifier $id' \in \mathbb{N}$, and output vector of While++ variables $\vec{x'} \in \overrightarrow{\mathbf{Var}}$: $Requirements(S, s, \vec{x}, id, (\Sigma_S, \mathcal{R}_S), \vec{v}, \vec{x'}, id')$ holds.

That means:

1. $\mathcal{T}(S, \vec{x}, id) = ((\Sigma_S, \mathcal{R}_S), \vec{x'}, id')$

   The translation function $\mathcal{T}$ is defined for the statement S, initial vector of variables $\vec{x}$, and for the starting identifier $id \in \mathbb{N}$.

2. $|\vec{x}| = |Dom(s)| = |\vec{v}|$

   The size of the initial vector of variables $\vec{x}$, the number of variables for which there exists a value in the state s, and the size of the initial vector of values are equal.

3. $\forall x_i \in \vec{x} : x_i \in Dom(s)$

   For all variables from the vector $\vec{x}$, there exists some value mapping in the state s.

4. $\forall i \in \{0, |\vec{v}| - 1\} : v_i \in NF(\rightarrow_{\mathcal{R}_{list}})$:

   All elements of the vector $\vec{v}$ are in the normal form with respect to reduction $\rightarrow_{\mathcal{R}_{list}}$.

5. $\forall i \in \{0, |\vec{x}| - 1\} : s\ x_i = [\![v_i]\!]_{combi}$

   For all variables $x_i \in \vec{x}$ it holds that the corresponding value from the variable vector $v_i \in \vec{v}$ matches with the value assigned to the variable x_i in state s.

Remark. We will only prove the **soundness** part of the correctness theorem, because the **completeness** is beyond the scope of this thesis project.

6.3 Soundness proof

We will show that the **soundness** property holds for these arbitrary variables:

$$Soundness(S, s, \vec{x'}, \vec{v}, \mathcal{R}_S, id + 1, id')$$

Let's assume that there exists a derivation tree with $\langle S, s\rangle \rightarrow s'$ as a conclusion. Using this fact, we will create a structural induction proof on the shape of the derivation tree for $\langle S, s\rangle \rightarrow s'$. We will create a case distinction on the last rule that was applied. For each of these cases, we will show that the consequence of the implication holds.

Remark 1. For all the cases we can generalize the verification of the condition: $stm_{id'}[\vec{v'}] \in NF(\rightarrow_{\mathcal{R}_{list} \cup \mathcal{R}_S})$:

- According to Lemma 1, the LCTRS constructed using the translation function $\mathcal{T}$ is weakly orthogonal. That means, $(\Sigma_{theory} \cup \Sigma_{list} \cup \Sigma_S, \mathcal{R}_{list} \cup \mathcal{R}_S, \mathcal{I}, \mathcal{J})$ is weakly orthogonal.

- According to the Theorem 1, the weakly orthogonal LCTRS is confluent. That means, $(\Sigma_{theory} \cup \Sigma_{list} \cup \Sigma_S, \mathcal{R}_{list} \cup \mathcal{R}_S, \mathcal{I}, \mathcal{J})$ is confluent.

- That means, the LCTRS $(\Sigma_{theory} \cup \Sigma_{list} \cup \Sigma_S, \mathcal{R}_{list} \cup \mathcal{R}_S, \mathcal{I}, \mathcal{J})$ has unique normal forms with respect to reduction.

- Because in these cases the translation function $\mathcal{T}$ does not generate any rules that can be applied to $stm_{id'}[\vec{v'}]$, this term is a normal form for $(\Sigma_{theory} \cup \Sigma_{list} \cup \Sigma_S, \mathcal{R}_{list} \cup \mathcal{R}_S, \mathcal{I}, \mathcal{J})$: $stm_{id'}[\vec{v'}] \in NF(\rightarrow_{\mathcal{R}_{list} \cup \mathcal{R}_S})$.

Proving the **soundness** property for different cases:

6.3.1 Case $[\mathbf{skip_{ns}}]$

That means $S = (\texttt{skip})$, and the derivation tree becomes an instance of the axiom rule:

$$\langle \texttt{skip}, s\rangle \to s^{[\mathrm{skip_{ns}}]}$$

Consider the application of the translation function $\mathcal{T}$ to the statement S, initial vector of variables $\vec{x}$, and the initial identifier id:

$$[\![\texttt{skip}]\!]_{\vec{x},id} = \begin{pmatrix} \begin{pmatrix} \{stm_{id+1} : signature(\mathcal{VT}(\vec{x}), list), \\ stm_{id+2} : signature(\mathcal{VT}(\vec{x}), list)\}, \\ \{stm_{id+1}[\mathcal{VT}(\vec{x})] \to stm_{id+2}[\mathcal{VT}(\vec{x})]\ [true]\} \end{pmatrix}, \\ \vec{x}, \\ id + 2 \end{pmatrix}$$

1. Showing that $\forall x'_i \in \vec{x'} : x'_i \in Dom(s')$ holds:
   From the derivation tree, it follows that $s' = s$.
   From the translation function definition, it follows that $\vec{x'} = \vec{x}$.
   From our assumptions about the state s and the initial vector of variables $\vec{x}$ it follows that: $\forall x_i \in \vec{x} : x_i \in Dom(s)$.
   That means the statement (1) holds.

Now, we need to show that there exists a vector $\vec{v'} \in \overrightarrow{\mathcal{T}erms(\Sigma_{theory} \cup \Sigma_{list})}$, such that the starting term $stm_{id+1}[\vec{v}]$ is reducible to the normal form $stm_{id'}[\vec{v'}]$, and this vector of values v' represents the final state s' in the derivation tree.

To show this, we need to prove that the following statements hold:

2. Showing that $stm_{id+1}[\vec{v}] \to^*_{\mathcal{R}_{list} \cup \mathcal{R}_S} stm_{id'}[\vec{v'}]$:
   From the translation function definition, it follows that $id' = id + 2$.
   Because we assumed $Requirements(S, s, \vec{x}, id, (\Sigma_S, \mathcal{R}_S), \vec{v}, \vec{x'}, id')$ holds, the rules from the set $\mathcal{R}_{list}$ can't be applied to the function symbol $stm_{id+1}(\dots)$ or to the arguments of this function symbol.
   The set of LCTRS rules $\mathcal{R}_S$ generated by the translation function consists of only one rule:
   $$\mathcal{R}_S = \{stm_{id+1}[\mathcal{VT}(\vec{x})] \to stm_{id+2}[\mathcal{VT}(\vec{x})]\ [true]\}$$
   If we apply this rule to the term $stm_{id+1}[\vec{v}]$, we will obtain the following term: $stm_{id+2}[\vec{v}]$.
   The statement (2) holds.
   We have proven that there exists such a vector $\vec{v'}$ if we choose the vector $\vec{v}$ as it.
3. Showing that $stm_{id'}[\vec{v'}] \in NF(\to_{\mathcal{R}_{list} \cup \mathcal{R}_S})$:
   Because all elements of the vector $\vec{v'}$ are in normal form (according to the assumption about the vector $\vec{v}$) and according to the Remark 1: (3) holds.
4. Showing that $(|\vec{x'}| = |Dom(s')| = |\vec{v'}|)$ holds:
   From the derivation tree, it follows that $s' = s$.
   From the translation function definition, it follows that $\vec{x'} = \vec{x}$.
   From (2) we learned that $|\vec{v'}| = |\vec{v}|$.
   From the assumptions about the state s, the initial vector of variables $\vec{x}$ and the initial vector of values it holds that: $|\vec{x}| = |Dom(s)| = |\vec{v}|$.
   That means the statement (4) holds.

5. Showing that $\forall i \in \{0, |\vec{x'}| - 1\} : s'\ x'_i = v'_i$ holds:
   From the derivation tree, it follows that $s' = s$.
   From the translation function definition, it follows that $\vec{x'} = \vec{x}$.
   From (2) it holds that $\vec{v'} = \vec{v}$.
   From the assumptions about the state s, the initial vector of variables $\vec{x}$ and the initial vector of values, it holds that: $\forall i \in \{0, |\vec{x}| - 1\} : s\ x_i = v_i$
   That means the statement (5) holds.

We proved that $Soundness(S, s, \vec{x'}, \vec{v}, \mathcal{R}_S, id + 1, id')$ holds in this case.

6.3.2 Case $[\mathbf{declare_{type}}]$

That means $S = (type\ var\ \texttt{:=}\ expr)$, and the derivation tree becomes an instance of the axiom rule:

$$\langle type\ var\ \texttt{:=}\ expr, s\rangle \rightarrow s[var \mapsto \mathcal{E}val_{type}[\![expr]\!]s]^{[\mathrm{declare_{type}}]}$$

Because we know that this rule was applied, the following condition holds:

$$\forall x' \in Dom(s) : x' \neq var \wedge var \in \mathbf{Var}_{type} \wedge \mathcal{E}val_{type}[\![expr]\!]s \downarrow$$

Consider the application of the translation function $\mathcal{T}$ to the statement S, initial vector of variables $\vec{x}$, and the initial identifier id:

$$\frac{var \notin \vec{x}}{[\![type\ var\ \texttt{:=}\ expr]\!]_{\vec{x},id} = \begin{pmatrix} \begin{pmatrix} \{stm_{id+1} : signature(\mathcal{VT}(\vec{x}), list), \\ stm_{id+2} : signature(\mathcal{VT}(\vec{x}) \cdot [\mathcal{ET}(expr) : type], list)\}, \\ \{stm_{id+1}[\mathcal{VT}(\vec{x})] \rightarrow stm_{id+2}[\mathcal{VT}(\vec{x}) \cdot [\mathcal{ET}(expr) : type]]\ [true]\} \end{pmatrix}, \\ \vec{x} \cdot [var : type], \\ id + 2 \end{pmatrix}}$$

1. Showing that $\forall x'_i \in \vec{x'} : x'_i \in Dom(s')$ holds:
   From the derivation tree it follows that $s' = s[var \mapsto \mathcal{E}val_{type}[\![expr]\!]s]$.
   From the translation function definition, it follows that $\vec{x'} = \vec{x} \cdot [var : type]$.
   From the translation function requirement, it follows that $\forall x_i \in \vec{x} : x_i \neq var$.
   From the translation function definition, it follows that $\forall x_i \in \vec{x} : x_i \in Dom(s)$.
   That means for the variable vector $\vec{x'}$ and the state s' it holds that:

   $$\forall i \in \{0, |\vec{x'}| - 2\} : (x_i = x'_i) \wedge (x'_i \in Dom(s')) \wedge x'_{|\vec{x}|} = var \wedge var \in Dom(s')$$

   From this it follows that: $\forall x'_i \in \vec{x'} : x'_i \in Dom(s')$.
   That means (1) holds.

Now, we need to show that there exists a vector $\vec{v'} \in \overrightarrow{\mathcal{T}erms(\Sigma_{theory} \cup \Sigma_{list})}$, such that the starting term $stm_{id+1}[\vec{v}]$ is reducible to the normal form $stm_{id'}[\vec{v'}]$, and this vector of values v' represents the final state s' in the derivation tree.

To show this, we need to prove that the following statements hold:

2. Showing that $stm_{id+1}[\vec{v}] \to^*_{\mathcal{R}_{list} \cup \mathcal{R}_S} stm_{id'}[\vec{v'}]$:
   From the translation function definition, it follows that $id' = id + 2$.
   Because we assumed $Requirements(S, s, \vec{x}, id, (\Sigma_S, \mathcal{R}_S), \vec{v}, \vec{x'}, id')$ holds, the rules from the set $\mathcal{R}_{list}$ can't be applied to the function symbol $stm_{id+1}(\dots)$ or to the arguments of this function symbol.
   The set of LCTRS rules $\mathcal{R}_S$ generated by the translation function consists of only one rule:

$$\mathcal{R}_S = \{stm_{id+1}[\mathcal{VT}(\vec{x})] \to stm_{id+2}[\mathcal{VT}(\vec{x}) \cdot \mathcal{ET}(expr) : type]\ [true]\}$$

   If we apply this rule to the term $stm_{id+1}[\vec{v}]$ and then apply all possible calculation reduction steps, we will obtain the following term: $stm_{id+2}[\vec{v} \cdot (\mathcal{ET}(expr)\sigma \downarrow_{calc})]$.
   The substitution σ is defined as:

$$\forall x_i \in \vec{x} : \sigma(\mathcal{VT}(x_i)) = v_i$$

   The statement (2) holds.
   We have proven that there exists such a vector $\vec{v'}$ if we choose the vector $\vec{v} \cdot (\mathcal{ET}(expr)\sigma \downarrow_{calc})$ as it.
   Also, we can see that $\mathcal{ET}(expr)\sigma \downarrow_{calc} \in \mathcal{T}erms(\Sigma_{theory} \cup \Sigma_{list})$, because the substitution σ replaces all possible variables that can exist in the term $\mathcal{ET}(expr)$ with values from the vector $\vec{v}$. It happens because the While++ expression $expr$ only includes the variables declared in the initial state s, and the vector $\vec{v}$ represents that initial state.
3. Showing that $stm_{id'}[\vec{v'}] \in NF(\to_{\mathcal{R}_{list} \cup \mathcal{R}_S})$:
   Because all elements of the vector $\vec{v'}$ are in normal form (according to the assumption about the vector $\vec{v}$ and because $\mathcal{ET}(expr)\sigma \downarrow_{calc}$ is in normal form) and according to Remark 1: (3) holds.
4. Showing that $(|\vec{x'}| = |Dom(s')| = |\vec{v'}|)$ holds:
   From the translation function definition, it follows that $\vec{x'} = \vec{x} \cdot var : type$. That means: $|\vec{x'}| = |\vec{x}| + 1$.
   From (2) it holds that $\vec{v'} = \vec{v} \cdot \mathcal{ET}(expr)\sigma \downarrow_{calc}$. That means: $|\vec{v'}| = |\vec{v}| + 1$.
   From the assumptions about the state s, the initial vector of variables $\vec{x}$ and the initial vector of values, it holds that: $|\vec{x}| = |Dom(s)| = |\vec{v}|$.
   From the translation function definition, it follows that $\forall x_i \in \vec{x} : x_i \in Dom(s)$.
   That means the domain of the state s exactly matches the set of variables from the vector $\vec{x}$.
   From the derivation tree it follows that $s' = s[var \mapsto \mathcal{E}val_{type}[\![expr]\!]s]$.
   Because by the rule definition $\forall x_i \in \vec{x} : x_i \neq var$, then $|Dom(s')| = |Dom(s)| + 1$ holds.
   Obviously: $|\vec{x}| + 1 = |Dom(s)| + 1 = |\vec{v}| + 1$. That means: $|\vec{x'}| = |Dom(s')| = |\vec{v'}|$, and (4) holds.
5. Showing that $\forall i \in \{0, |\vec{x'}| - 1\} : s'\ x'_i = [\![v'_i]\!]_{combi}$ holds:
   From the translation function definition it follows that $\vec{x'} = \vec{x} \cdot var$.
   That means:
   $\forall i \in \{0, |\vec{x'}| - 2\} : x'_i = x_i$ and $x'_{|\vec{x'}|-1} = var$
   From (2) it holds that $\vec{v'} = \vec{v} \cdot (\mathcal{ET}(expr)\sigma \downarrow_{calc})$.
   That means:

– $\forall i \in \{0, |\vec{v'}| - 2\} : v'_i = v_i$,
– $v'_{|\vec{v'}-1|} = \mathcal{ET}(expr)\sigma \downarrow_{calc}$

From the derivation tree it follows that $s' = s[var \mapsto \mathcal{E}val_{type}[\![expr]\!]s]$.
That means: $s'\ var = \mathcal{E}val_{type}[\![expr]\!]s$.
From the assumptions about the state s, the initial vector of variables $\vec{x}$ and the initial vector of values, it holds that:

$$\forall i \in \{0, |\vec{x}| - 1\} : s\ x_i = [\![v_i]\!]_{combi}$$

We can say that $s'\ x'_{|\vec{x'}|-1} = [\![v'_{|\vec{v'}|-1}]\!]_{combi}$, because:
– $s'\ x'_{|\vec{x'}|-1} = s'\ var = \mathcal{E}val_{type}[\![expr]\!]s$,
– $v'_{|\vec{v'}|-1} = \mathcal{ET}(expr)\sigma \downarrow_{calc}$,
– and, $\mathcal{E}val_{type}[\![expr]\!]s = [\![\mathcal{ET}(expr)\sigma \downarrow_{calc}]\!]_{combi}$

That means the following holds:

$$(\forall i \in \{0, |\vec{x'}| - 2\} : s'\ x'_i = [\![v'_i]\!]_{combi}) \wedge (s'\ x'_{|\vec{x'}|-1} = [\![v'_{|\vec{x'}|-1}]\!]_{combi})$$

From this it follows: $(\forall i \in \{0, |\vec{x'}| - 1\} : s'\ x'_i = [\![v'_i]\!]_{combi})$, and (5) holds.

We proved that $Soundness(S, s, \vec{x'}, \vec{v}, \mathcal{R}_S, id + 1, id')$ holds in this case.

6.3.3 Case [$\mathbf{ass_{type}}$]

That means $S = (var\ \texttt{:=}\ expr)$, and the derivation tree becomes an instance of the axiom rule:

$$\langle var\ \texttt{:=}\ expr, s\rangle \rightarrow s[var \mapsto \mathcal{E}val_{type}[\![expr]\!]s]^{[\mathrm{ass_{type}}]}$$

Consider the application of the translation function $\mathcal{T}$ to the statement S, initial vector of variables $\vec{x}$, and the initial identifier id:

$$\frac{\begin{array}{l}(|\vec{x}| = |\vec{w}|) \wedge \\ \exists p \in \{0, |\vec{x}| - 1\} : \\ \quad \begin{pmatrix}((x_p = var) \wedge (w_p = \mathcal{ET}(expr))) \quad \wedge \\ (\forall i \in \{0, |\vec{x}| - 1\} : (i \neq p) \implies (w_i = \mathcal{ET}(x_i)))\end{pmatrix}\end{array}}{[\![var\ \texttt{:=}\ expr]\!]_{\vec{x},id} = \left(\begin{pmatrix}\{stm_{id+1} : signature(\mathcal{VT}(\vec{x}), list), \\ stm_{id+2} : signature(\mathcal{VT}(\vec{x}), list)\}, \\ \{stm_{id+1}[\mathcal{VT}(\vec{x})] \rightarrow stm_{id+2}[\vec{w}]\ [true]\}\end{pmatrix}, \vec{x}, id + 2\right)}$$

1. Showing that $\forall x'_i \in \vec{x'} : x'_i \in Dom(s')$ holds:
   From the translation function, it follows that $\vec{x'} = \vec{x}$.
   From the derivation tree it follows that: $s' = s[var \mapsto \mathcal{E}val_{type}[\![expr]\!]s]$.
   That means $Dom(s) = Dom(s')$.
   From our assumptions about the initial state s and the initial vector of variables $\vec{x}$, it follows that $\forall x_i \in \vec{x} : x_i \in Dom(s)$.
   That means $\forall x'_i \in \vec{x'} : x'_i \in Dom(s')$.
   So, we proved that (1) holds.

Now, we need to show that there exists a vector $\vec{v'} \in \overrightarrow{\mathcal{T}erms(\Sigma_{theory} \cup \Sigma_{list})}$, such that the starting term $stm_{id+1}[\vec{v}]$ is reducible to the normal form $stm_{id'}[\vec{v'}]$, and this vector of values v' represents the final state s' in the derivation tree.

To show this, we need to prove that the following statements hold:

2. Showing that $stm_{id+1}[\vec{v}] \to^*_{\mathcal{R}_{list} \cup \mathcal{R}_S} stm_{id'}[\vec{v'}]$ holds:
   From the translation function definition, it follows that $id' = id + 2$.
   Because we assumed $Requirements(S, s, \vec{x}, id, (\Sigma_S, \mathcal{R}_S), \vec{v}, \vec{x'}, id')$ holds, the rules from the set $\mathcal{R}_{list}$ can't be applied to the function symbol $stm_{id+1}(\dots)$ or to the arguments of this function symbol.
   The set of LCTRS rules $\mathcal{R}_S$ generated by the translation function consists of only one rule:
   $$\mathcal{R}_S = \{stm_{id+1}[\mathcal{VT}(\vec{x})] \to stm_{id+2}[\vec{w}]\ [true]\}$$
   If we apply this rule to the term $stm_{id+1}[\vec{v}]$ and then apply all possible calculation reduction steps, we will obtain the following term: $stm_{id+2}[\vec{v'}]$.
   And for this term $stm_{id+2}[\vec{v'}]$ it holds that $\exists p \in \{0, |\vec{x}|-1\} : v'_p = \mathcal{ET}(expr)\sigma \downarrow_{calc}$, and the remaining values stay as in vector $\vec{v}$.
   Where σ is a substitution defined as:
   $$\forall x_i \in \vec{x} : \sigma(\mathcal{VT}(x_i)) = v_i$$
   The statement (2) holds.
   We have proven that there exists such a vector $\vec{v'}$.
   Also, we can see that $\mathcal{ET}(expr)\sigma \downarrow_{calc} \in \mathcal{T}erms(\Sigma_{theory} \cup \Sigma_{list})$, because the substitution σ replaces all possible variables that can exist in the term $\mathcal{ET}(expr)$ with values from the vector $\vec{v}$. It happens because the While++ expression $expr$ only includes the variables declared in the initial state s, and the vector $\vec{v}$ represents that initial state.
3. Showing that $stm_{id'}[\vec{v'}] \in NF(\to_{\mathcal{R}_{list} \cup \mathcal{R}_S})$ holds:
   Because all elements of the vector $\vec{v'}$ are in normal form (according to the assumption about the vector $\vec{v}$ and because $\mathcal{ET}(expr)\sigma \downarrow_{calc}$ is in normal form) and according to Remark 1: (3) holds.
4. Showing that $|\vec{x'}| = |Dom(s')| = |\vec{v'}|$ holds:
   According to the translation function definition, we know that: $|\vec{x}| = |\vec{x'}|$.
   Because of the properties of the set of LCTRS rules $\mathcal{R}_S$, we know that: $|\vec{v}| = |\vec{v'}|$.
   From the derivation tree, it follows that $Dom(s) = Dom(s')$.
   From our assumptions about the vectors $\vec{x'}$, $\vec{v}$, and the initial state s, it follows that $|\vec{x}| = |Dom(s)| = |\vec{v}|$.
   That means (4) holds.
5. Showing that $\forall i \in \{0, |\vec{x'}| - 1\} : s'\ x'_i = [\![v'_i]\!]_{combi}$ holds:
   From our assumptions about the initial state s, initial vector of variables $\vec{x}$ and initial vector of values $\vec{v}$ it follows that $\forall i \in \{0, |\vec{x}| - 1\} : s\ x_i = [\![v_i]\!]_{combi}$.
   From the translation function, it follows that:
   $$\exists p \in \{0, |\vec{x}| - 1\} : \begin{pmatrix} (x_p = var) \wedge (w_p = \mathcal{ET}(expr)) \wedge \\ \forall i \in \{0, |\vec{x}| - 1\} : (i \neq p) \implies (w_i = \mathcal{ET}(x_i)) \end{pmatrix}$$

That means the resulting values vector $\vec{v'}$ will contain all the values of the vector $\vec{v}$, but on the position of the variable *var* in the vector $\vec{x}$, the vector $\vec{v'}$ will contain the value $\mathcal{ET}(expr)\sigma \downarrow_{calc}$.
The substitution σ is defined as:

$$\forall x_i \in \vec{x} : \sigma(\mathcal{VT}(x_i)) = v_i$$

From the derivation tree, it follows that:

$$\forall x \in Dom(s') : \begin{pmatrix} (x \neq var) \implies (s'\ x = s\ x) \\ (x = var) \implies (s'\ x = \mathcal{E}val_{type}[\![expr]\!]s) \end{pmatrix}$$

Also, we know, that $\mathcal{E}val_{type}[\![expr]\!]s = [\![\mathcal{ET}(expr)\sigma \downarrow_{calc}]\!]_{combi}$.
From this it follows that $\forall i \in \{0, |\vec{x'}| - 1\} : s'\ x'_i = [\![v'_i]\!]_{combi}$.
That means (5) holds.

We proved that $Soundness(S, s, \vec{x'}, \vec{v}, \mathcal{R}_S, id + 1, id')$ holds in this case.

6.3.4 Case [$\mathbf{print_{type}}$]

That means $S =$ (`printType(`*expr*`)`), and the derivation tree becomes an instance of the axiom rule:

$$\langle \texttt{printType(}expr\texttt{)}, s\rangle \rightarrow s[Out \mapsto \mathbf{consT}(\mathcal{E}val[\![expr]\!]s, s\ Out)]^{[\text{print}_{\text{type}}]}$$

Because we know that this rule was applied, the following condition holds:

$$Out \in Dom(s) \wedge \mathcal{E}val_{type}[\![expr]\!]s \downarrow$$

Consider the application of the translation function $\mathcal{T}$ to the statement S, initial vector of variables $\vec{x}$, and the initial identifier *id*:

$$\frac{\begin{array}{l}(|\vec{x}| = |\vec{w}|)\wedge \\ \exists p \in \{0, |\vec{x}| - 1\} : \\ \quad \begin{pmatrix} (x_p = \texttt{Out}) \wedge (w_p = consT(\mathcal{ET}(expr), \mathcal{VT}(\texttt{Out})))\wedge \\ (\forall i \in \{0, |\vec{x}| - 1\} : (i \neq p) \implies (w_i = \mathcal{ET}(x_i))) \end{pmatrix}\end{array}}{[\![\texttt{printType(}expr\texttt{)}]\!]_{\vec{x},id} = \left(\begin{pmatrix} \{stm_{id+1} : signature(\mathcal{VT}(\vec{x}), list), \\ stm_{id+2} : signature(\mathcal{VT}(\vec{x}), list)\}, \\ \{stm_{id+1}[\mathcal{VT}(\vec{x})] \rightarrow stm_{id+2}[\vec{w}]\ [true]\} \end{pmatrix}, \vec{x}, id + 2\right)}$$

1. Showing that $\forall x'_i \in \vec{x'} : x'_i \in Dom(s')$ holds:
   From the translation function, it follows that $\vec{x'} = \vec{x}$.
   From the derivation tree, it follows that:
   - $s' = s[\texttt{Out} \mapsto \mathbf{consT}(\mathcal{E}val[\![expr]\!]s, s\ \texttt{Out})]$,
   - $\texttt{Out} \in Dom(s)$.

   That means $Dom(s) = Dom(s')$.
   From our assumptions about the initial state s and the initial vector of variables $\vec{x}$, it follows that $\forall x_i \in \vec{x} : x_i \in Dom(s)$.
   That means $\forall x'_i \in \vec{x'} : x'_i \in Dom(s')$.
   So, we proved that (1) holds.

Now, we need to show that there exists a vector $\vec{v'} \in \overrightarrow{\mathcal{T}erms(\Sigma_{theory} \cup \Sigma_{list})}$, such that the starting term $stm_{id+1}[\vec{v}]$ is reducible to the normal form $stm_{id'}[\vec{v'}]$, and this vector of values v' represents the final state s' in the derivation tree.

To show this, we need to prove that the following statements hold:

2. Showing that $stm_{id+1}[\vec{v}] \to^*_{\mathcal{R}_{list} \cup \mathcal{R}_S} stm_{id'}[\vec{v'}]$:

   Because we assumed $Requirements(S, s, \vec{x}, id, (\Sigma_S, \mathcal{R}_S), \vec{v}, \vec{x'}, id')$ holds, the rules from the set $\mathcal{R}_{list}$ can't be applied to the function symbol $stm_{id+1}(\dots)$ or to the arguments of this function symbol.

   The set of LCTRS rules $\mathcal{R}_S$ generated by the translation function consists of only one rule:

   $$\mathcal{R}_S = \{stm_{id+1}[\mathcal{VT}(\vec{x})] \to stm_{id+2}[\vec{w}]\ [true]\}$$

   If we apply this rule to the term $stm_{id+1}[\vec{v}]$ and then apply all possible calculation reduction steps, we will obtain the following term: $stm_{id+2}[\vec{v'}]$.

   And for this term $stm_{id+2}[\vec{v'}]$ it holds that:

   $$\exists p \in \{0, |\vec{x}| - 1\} : v'_p = consT(\mathcal{ET}(expr)\sigma \downarrow_{calc}, \mathcal{VT}(\texttt{Out}))$$

   And the remaining values stay as in vector $\vec{v}$.

   The substitution σ is defined as:

   $$\forall x_i \in \vec{x} : \sigma(\mathcal{VT}(x_i)) = v_i$$

   The statement (2) holds.

   We have proven that there exists such a vector $\vec{v'}$.

   Also, we can see that $consT(\mathcal{ET}(expr)\sigma \downarrow_{calc}, \mathcal{VT}(\texttt{Out})) \in \mathcal{T}erms(\Sigma_{theory} \cup \Sigma_{list})$, because the substitution σ replaces all possible variables that can exist in the term $\mathcal{ET}(expr)$ with values from the vector $\vec{v}$. It happens because the While++ expression $expr$ only includes the variables declared in the initial state s, and the vector $\vec{v}$ represents that initial state.

3. Showing that $stm_{id'}[\vec{v'}] \in NF(\to_{\mathcal{R}_{list} \cup \mathcal{R}_S})$:

   Because all elements of the vector $\vec{v'}$ are in normal form (according to the assumption about the vector $\vec{v}$ and because $consT(\mathcal{ET}(expr)\sigma \downarrow_{calc}, \mathcal{VT}(\texttt{Out}))$ is in normal form) and according to Remark 1: (3) holds.

4. Showing that $(|\vec{x'}| = |Dom(s')| = |\vec{v'}|)$ holds:

   According to the translation function definition, we know that: $|\vec{x}| = |\vec{x'}|$.

   Because of the properties of the set of LCTRS rules $\mathcal{R}_S$, we know that $|\vec{v}| = |\vec{v'}|$.

   From the derivation tree, it follows that $Dom(s) = Dom(s')$.

   From our assumptions about the initial state s, initial vector of variables $\vec{x'}$ and the initial vector of values, it follows that $|\vec{x}| = |Dom(s)| = |\vec{v}|$.

   That means (4) holds.

5. Showing that $\forall i \in \{0, |\vec{x'}| - 1\} : s'\ x'_i = [\![v'_i]\!]_{combi}$ holds:

   From our assumptions about the initial state s, initial vector of variables $\vec{x}$, and initial vector of values $\vec{v}$: it follows that $\forall i \in \{0, |\vec{x}| - 1\} : s\ x_i = [\![v_i]\!]_{combi}$.

   From the translation function, it follows that:

   $$\exists p \in \{0, |\vec{x}| - 1\} : \begin{pmatrix} (x_p = \texttt{Out}) \wedge (w_p = consT(\mathcal{ET}(expr), \mathcal{VT}(\texttt{Out}))) \wedge \\ (\forall i \in \{0, |\vec{x}| - 1\} : (i \neq p) \implies (w_i = \mathcal{ET}(x_i))) \end{pmatrix}$$

That means the resulting values vector $\vec{v'}$ will contain all the values of the vector $\vec{v}$, but in the position of the variable Out in the vector $\vec{x}$, the vector $\vec{v'}$ will contain the value $consT(\mathcal{ET}(expr)\sigma \downarrow_{calc}, v_p)$.
Where σ is a substitution defined as:

$$\forall x_i \in \vec{x} : \sigma(\mathcal{VT}(x_i)) = v_i$$

From the derivation tree, it follows that:

$$\forall x \in Dom(s') : \begin{pmatrix} (x \neq \texttt{Out}) \implies (s'\ x = s\ x) \\ (x = \texttt{Out}) \implies (s'\ x = \mathbf{consT}(\mathcal{E}val[\![expr]\!]s, s\ x)) \end{pmatrix}$$

Also we know that $\mathcal{E}val[\![expr]\!]s = [\![\mathcal{ET}(expr)\sigma \downarrow_{calc}]\!]_{combi}$.
From this it follows that the lists:

- $consT(\mathcal{ET}(expr)\sigma \downarrow_{calc}, v_p)$,
- and $\mathbf{consT}(\mathcal{E}val[\![expr]\!]s, s\ \texttt{Out})$

are equal too.
From this it follows that $\forall i \in \{0, |\vec{x'}| - 1\} : s'\ x'_i = [\![v'_i]\!]_{combi}$.
That means (5) holds.

We proved that $Soundness(S, s, \vec{x'}, \vec{v}, \mathcal{R}_S, id + 1, id')$ holds in this case.

6.3.5 Case $[\mathbf{read_{type}}]$

That means $S = (\texttt{readType(}var\texttt{)})$, and the derivation tree becomes an instance of the axiom rule:

$$\langle \texttt{readType(}var\texttt{)}, s\rangle \rightarrow s[var \mapsto \mathbf{headT}(s\ \texttt{In}) : type, \texttt{In} \mapsto \mathbf{tail}(s\ \texttt{In}) : \mathbb{L}]^{[\text{read}_{\text{type}}]}$$

Because we know that the rule was applied, the following condition holds:

$$var \in Dom(s) \wedge \texttt{In} \in Dom(s) \wedge \mathbf{headT}(s\ \texttt{In}) \downarrow \wedge\ \mathbf{tail}(s\ \texttt{In}) \downarrow$$

Consider the application of the translation function $\mathcal{T}$ to the statement S, initial vector of variables $\vec{x}$, and the initial identifier id:

$$\frac{\begin{array}{l}(|\vec{x}| = |\vec{w}|) \wedge \\ \exists p_1, p_2 \in \{0, |\vec{x}| - 1\} : \\ \quad \begin{pmatrix} (p_1 \neq p_2) \wedge (x_{p_1} = var) \wedge (x_{p_2} = \texttt{In}) \quad \wedge \\ (w_{p_1} = headT(\mathcal{VT}(\texttt{In}))) \wedge (w_{p_2} = tailT(\mathcal{VT}(\texttt{In}))) \quad \wedge \\ (\forall i \in \{0, |\vec{x}| - 1\} : (i \neq p_1 \wedge i \neq p_2) \implies (w_i = \mathcal{ET}(x_i))) \end{pmatrix}\end{array}}{[\![\texttt{readType(}var\texttt{)}]\!]_{\vec{x},id} = \left(\begin{pmatrix} \{stm_{id+1} : signature(\mathcal{VT}(\vec{x}), list), \\ stm_{id+2} : signature(\mathcal{VT}(\vec{x}), list)\}, \\ \{stm_{id+1}[\mathcal{VT}(\vec{x})] \rightarrow stm_{id+2}[\vec{w}]\ [true]\} \end{pmatrix}, \vec{x}, id + 2 \right)}$$

1. Showing that $\forall x'_i \in \vec{x'} : x'_i \in Dom(s')$ holds:
   From the translation function, it follows that $\vec{x'} = \vec{x}$.
   From the derivation tree, it follows that:
   - $s' = s[var \mapsto \mathbf{headT}(s\ \texttt{In}) : type, \texttt{In} \mapsto \mathbf{tail}(s\ \texttt{In}) : \mathbb{L}]$,

– $var \in Dom(s)$,
– $\texttt{In} \in Dom(s)$

That means: $Dom(s) = Dom(s')$

From our assumptions about the initial state s and the initial vector of variables $\vec{x}$, it follows that $\forall x_i \in \vec{x} : x_i \in Dom(s)$.

That means $\forall x'_i \in \vec{x'} : x'_i \in Dom(s')$.

So, we proved that (1) holds.

Now, we need to show that there exists a vector $\vec{v'} \in \overrightarrow{\mathcal{T}erms(\Sigma_{theory} \cup \Sigma_{list})}$, such that the starting term $stm_{id+1}[\vec{v}]$ is reducible to the normal form $stm_{id'}[\vec{v'}]$, and this vector of values v' represents the final state s' in the derivation tree.

To show this, we need to prove that the following statements hold:

2. Showing that $stm_{id+1}[\vec{v}] \to^*_{\mathcal{R}_{list} \cup \mathcal{R}_S} stm_{id'}[\vec{v'}]$:
   From the translation function definition it follows that $id' = id + 2$.
   Because we assumed $Requirements(S, s, \vec{x}, id, (\Sigma_S, \mathcal{R}_S), \vec{v}, \vec{x'}, id')$ holds, the rules from the set $\mathcal{R}_{list}$ can't be applied to the function symbol $stm_{id+1}(\dots)$ or to the arguments of this function symbol.
   The set of LCTRS rules $\mathcal{R}_S$ generated by the translation function consists of only one rule:
   $$\mathcal{R}_S = \{stm_{id+1}[\mathcal{VT}(\vec{x})] \to stm_{id+2}[\vec{w}]\ [true]\}$$
   If we apply this rule to the term $stm_{id+1}[\vec{v}]$ and then apply the reduction rules from $\mathcal{R}_{list}$, we will obtain the following term: $stm_{id+2}[\vec{v'}]$.
   And for this term $stm_{id+2}[\vec{v'}]$ it holds that:
   $$\exists p_1, p_2 \in \{0, |\vec{x}| - 1\} : v'_{p_1} = head(\mathcal{VT}(\texttt{In}))\sigma \downarrow_{\mathcal{R}_{list}} \wedge v'_{p_1} = tail(\mathcal{VT}(\texttt{In}))\sigma \downarrow_{\mathcal{R}_{list}}$$
   And the remaining values stay as in vector $\vec{v}$.
   The substitution σ is defined as:
   $$\forall x_i \in \vec{x} : \sigma(\mathcal{VT}(x_i)) = v_i$$
   The statement (2) holds.
   We have proven that there exists such a vector $\vec{v'}$.
   Also, we can see that:
   – $head(\mathcal{VT}(\texttt{In}))\sigma \downarrow_{\mathcal{R}_{list}} \in \mathcal{T}erms(\Sigma_{theory} \cup \Sigma_{list})$,
   – and $tail(\mathcal{VT}(\texttt{In}))\sigma \downarrow_{\mathcal{R}_{list}} \in \mathcal{T}erms(\Sigma_{theory} \cup \Sigma_{list})$

   because the substitution σ replaces the variable $\mathcal{VT}(\texttt{In})$ with its value from the vector $\vec{v}$.
3. Showing that $stm_{id'}[\vec{v'}] \in NF(\to_{\mathcal{R}_{list} \cup \mathcal{R}_S})$:
   Because all elements of the vector $\vec{v'}$ are in normal form (according to the assumption about the vector $\vec{v}$ and because $head(\mathcal{VT}(\texttt{In}))\sigma \downarrow_{\mathcal{R}_{list}}$ and $tail(\mathcal{VT}(\texttt{In}))\sigma \downarrow_{\mathcal{R}_{list}}$ are in normal form) and according to Remark 1: (3) holds.
4. Showing that $(|\vec{x'}| = |Dom(s')| = |\vec{v'}|)$ holds:
   According to the translation function definition, we know that: $|\vec{x}| = |\vec{x'}|$.
   Because of the properties of the set of LCTRS rules $\mathcal{R}_S$, we know that: $|\vec{v}| = |\vec{v'}|$.
   From the derivation tree, it follows that $Dom(s) = Dom(s')$.
   From our assumptions about the initial state s, initial vector of variables $\vec{x'}$ and the initial vector of values $\vec{v}$ it follows that $|\vec{x}| = |Dom(s)| = |\vec{v}|$.
   That means (4) holds.

5. Showing that $\forall i \in \{0, |\vec{x'}| - 1\} : s'\ x'_i = [\![v'_i]\!]_{combi}$ holds:
   From our assumptions about the initial state s, initial vector of variables $\vec{x}$ and the initial vector of values $\vec{v}$ it follows that $\forall i \in \{0, |\vec{x}| - 1\} : s\ x_i = [\![v_i]\!]_{combi}$.
   From the translation function definition, it follows that:

$$\exists p_1, p_2 \in \{0, |\vec{x}|-1\} : \begin{pmatrix} (p_1 \neq p_2) \wedge (x_{p_1} = var) \wedge (x_{p_2} = \texttt{In}) \quad \wedge \\ (w_{p_1} = headT(\mathcal{VT}(\texttt{In}))) \wedge (w_{p_2} = tail(\mathcal{VT}(\texttt{In}))) \quad \wedge \\ (\forall i \in \{0, |\vec{x}| - 1\} : (i \neq p_1 \wedge i \neq p_2) \implies (w_i = \mathcal{ET}(x_i))) \end{pmatrix}$$

   That means the resulting values vector $\vec{v'}$ contains all the values of the vector $\vec{v}$, but on the position of the variable var in the vector $\vec{x}$, the vector $\vec{v'}$ contains the value $headT(\mathcal{VT}(\texttt{In}))\sigma \downarrow_{\mathcal{R}_{list}}$. And on the position of the variable In in the vector $\vec{x}$, the vector $\vec{v'}$ contains the value $tail(\mathcal{VT}(\texttt{In}))\sigma \downarrow_{\mathcal{R}_{list}}$.
   From the derivation tree, it follows that:

$$\forall x \in Dom(s') : \begin{pmatrix} (x \neq var \wedge x \neq In) \implies (s'\ x = s\ x) \\ (x = var) \implies (s'\ x = \mathbf{head}T(s\ \texttt{In})) \\ (x = In) \implies (s'\ x = \mathbf{tail}(s\ \texttt{In})) \end{pmatrix}$$

   From this it follows that $\forall i \in \{0, |\vec{x'}| - 1\} : s'\ x'_i = [\![v'_i]\!]_{combi}$.
   That means (5) holds.

We proved that $Soundness(S, s, \vec{x'}, \vec{v}, \mathcal{R}_S, id + 1, id')$ holds in this case.

6.3.6 Case [$\mathbf{while}^{\perp}_{\mathbf{ns}}$]

That means $S = (\texttt{while}\ b\ \texttt{do}\ S)$, $\mathcal{B}[\![b]\!]s = \perp$, and the derivation tree becomes an instance of the axiom rule:

$$\langle \texttt{while}\ b\ \texttt{do}\ S, s\rangle \rightarrow s^{[\text{while}^{\perp}_{\text{ns}}]}$$

Consider the application of the translation function $\mathcal{T}$ to the statement S, initial vector of variables $\vec{x}$, and the initial identifier id:

$$\frac{[\![S]\!]_{\vec{x},id+1} = \left(\begin{pmatrix}\Sigma_{terms_1}, \\ \mathcal{R}_1\end{pmatrix}, \vec{x} \cdot \vec{y}, id''\right)}{[\![\texttt{while}\ b\ \texttt{do}\ S]\!]_{\vec{x},id} = \left(\begin{pmatrix} \Sigma_{terms_1} \cup \\ \{stm_{id+1} : signature(\mathcal{VT}(\vec{x}), list), \\ stm_{id''+1} : signature(\mathcal{VT}(\vec{x}), list)\}, \\ \{stm_{id+1}[\mathcal{VT}(\vec{x})] \rightarrow stm_{id+2}[\mathcal{VT}(\vec{x})]\ [\mathcal{BT}(b)]\} \cup \mathcal{R}_1 \cup \\ \{stm_{id''}[\mathcal{VT}(\vec{x} \cdot \vec{y})] \rightarrow stm_{id+1}[\mathcal{VT}(\vec{x})]\ [true] \\ stm_{id+1}[\mathcal{VT}(\vec{x})] \rightarrow stm_{id''+1}[\mathcal{VT}(\vec{x})]\ [not(\mathcal{BT}(b))]\} \end{pmatrix}, \vec{x}, id'' + 1\right)}$$

1. Showing that $\forall x'_i \in \vec{x'} : x'_i \in Dom(s')$ holds:
   From the translation function it follows that $\vec{x'} = \vec{x}$.
   From the derivation tree it follows that $s' = s$.
   From our assumptions about the initial state s and initial vector of variables it follows that $\forall x_i \in \vec{x} : x_i \in Dom(s)$.
   That means (1) holds.

Now, we need to show that there exists a vector $\vec{v'} \in \overrightarrow{\mathcal{T}erms(\Sigma_{theory} \cup \Sigma_{list})}$, such that the starting term $stm_{id+1}[\vec{v}]$ is reducible to the normal form $stm_{id'}[\vec{v'}]$, and this vector of values v' represents the final state s' in the derivation tree.

To show this, we need to prove that the following statements hold:

2. Showing that $stm_{id+1}[\vec{v}] \to^*_{\mathcal{R}_{list} \cup \mathcal{R}_S} stm_{id'}[\vec{v'}]$:
   From the translation function, it follows that $id' = id'' + 1$.
   From the derivation tree it follows that $\mathcal{B}[\![b]\!]s = \bot$.
   From our assumptions about the vectors $\vec{x}$, $\vec{v}$, and initial state s, it follows that:

   $$\forall i \in \{0, |\vec{x}| - 1\} : s\ x_i = [\![v_i]\!]_{combi}$$

   That means when we apply the rule:

   $$stm_{id+1}[\mathcal{VT}(\vec{x})] \to stm_{id''+1}[\mathcal{VT}(\vec{x})]\ [not(\mathcal{BT}(b))]$$

   to the initial term $stm_{id+1}[\vec{v}]$, the rule constraint evaluates to $\top$:

   $$[\![not(\mathcal{BT}(b))\sigma]\!] = \top$$

   Where σ is a substitution defined as:

   $$\forall x_i \in \vec{x} : \sigma(\mathcal{VT}(x_i)) = v_i$$

   That means we can apply this rule and obtain the term $stm_{id''+1}[\vec{v'}] = stm_{id'}[\vec{v'}]$.
   We can conclude that (2) holds.
   We have proven that there exists such a vector $\vec{v'}$ if we choose the vector $\vec{v}$ as it.
3. Showing that $stm_{id'}[\vec{v'}] \in NF(\to_{\mathcal{R}_{list} \cup \mathcal{R}_S})$:
   Because all elements of the vector $\vec{v'}$ are in normal form (according to the assumption about the vector $\vec{v}$) and according to Remark 1: (3) holds.
4. Showing that $(|\vec{x'}| = |Dom(s')| = |\vec{v'}|)$ holds:
   From the translation function, it follows that $\vec{x'} = \vec{x}$.
   From the derivation tree, it follows that $s' = s$.
   Because of the properties of the set of LCTRS rules $\mathcal{R}_S$, we know that: $\vec{v'} = \vec{v}$.
   From our assumptions about the initial state s and initial vector of variables $\vec{x}$, it follows that $|\vec{x}| = |Dom(s)| = |\vec{v}|$.
   That means (4) holds.
5. Showing that $\forall i \in \{0, |\vec{x'}| - 1\} : s'\ x'_i = [\![v'_i]\!]_{combi}$ holds:
   From the translation function, it follows that $\vec{x'} = \vec{x}$.
   From the derivation tree, it follows that $s' = s$.
   Because of the properties of the set of LCTRS rules $\mathcal{R}_S$, we know that: $\vec{v'} = \vec{v}$.
   From our assumptions about the initial state s and vectors $\vec{x}$, $\vec{v}$, it follows that:

   $$\forall i \in \{0, |\vec{x}| - 1\} : s\ x_i = [\![v_i]\!]_{combi}$$

   That means (5) holds.

We proved that $Soundness(S, s, \vec{x'}, \vec{v}, \mathcal{R}_S, id + 1, id')$ holds in this case.

6.3.7 Case [comp_{ns}]

That means $S = (S_1; S_2)$, and the composite derivation tree has the following shape:

$$\dfrac{\dfrac{\boxed{\text{T1}}}{\langle S_1, s\rangle \to s''}\,[\dots] \qquad \dfrac{\boxed{\text{T2}}}{\langle S_2, s''\rangle \to s'}\,[\dots]}{\langle S_1; S_2, s\rangle \to s'}\,[\text{comp}_{\text{ns}}]$$

Consider the application of the translation function $\mathcal{T}$ to the statement S, initial vector of variables $\vec{x}$, and the initial identifier id:

$$\dfrac{[\![S_1]\!]_{\vec{x},id} = \left(\begin{pmatrix}\Sigma_{terms_1},\\ \mathcal{R}_1\end{pmatrix}, \vec{x}\cdot\vec{y}, id_1\right) \qquad [\![S_2]\!]_{\vec{x}\cdot\vec{y},id_1} = \left(\begin{pmatrix}\Sigma_{terms_2},\\ \mathcal{R}_2\end{pmatrix}, \vec{x}\cdot\vec{y}\cdot\vec{z}, id'\right)}{[\![S_1\,;\ S_2]\!]_{\vec{x},id} = \left(\begin{pmatrix}\Sigma_{terms_1}\cup\Sigma_{terms_2},\\ \mathcal{R}_1\cup\\ \{stm_{id_1}[\mathcal{VT}(\vec{x}\cdot\vec{y})] \to stm_{id_1+1}[\mathcal{VT}(\vec{x}\cdot\vec{y})]\ [true]\}\cup\\ \mathcal{R}_2\end{pmatrix}, \vec{x}\cdot\vec{y}\cdot\vec{z}, id'\right)}$$

Induction hypothesis (IH1):

Let's assume that $Soundness(S_1, s, \vec{x}\cdot\vec{y}, \vec{v}, \mathcal{R}_1, id+1, id_1)$ holds.

Because we assumed that the last applied rule was [comp_{ns}], there exists a subtree with conclusion $\langle S_1, s\rangle \to s''$.

That means the premise of the *Soundness*' implication holds, and we know that:

1. $\forall var_i \in v\vec{ar} : var_i \in Dom(s'')$, where $v\vec{ar} = \vec{x}\cdot\vec{y}$
2. $\exists \vec{v''} \in \overrightarrow{\mathcal{T}erms(\Sigma_{theory}\cup\Sigma_{list})}$:
   - $stm_{id+1}[\vec{v}] \to^*_{\mathcal{R}_{list}\cup\mathcal{R}_1} stm_{id_1}[\vec{v''}]$
   - $stm_{id_1}[\vec{v''}] \in NF(\to_{\mathcal{R}_{list}\cup\mathcal{R}_1})$
   - $|v\vec{ar}| = |Dom(s'')| = |\vec{v''}|$
   - $\forall i \in \{0, |v\vec{ar}|-1\} : s''\ var_i = [\![v''_i]\!]_{combi}$

Induction hypothesis (IH2):

Let's assume that $Soundness(S_2, s'', \vec{x}\cdot\vec{y}\cdot\vec{z}, \vec{v''}, \mathcal{R}_2, id_1+1, id')$ holds.

Because we assumed that the last applied rule was [comp_{ns}], there exists a subtree with conclusion $\langle S_2, s''\rangle \to s'$.

That means the premise of the *Soundness*' implication holds, and we know that:

1. $\forall var'_i \in \vec{var'} : var'_i \in Dom(s')$, where $\vec{var'} = \vec{x}\cdot\vec{y}\cdot\vec{z}$
2. $\exists \vec{v'} \in \overrightarrow{\mathcal{T}erms(\Sigma_{theory}\cup\Sigma_{list})}$:
   - $stm_{id_1+1}[\vec{v''}] \to^*_{\mathcal{R}_{list}\cup\mathcal{R}_2} stm_{id'}[\vec{v'}]$
   - $stm_{id'}[\vec{v'}] \in NF(\to_{\mathcal{R}_{list}\cup\mathcal{R}_2})$
   - $|\vec{var'}| = |Dom(s')| = |\vec{v'}|$
   - $\forall i \in \{0, |\vec{var'}|-1\} : s'\ var'_i = [\![v'_i]\!]_{combi}$

Using the information from the **IH1** and **IH2**, we need to prove the following:

1. Showing that $\forall x'_i \in \vec{x'} : x'_i \in Dom(s')$ holds:
   From the translation function definition, it follows that $\vec{x'} = \vec{x} \cdot \vec{y} \cdot \vec{z}$.
   From the **IH2** it follows that $\forall var'_i \in \vec{var'} : var'_i \in Dom(s')$, where $\vec{var'} = \vec{x} \cdot \vec{y} \cdot \vec{z}$.
   That means (1) holds.

Now, we need to show that there exists a vector $\vec{v'} \in \overrightarrow{\mathcal{T}erms(\Sigma_{theory} \cup \Sigma_{list})}$, such that the starting term $stm_{id+1}[\vec{v}]$ is reducible to the normal form $stm_{id'}[\vec{v'}]$, and this vector of values v' represents the final state s' in the derivation tree.

To show this, we need to prove that the following statements hold:

2. Showing that $stm_{id+1}[\vec{v}] \to^*_{\mathcal{R}_{list} \cup \mathcal{R}_S} stm_{id'}[\vec{v'}]$:
   From the **IH1** we know that $stm_{id+1}[\vec{v}] \to^*_{\mathcal{R}_{list} \cup \mathcal{R}_1} stm_{id_1}[\vec{v''}]$. That means, using only the rules from $\mathcal{R}_{list} \cup \mathcal{R}_1$ we are able to reduce the term $stm_{id+1}[\vec{v}]$ to the term $stm_{id_1}[\vec{v''}]$.
   Consider the LCTRS rule:
   $$stm_{id_1}[\mathcal{VT}(\vec{x} \cdot \vec{y})] \to stm_{id_1+1}[\mathcal{VT}(\vec{x} \cdot \vec{y})]\ [true]$$
   If we apply this rule to the term $stm_{id_1}[\vec{v''}]$, we obtain the following term: $stm_{id_1+1}[\vec{v''}]$.
   From the **IH2** we know that:
   $$stm_{id_1+1}[\vec{v''}] \to^*_{\mathcal{R}_{list} \cup \mathcal{R}_2} stm_{id'}[\vec{v'}]$$
   That means, using only the rules from $\mathcal{R}_{list} \cup \mathcal{R}_2$ we are able to reduce the term $stm_{id_1+1}[\vec{v''}]$ to the term $stm_{id'}[\vec{v'}]$.
   That means, for the set of the LCTRS rules $\mathcal{R}_{list} \cup \mathcal{R}_S$ we have the following reduction:
   $$stm_{id+1}[\vec{v}] \to^*_{\mathcal{R}_{list} \cup \mathcal{R}_S} stm_{id_1}[\vec{v''}] \to_{\mathcal{R}_{list} \cup \mathcal{R}_S} stm_{id_1+1}[\vec{v''}] \to^*_{\mathcal{R}_{list} \cup \mathcal{R}_S} stm_{id'}[\vec{v'}]$$
   From this, it follows that there exists the following reduction:
   $$stm_{id+1}[\vec{v}] \to^*_{\mathcal{R}_{list} \cup \mathcal{R}_S} stm_{id'}[\vec{v'}]$$
   That means (2) holds.
3. Showing that $stm_{id'}[\vec{v'}] \in NF(\to_{\mathcal{R}_{list} \cup \mathcal{R}_S})$:
   Because all elements of the vector $\vec{v'}$ are in normal form (according to the **IH2**) and according to Remark 1: (3) holds.
4. Showing that $(|\vec{x'}| = |Dom(s')| = |\vec{v'}|)$ holds:
   From the translation function definition, it follows that $\vec{x'} = \vec{x} \cdot \vec{y} \cdot \vec{z}$.
   From the **IH2** it follows that $|\vec{var'}| = |Dom(s')| = |\vec{v'}|$, where $\vec{var'} = \vec{x} \cdot \vec{y} \cdot \vec{z}$.
   That means (4) holds.
5. Showing that $\forall i \in \{0, |\vec{x'}| - 1\} : s'\ x'_i = [\![v'_i]\!]_{combi}$ holds:
   From the translation function definition it follows that $\vec{x'} = \vec{x} \cdot \vec{y} \cdot \vec{z}$.
   From the **IH2** it follows that $\forall i \in \{0, |\vec{var'}| - 1\} : s'\ var'_i = [\![v'_i]\!]_{combi}$, where $\vec{var'} = \vec{x} \cdot \vec{y} \cdot \vec{z}$.
   That means (5) holds.

We proved that $Soundness(S, s, \vec{x'}, \vec{v}, \mathcal{R}_S, id + 1, id')$ holds in this case.

6.3.8 Case $[\mathbf{if}_{\mathbf{ns}}^{\top}]$

That means $S = (\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2)$, $\mathcal{B}[\![b]\!]s = \top$, and the composite derivation tree has the following shape:

$$\dfrac{\dfrac{\boxed{\text{T1}}}{\langle S_1, s\rangle \to s''}\,[\dots]}{\langle \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, s\rangle \to \mathit{LeaveScope}(s, s'')}\,[\text{if}_{\text{ns}}^{\top}]$$

Because in this case we assumed the rule $[\text{if}_{\text{ns}}^{\top}]$ was applied, we can conclude that $\mathcal{B}[\![b]\!]s = \top$.

From the derivation tree, it follows that $s' = \mathit{LeaveScope}(s, s'')$.

Consider the application of the translation function $\mathcal{T}$ to the statement S, initial vector of variables $\vec{x}$, and the initial identifier id:

$$\dfrac{[\![S_1]\!]_{\vec{x},id+1} = \left(\begin{pmatrix}\Sigma_{terms_1},\\ \mathcal{R}_1\end{pmatrix}, \vec{x}\cdot\vec{y}, id'''\right) \qquad [\![S_2]\!]_{\vec{x},id'''} = \left(\begin{pmatrix}\Sigma_{terms_2},\\ \mathcal{R}_2\end{pmatrix}, \vec{x}\cdot\vec{z}, id''\right)}{[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]_{\vec{x},id} = \left(\begin{pmatrix}\Sigma_{terms_1}\cup\Sigma_{terms_2}\cup\\ \{stm_{id+1} : \mathit{signature}(\mathcal{VT}(\vec{x}), \mathit{list}),\\ stm_{id''+1} : \mathit{signature}(\mathcal{VT}(\vec{x}), \mathit{list})\},\\ \{stm_{id+1}[\mathcal{VT}(\vec{x})] \to stm_{id+2}[\mathcal{VT}(\vec{x})]\ [\mathcal{BT}(b)]\}\cup\mathcal{R}_1\cup\\ \{stm_{id+1}[\mathcal{VT}(\vec{x})] \to stm_{id'''+1}[\mathcal{VT}(\vec{x})]\ [\mathit{not}(\mathcal{BT}(b))]\}\cup\mathcal{R}_2\cup\\ \{stm_{id'''}[\mathcal{VT}(\vec{x}\cdot\vec{y})] \to stm_{id''+1}[\mathcal{VT}(\vec{x})]\ [\mathit{true}],\\ stm_{id''}[\mathcal{VT}(\vec{x}\cdot\vec{z})] \to stm_{id''+1}[\mathcal{VT}(\vec{x})]\ [\mathit{true}]\}\end{pmatrix}, \vec{x}, id''+1\right)}$$

Induction hypothesis (IH):

Let's assume that $\mathit{Soundness}(S_1, s, \vec{x}\cdot\vec{y}, \vec{v}, \mathcal{R}_1, id+1, id''')$ holds.

Because we assumed that the last applied rule was $[\text{if}_{\text{ns}}^{\top}]$, there exists a subtree with conclusion $\langle S_1, s\rangle \to s''$.

That means the premise of the *Soundness*' implication holds, and we know that:

1. $\forall var_i \in v\vec{a}r : var_i \in \mathit{Dom}(s'')$, where $v\vec{a}r = \vec{x}\cdot\vec{y}$
2. $\exists \vec{v''} \in \overrightarrow{\mathcal{T}erms(\Sigma_{theory}\cup\Sigma_{list})}$:
   - $stm_{id+2}[\vec{v}] \to^*_{\mathcal{R}_{list}\cup\mathcal{R}_1} stm_{id'''}[\vec{v''}]$
   - $stm_{id'''}[\vec{v''}] \in NF(\to_{\mathcal{R}_{list}\cup\mathcal{R}_1})$
   - $|v\vec{a}r| = |\mathit{Dom}(s'')| = |\vec{v''}|$
   - $\forall i \in \{0, |v\vec{a}r| - 1\} : s''\ var_i = [\![v_i'']\!]_{combi}$

Using the information from the **IH** we need to prove the following:

1. Showing that $\forall x_i' \in \vec{x'} : x_i' \in \mathit{Dom}(s')$ holds:
   From the translation function definition, it follows that $\vec{x'} = \vec{x}$.
   The derivation tree shows that $s' = \mathit{LeaveScope}(s, s'')$. That means: $\mathit{Dom}(s') = \mathit{Dom}(s)$.
   From our assumptions about the state s and the initial vector of variables $\vec{x}$ it follows that: $\forall x_i \in \vec{x} : x_i \in \mathit{Dom}(s)$.
   That means (1) holds.

Now, we need to show that there exists a vector $\vec{v'} \in \overrightarrow{\mathcal{T}erms(\Sigma_{theory} \cup \Sigma_{list})}$, such that the starting term $stm_{id+1}[\vec{v}]$ is reducible to the normal form $stm_{id'}[\vec{v'}]$, and this vector of values v' represents the final state s' in the derivation tree.

To show this, we need to prove that the following statements hold:

2. Showing that $stm_{id+1}[\vec{v}] \to^*_{\mathcal{R}_{list} \cup \mathcal{R}_S} stm_{id'}[\vec{v'}]$ holds:
   From the derivation tree it follows that $\mathcal{B}[\![b]\!]s = \top$.
   From our assumptions about vectors $\vec{x}$, $\vec{v}$, and the initial state s, we know that: $\forall i \in \{0, |\vec{x}| - 1\} : s\ x_i = [\![v_i]\!]_{combi}$.
   That means when we apply the rule:

   $$stm_{id+1}[\mathcal{VT}(\vec{x})] \to_{\mathcal{R}_S} stm_{id+2}[\mathcal{VT}(\vec{x})]\ [\mathcal{BT}(b)]$$

   to the initial term $stm_{id+1}[\vec{v}]$, the rule constraint evaluates to $\top$:

   $$[\![\mathcal{BT}(b)\sigma]\!] = \top$$

   Where σ is a substitution defined as:

   $$\forall x_i \in \vec{x} : \sigma(\mathcal{VT}(x_i)) = v_i$$

   That means we can apply this rule and obtain the term $stm_{id+2}[\vec{v}]$
   From **IH**, we know that $stm_{id+2}[\vec{v}] \to^*_{\mathcal{R}_1} stm_{id'''}[\vec{v''}]$. That means, using only the rules from $\mathcal{R}_1$ we will be able to reduce the term $stm_{id+2}[\vec{v}]$ to the term $stm_{id'''}[\vec{v''}]$.
   Consider the term $stm_{id'''}[\vec{v''}]$: in the set of LCTRS rules $\mathcal{R}_S$ there exists the rule:

   $$stm_{id'''}[\mathcal{VT}(\vec{x} \cdot \vec{y})] \to_{\mathcal{R}_S} stm_{id''+1}[\mathcal{VT}(\vec{x})]\ [true]$$

   That means by applying this rule to the term, we can reduce $stm_{id'''}[\vec{v''}]$ to $stm_{id'}[\vec{v'}]$.
   That means (2) holds.
3. Showing that $stm_{id'}[\vec{v'}] \in NF(\to_{\mathcal{R}_{list} \cup \mathcal{R}_S})$ holds:
   Because all elements of the vector $\vec{v'}$ are in normal form (according to the **IH**) and according to Remark 1: (3) holds.
4. Showing that $(|\vec{x'}| = |Dom(s')| = |\vec{v'}|)$ holds:
   From the translation function definition, it follows that $\vec{x'} = \vec{x}$.
   The derivation tree shows that $s' = LeaveScope(s, s'')$. That means: $Dom(s') = Dom(s)$.
   Because we can obtain the term $stm_{id'}[\vec{v'}]$ only by applying the rule:

   $$stm_{id'''}[\mathcal{VT}(\vec{x} \cdot \vec{y})] \to_{\mathcal{R}_S} stm_{id''+1}[\mathcal{VT}(\vec{x})]\ [true]$$

   It holds that $|\vec{v'}| = |\vec{x}|$.
   From the assumptions about the state s, the initial vector of variables $\vec{x}$ and the initial vector of values, it holds that: $|\vec{x}| = |Dom(s)| = |\vec{v}|$.
   That means (4) holds.
5. Showing that $\forall i \in \{0, |\vec{x'}| - 1\} : s'\ x'_i = [\![v'_i]\!]_{combi}$ holds:
   From the translation function definition, it follows that $\vec{x'} = \vec{x}$.

From the **IH** it follows that:

$$\forall i \in \{0, |\vec{var}| - 1\} : s'' \; var_i = [\![v_i'']\!]_{combi}$$

Where $\vec{var} = \vec{x} \cdot \vec{y}$.
Because we can obtain the term $stm_{id'}[\vec{v'}]$ only by applying the rule:

$$stm_{id'''}[\vec{x} \cdot \vec{y}] \rightarrow_{\mathcal{R}_S} stm_{id''+1}[\vec{x}] \; [true]$$

It holds that the value vector $\vec{v'}$ is just the vector $\vec{v''}$ with removed last $|\vec{y}|$ elements.
That means:

$$\forall i \in \{0, |\vec{x'}| - 1\} : s'' \; x_i' = [\![v_i']\!]_{combi}$$

The derivation tree shows that $s' = LeaveScope(s, s'')$. That means: $Dom(s') = Dom(s)$ and $\forall x_i \in \vec{x} : s' \; x_i = s'' \; x_i$.
We can conclude that (5) holds.

We proved that $Soundness(S, s, \vec{x'}, \vec{v}, \mathcal{R}_S, id + 1, id')$ holds in this case.

6.3.9 Case $[\mathbf{if}^{\perp}_{\mathbf{ns}}]$

The proof is identical to the previous case.

6.3.10 Case $[\mathbf{while}^{\top}_{\mathbf{ns}}]$

That means $S = (\texttt{while}\ b\ \texttt{do}\ S)$, $\mathcal{B}[\![b]\!]s = \top$, and the composite derivation tree has the following shape:

$$\dfrac{\dfrac{\boxed{\text{T1}}}{\langle S, s\rangle \rightarrow s''}\,[\dots] \qquad \dfrac{\boxed{\text{T2}}}{\langle \texttt{while}\ b\ \texttt{do}\ S, LeaveScope(s, s'')\rangle \rightarrow s'''}\,[\dots]}{\langle \texttt{while}\ b\ \texttt{do}\ S, s\rangle \rightarrow LeaveScope(s, s''')}\,[\text{while}^{\top}_{\text{ns}}]$$

From the derivation tree, it follows that $s' = LeaveScope(s, s''')$.

Consider the application of the translation function $\mathcal{T}$ to the statement S, initial vector of variables $\vec{x}$, and the initial identifier id:

$$[\![S]\!]_{\vec{x},id+1} = \left(\begin{pmatrix}\Sigma_{terms_1},\\ \mathcal{R}_1\end{pmatrix}, \vec{x} \cdot \vec{y}, id'\right)$$

$$[\![\texttt{while}\ b\ \texttt{do}\ S]\!]_{\vec{x},id} = \left(\begin{pmatrix}\Sigma_{terms_1} \cup \\ \{stm_{id+1} : signature(\mathcal{VT}(\vec{x}), list),\\ stm_{id'+1} : signature(\mathcal{VT}(\vec{x}), list)\},\\ \{stm_{id+1}[\mathcal{VT}(\vec{x})] \rightarrow stm_{id+2}[\mathcal{VT}(\vec{x})]\ [\mathcal{BT}(b)]\} \cup \mathcal{R}_1 \cup \\ \{stm_{id'}[\mathcal{VT}(\vec{x} \cdot \vec{y})] \rightarrow stm_{id+1}[\mathcal{VT}(\vec{x})]\ [true]\\ stm_{id+1}[\mathcal{VT}(\vec{x})] \rightarrow stm_{id'+1}[\mathcal{VT}(\vec{x})]\ [not(\mathcal{BT}(b))]\}\end{pmatrix}, \vec{x}, id' + 1\right)$$

According to the translation function, it follows that $\vec{x'} = \vec{x}$.

Induction hypothesis (IH1):

Let's assume that $Soundness(S, s, \vec{x} \cdot \vec{y}, \vec{v}, \mathcal{R}_1, id + 2, id')$ holds.

Because we assumed that the last applied rule was $[\text{while}^{\top}_{\text{ns}}]$, there exists a subtree with the conclusion: $\langle S, s\rangle \to s''$.

That means the premise of the *Soundness*' implication holds, and we know that:

1. $\forall var_i \in v\vec{a}r : var_i \in Dom(s'')$, where $v\vec{a}r = \vec{x} \cdot \vec{y}$
2. $\exists \vec{v''} \in \overrightarrow{\mathcal{T}erms(\Sigma_{theory} \cup \Sigma_{list})}$:
   - $stm_{id+2}[\vec{v}] \to^*_{\mathcal{R}_{list} \cup \mathcal{R}_1} stm_{id'}[\vec{v''}]$
   - $stm_{id'}[\vec{v''}] \in NF(\to_{\mathcal{R}_{list} \cup \mathcal{R}_1})$
   - $|v\vec{a}r| = |Dom(s'')| = |\vec{v''}|$
   - $\forall i \in \{0, |v\vec{a}r| - 1\} : s''\ var_i = [\![v''_i]\!]_{combi}$

Induction hypothesis (IH2):

Let's assume that :

$Soundness((\texttt{while}\ b\ \texttt{do}\ S), LeaveScope(s, s''), \vec{x}, \vec{v''}, \mathcal{R}_1, id + 1, id' + 1)$ holds.

Because we assumed that the last applied rule was $[\text{while}^{\top}_{\text{ns}}]$, there exists a subtree with conclusion $\langle \texttt{while}\ b\ \texttt{do}\ S, LeaveScope(s, s'')\rangle \to s'''$.

That means the premise of the *Soundness*' implication holds, and we know that:

1. $\forall var'_i \in \vec{var'} : var'_i \in Dom(s''')$, where $\vec{var'} = \vec{x}$
2. $\exists \vec{v'} \in \overrightarrow{\mathcal{T}erms(\Sigma_{theory} \cup \Sigma_{list})}$:
   - $stm_{id+1}[\vec{v''}] \to^*_{\mathcal{R}_{list} \cup \mathcal{R}_S} stm_{id'+1}[\vec{v'}]$
   - $stm_{id'+1}[\vec{v'}] \in NF(\to_{\mathcal{R}_{list} \cup \mathcal{R}_S})$
   - $|\vec{var'}| = |Dom(s''')| = |\vec{v'}|$
   - $\forall i \in \{0, |\vec{var'}| - 1\} : s'''\ var'_i = [\![v'_i]\!]_{combi}$

Using the information from the **IH1** and **IH2** we need to prove the following:

1. Showing that $\forall x'_i \in \vec{x'} : x'_i \in Dom(s')$ holds:
   According to the definition of the function *LeaveScope* it follows that $Dom(s') = Dom(s)$.
   Also, we know that $\vec{x'} = \vec{x}$.
   From our assumptions about the initial state s and initial vector of variables $\vec{x}$ we know that: $\forall x_i \in \vec{x} : x_i \in Dom(s)$.
   That means for the state s' and vector $\vec{x'}$ the following holds: $\forall x'_i \in \vec{x'} : x'_i \in Dom(s')$.
   And that means (1) holds.

Now, we need to show that there exists a vector $\vec{v'} \in \overrightarrow{\mathcal{T}erms(\Sigma_{theory} \cup \Sigma_{list})}$, such that the starting term $stm_{id+1}[\vec{v}]$ is reducible to the normal form $stm_{id'}[\vec{v'}]$, and this vector of values v' represents the final state s' in the derivation tree.

To show this, we need to prove that the following statements hold:

2. Showing that $stm_{id+1}[\vec{v}] \to^*_{\mathcal{R}_{list} \cup \mathcal{R}_S} stm_{id'+1}[\vec{v'}]$ holds:
   From the derivation tree it follows that $\mathcal{B}[\![b]\!]s = \top$.
   From our assumptions about vectors $\vec{x}$, $\vec{v}$, and the initial state s, we know that:

$$\forall i \in \{0, |\vec{x}| - 1\} : s\ x_i = [\![v_i]\!]_{combi}$$

That means when we apply the rule:

$$stm_{id+1}[\vec{x}] \to stm_{id+2}[\vec{x}]\ [\mathcal{BT}(b)]$$

to the initial term $stm_{id+1}[\vec{v}]$, the rule constraint evaluates to $\top$:

$$[\![\mathcal{BT}(b)\sigma]\!] = \top$$

Where σ is a substitution defined as:

$$\forall x_i \in \vec{x} : \sigma(\mathcal{VT}(x_i)) = v_i$$

That means we can apply this rule and obtain the term $stm_{id+2}[\vec{v}]$.
According to the **IH1** we know that $stm_{id+2}[\vec{v}] \to^*_{\mathcal{R}_{list} \cup \mathcal{R}_1} stm_{id'}[\vec{v'''}]$.
That means, using the rules from $\mathcal{R}_{list} \cup \mathcal{R}_1$ we will be able to reduce the term $stm_{id+1}[\vec{v}]$ to the term $stm_{id'}[\vec{v'''}]$.
Consider the term $stm_{id'}[\vec{v'''}]$: in the set of LCTRS rules $\mathcal{R}_S$ there exists the rule:

$$stm_{id'}[\mathcal{VT}(\vec{x} \cdot \vec{y})] \to stm_{id+1}[\mathcal{VT}(\vec{x})]\ [true]$$

That means, by applying this rule to the term $stm_{id'}[\vec{v'''}]$, we can reduce it to $stm_{id+1}[\vec{v''}]$.
According to the **IH2** we know that $stm_{id+1}[\vec{v''}] \to^*_{\mathcal{R}_{list} \cup \mathcal{R}_S} stm_{id'+1}[\vec{v'}]$.
That means, using the rules from $\mathcal{R}_{list} \cup \mathcal{R}_S$ we will be able to reduce the term $stm_{id+1}[\vec{v''}]$ to the term $stm_{id'+1}[\vec{v'}]$.
That means (2) holds.

3. Showing that $stm_{id'}[\vec{v'}] \in NF(\to_{\mathcal{R}_{list} \cup \mathcal{R}_S})$ holds:
   Because all elements of the vector $\vec{v'}$ are in normal form (according to the **IH2**) and according to Remark 1: (3) holds.
4. Showing that $(|\vec{x'}| = |Dom(s')| = |\vec{v'}|)$ holds:
   We know that $\vec{x'} = \vec{x}$.
   Also, we know that $s' = LeaveScope(s, s''')$, which means: $Dom(s') = Dom(s)$.
   From our assumptions about the initial state s and initial vector of variables $\vec{x}$ we know that: $|\vec{x}| = |Dom(s)| = |\vec{v}|$. That means: $|\vec{x'}| = |Dom(s')|$.
   We can obtain the term $stm_{id'+1}[\vec{v'}]$ only by applying the rule:

$$stm_{id+1}[\mathcal{VT}(\vec{x})] \to stm_{id'+1}[\mathcal{VT}(\vec{x})]\ [not(\mathcal{BT}(b))]$$

   That means: $|\vec{x'}| = |\vec{v'}|$.
   From this it follows that: $|\vec{x'}| = |Dom(s')| = |\vec{v'}|$.
   We proved that (4) holds.
5. Showing that $\forall i \in \{0, |\vec{x'}| - 1\} : s'\ x'_i = [\![v'_i]\!]_{combi}$ holds:
   We know that $\vec{x'} = \vec{x}$.
   From the **IH2** it follows that $\forall i \in \{0, |\vec{var'}| - 1\} : s'''\ var'_i = [\![v'_i]\!]_{combi}$, where $\vec{var'} = \vec{x}$.
   That means $\vec{v'}$ represents the values of the variables from $\vec{x'}$ in the state s'''.
   The derivation tree shows that $s' = LeaveScope(s, s''')$.
   That means $Dom(s') = Dom(s)$ and $\forall x_i \in \vec{x} : s'\ x_i = s'''\ x_i$.
   That means (5) holds.

We proved that $Soundness(S, s, \vec{x'}, \vec{v}, \mathcal{R}_S, id + 1, id')$ holds in this case.

We proved that the **soundness** property holds for the arbitrary While++ program S. □

Chapter 7

Implementation

This section covers the description of the project `WPP2TRS`[1] - the Java implementation of the translation process that allows converting imperative programs into the Logically Constrained Term rewriting systems described in Chapter 5. Specifically for this project, the parser for the While++ programming language and the converter were implemented. This converter allows the transformation of the intermediate class representation of the While++ program into the logically constrained TRS. The resulting term rewriting system is compatible with `cora` [3] - an open-source tool for constrained higher-order term rewriting.

7.1 ANTLR Grammar

The parser for the While++ programming language is implemented using the `ANTLR` [14] parser generator. The grammar for While++ defined in Section 4.1 is adapted to use in `ANTLR`. For instance, one difference between the theoretical definition of the While++ grammar and the practical implementation in the `WPP2TRS` project is that in the implementation, the explicit keywords are replaced with the lexer rules. Also, the explicit program statement definitions are grouped into specific parser rules. Finally, the rules for arithmetic and boolean expressions are placed in a specific order to ensure the correct parsing of the infix notation.

Example 48. Part of the program statements grammar implemented in `ANTLRv4` for `WPP2TRS` project:

```
ASSIGN: ':=' ;
SKIP_STM: 'skip' ;
PRINT_INT_STM: 'printInt' ;
...
INT_TYPE: 'int' ;
BOOL_TYPE: 'bool' ;
STRING_TYPE: 'string' ;
...
LPAREN: '(' ;
RPAREN: ')' ;
...
stm: unaryStm | blockStm | stm SEMICOLON stm;
```

[1] `https://github.com/mikhirurg/WPP2TRS`

```
nestedStm: unaryStm | blockStm ;
condition: bexp ;
unaryStm: skipStm
        | declareStm
        | printIntStm
        | printStringStm
        | printBoolStm
        | readIntStm
        | readStringStm
        | readBoolStm
        | ifStm
        | whileStm
        | assignStm
        ;
skipStm: SKIP_STM ;
printIntStm: PRINT_INT_STM LPAREN aexp RPAREN ;
...
blockStm: LPAREN stm RPAREN ;
declareStm: (INT_TYPE | BOOL_TYPE | STRING_TYPE) VAR ASSIGN expr ;
```

7.2 Class representations

`WPP2TRS` uses the special class representations for the parsed syntactic constructions of While++ programs. The class bindings exist for all syntactic constructions: arithmetic expressions, boolean expressions, strings, variables, and program statements. For the variables, there exist three different classes: `WhileIntVar` - for integer variables, `WhileBoolVar` - for boolean variables, and `WhileStringVar` - for string variables.

7.3 WhileListener (ParseTreeListener)

The abstract syntactic tree (AST) representation generated by the basic ANTLR parser from the While++ grammar is not very suitable for immediate translation of the program to the LCTRS. For example, the basic parser doesn't allow static type checks on the program variables. To clarify the translation process, we transform the syntactic constructions into the class representations described in Section 7.2. The special implementation of the parse tree listener, `WhileListener`, allows the transformation of the AST generated by `ANTLR` to the desired class entities.

The `WhileListener` can be used together with the `ParseTreeWalker` to perform the deep first search on the AST and construct the classes representing While++ constructions. The nested constructions are handled using stacks, which collect arithmetic expressions, boolean expressions, strings, and statements. Additionally, this parse tree listener keeps track of the variable declarations and the appropriate usage of variables. If in some type-dependent construction (such as `printInt`) there used a variable of the wrong type, then the `WhileListener` will throw the corresponding exception. If the processing of the AST is successful, then by calling the method `getProgram()`, we can access the class representation of the parsed While++ program.

7.4 ProgramBuilder

The utility class `ProgramBuilder` provides a simple interface for transforming the input While++ programs into the class representation 7.2. The method `parseProgram(String program)` parses the input program using the lexer and parser generated by `ANTLR`, and then uses the `ParseTreeWalker` to perform a deep first search on the AST. While walking through the syntax tree, the `WhileListener` is used to generate the class representation of the input program - the result of the method `parseProgram(String program)`.

7.5 TRS Printer

The class `TRSPrinter` generates the `cora`-compatible Logically Constrained TRS using the While++ class representations. The translation implementation is based on the definition of the translation function $\mathcal{T}$, described in Section 5.3.1. Because the translation function definition is recursive, this approach can be naturally implemented using the visitor pattern. Each class representing the While++ program statements has a special method `acceptTRSPrinter(TRSPrinter trsPrinter)`, which calls the appropriate visit method of the `TRSPrinter`. Finally, each visit method implements the translation logic related to a particular syntactic construction.

7.6 Example

In this section we will consider the example translation of the While++ program into the LCTRS.

Example 49. A simple primality test algorithm implemented in While++:

```
bool isPrime := true;
int i := 2;
int x := 0;
readInt(x);
while (i*i <= x and isPrime) do (
    if (x % i = 0) then
        isPrime := false
    else
        skip;
    i := i + 1
);
printBool(isPrime)
```

The resulting LCTRS generated by WPP2TRS:

```
nil :: list
consI :: Int -> list -> list
consB :: Bool -> list -> list
consS :: String -> list -> list

headI :: list -> Int
headB :: list -> Bool
headS :: list -> String

tailI :: list -> list
tailB :: list -> list
tailS :: list -> list
```

```
headI(consI(i, l)) -> i
headB(consB(b, l)) -> b
headS(consS(str, l)) -> str

tailI(consI(i, l)) -> l
tailB(consB(b, l)) -> l
tailS(consS(str, l)) -> l

stm_1 :: list -> list -> list
stm_2 :: list -> list -> Bool -> list
stm_3 :: list -> list -> Bool -> list
stm_4 :: list -> list -> Bool -> Int -> list
stm_5 :: list -> list -> Bool -> Int -> list
stm_6 :: list -> list -> Bool -> Int -> Int -> list
stm_7 :: list -> list -> Bool -> Int -> Int -> list
stm_8 :: list -> list -> Bool -> Int -> Int -> list
stm_9 :: list -> list -> Bool -> Int -> Int -> list
stm_10 :: list -> list -> Bool -> Int -> Int -> list
stm_11 :: list -> list -> Bool -> Int -> Int -> list
stm_13 :: list -> list -> Bool -> Int -> Int -> list
stm_12 :: list -> list -> Bool -> Int -> Int -> list
stm_14 :: list -> list -> Bool -> Int -> Int -> list
stm_15 :: list -> list -> Bool -> Int -> Int -> list
stm_16 :: list -> list -> Bool -> Int -> Int -> list
stm_17 :: list -> list -> Bool -> Int -> Int -> list
stm_18 :: list -> list -> Bool -> Int -> Int -> list
stm_19 :: list -> list -> Bool -> Int -> Int -> list
stm_20 :: list -> list -> Bool -> Int -> Int -> list

stm_1(in, out) -> stm_2(in, out, true) | true
stm_2(in, out, isPrime) -> stm_3(in, out, isPrime) | true
stm_3(in, out, isPrime) -> stm_4(in, out, isPrime, 2) | true
stm_4(in, out, isPrime, i) -> stm_5(in, out, isPrime, i) | true
stm_5(in, out, isPrime, i) -> stm_6(in, out, isPrime, i, 0) | true
stm_6(in, out, isPrime, i, x) -> stm_7(in, out, isPrime, i, x) | true
stm_7(in, out, isPrime, i, x) -> stm_8(tailI(in), out, isPrime, i, headI(in)) | true
stm_8(in, out, isPrime, i, x) -> stm_9(in, out, isPrime, i, x) | true
stm_9(in, out, isPrime, i, x) -> stm_10(in, out, isPrime, i, x) | (((i * i) <= x) /\ isPrime)
stm_10(in, out, isPrime, i, x) -> stm_11(in, out, isPrime, i, x) | ((x % i) = 0)
stm_11(in, out, isPrime, i, x) -> stm_12(in, out, false, i, x) | true
stm_10(in, out, isPrime, i, x) -> stm_13(in, out, isPrime, i, x) | not(((x % i) = 0))
stm_13(in, out, isPrime, i, x) -> stm_14(in, out, isPrime, i, x) | true
stm_12(in, out, isPrime, i, x) -> stm_15(in, out, isPrime, i, x) | true
stm_14(in, out, isPrime, i, x) -> stm_15(in, out, isPrime, i, x) | true
stm_15(in, out, isPrime, i, x) -> stm_16(in, out, isPrime, i, x) | true
stm_16(in, out, isPrime, i, x) -> stm_17(in, out, isPrime, (i + 1), x) | true
stm_17(in, out, isPrime, i, x) -> stm_9(in, out, isPrime, i, x) | true
stm_9(in, out, isPrime, i, x) -> stm_18(in, out, isPrime, i, x) | not((((i * i) <= x) /\ isPrime))
stm_18(in, out, isPrime, i, x) -> stm_19(in, out, isPrime, i, x) | true
stm_19(in, out, isPrime, i, x) -> stm_20(in, consB(isPrime, out), isPrime, i, x) | true
stm_20(in, out, isPrime, i, x) -> out | true
```

Chapter 8

Conclusion

In this Bachelor's thesis project, we studied the Logically Constrained Term Rewriting Systems. In particular, we developed the systematic approach of translating imperative programs into the Logically Constrained TRS. To abstract ourselves from the details of the implementation of the real programming languages, we designed a simple imperative programming language, While++ (Chapter 4). We described the syntax and the natural semantics of this programming language.

After that, we defined special translation functions that allow translating arbitrary While++ programs into the LCTRS.

Finally, we defined of equivalence of the While++ program and the Logically Constrained TRS generated using the translation function (Definition 6.2.1). Based on that definition, we formulated the correctness theorem for the translation function $\mathcal{T}$(Lemma 2) and proved the **soundness** part of that theorem (Section 6.3).

As a practical part of this project we developed the project `WPP2TRS` (Chapter 7): a Java implementation of the translation of the While++ programs into the LCTRS. The output LCTRS are compatible with the input format of the open-source tool for constrained higher-order term rewriting - `cora` [3].

Chapter 9

Future work

Considering the current thesis project, we only proved the **soundness** part of the translation function correctness theorem. To finalize the proof, it would be a good idea to prove the **completeness** of the theorem.

The current version of the While++ programming language supports only the basic features of the imperative programming languages. It would be interesting to extend While++ to support more complex data types, such as arrays and collections, in future work. Adding more complex language constructions, such as exceptions, functions, and lambda expressions, is also possible. Developing the translation for the listed language constructions into the LCTRS will contribute to analyzing modern imperative programming languages.

Appendix A

Proof of the Lemma 1

Proof. We will organize the proof as follows: first, we will prove by induction that the application of the translation function $\mathcal{T}$ to any While++ program will result in the following:

- The left-hand side of each rule in the generated list of LCTRS rules is linear
- For every pair of rules ρ_1, ρ_2 in the generated list of rules, every critical pair between ρ_1 and the variable-renamed copy of ρ_2 is trivial.

After that we will use this fact about the generated list of LCTRS rules to draw a conclusion about the orthogonality of the LCTRS generated by using the translation function $\mathcal{T}$.
First, let's consider the set of rules $\mathcal{R}_{list}$. We need to show that:

- The left-hand side of each rule in $\mathcal{R}_{list}$ is linear.
- For every pair of rules $\rho_1, \rho_2 \in \mathcal{R}_{list}$: every critical pair between ρ_1 and the variable-renamed copy of ρ_2 is trivial.

Let's consider each rule $\rho \in \mathcal{R}_{list}$ and verify whether if all of them are linear:

1. $R_1 : headI(consI(i, l)) \to i\ [true]$

   The left side of this rule is a term $headI(consI(i, l))$. It has only two different variables: variable i of sort int and variable l of sort $list$. That means the left side of the rule is linear.

2. $R_2 : headB(consB(b, l)) \to b\ [true]$

   The left side of this rule is a term $headB(consB(b, l))$. It has only two different variables: variable b of sort $bool$ and variable l of sort $list$. That means the left side of the rule is linear.

3. $R_3 : headS(consS(str, l)) \to str\ [true]$

   The left side of this rule is a term $headS(consS(str, l))$. It has only two different variables: variable str of sort $string$ and variable l of sort $list$. That means the left side of the rule is linear.

4. $R_4 : tail(consI(i, l)) \to l\ [true]$

   The left side of this rule is a term $tail(consI(i, l))$. It has only two different variables: variable i of sort int and variable l of sort $list$. That means the left side of the rule is linear.

5. $R_5 : tail(consB(b, l)) \to l\ [true]$

   The left side of this rule is a term $tail(consI(i, l))$. It has only two different variables: variable i of sort int and variable l of sort $list$. That means the left side of the rule is linear.

6. $R_6 : tail(consS(str, l)) \to l\ [true]$

   The left side of this rule is a term $tail(consB(str, l))$. It has only two different variables: variable str of sort $string$ and variable l of sort $list$. That means the left side of the rule is linear.

The left side of each rule in the set $\mathcal{R}_{list}$ is linear. Now, we need to show that all critical pairs generated from the set of rules $\mathcal{R}_{list}$ are trivial. We will consider all combinations $\rho_1 = l_1 \to r_1\ [\varphi_1], \rho_2 = l_2 \to r_2\ [\varphi_2]$ of LCTRS rules in $\mathcal{R}_{list}$. We will rename variables such that l_1 and l_2 have no variables in common. Then we will consider all non-variable subterms t of l_2 (possibly equal to l_2), For the subterm t and the term l_1, we will check whether they unify with some substitution σ. We will construct the critical pairs using the unifying terms t and l_1 and the rewriting rules.
Consider candidate critical pairs:

Rule pairs	l_1	l_2	t	σ	Candidate Critical Pair
$R_1 \times R_1$	$headI(consI(i_1, l_1))$	$headI(consI(i_2, l_2))$	$headI(consI(i_2, l_2))$	$\sigma(i_1) = i_2, \sigma(l_1) = l_1$ $\sigma(i_2) = i_1, \sigma(l_2) = l_1$	$\langle i_1, i_1, and(true, true)\rangle$
$R_2 \times R_2$	$headB(consB(b_1, l_1))$	$headB(consB(b_2, l_2))$	$headB(consB(b_2, l_2))$	$\sigma(b_1) = b_2, \sigma(l_1) = l_1$ $\sigma(b_2) = b_1, \sigma(l_2) = l_1$	$\langle b_1, b_1, and(true, true)\rangle$
$R_3 \times R_3$	$headS(consS(str_1, l_1))$	$headS(consS(str_2, l_2))$	$headS(consS(str_2, l_2))$	$\sigma(str_1) = str_1, \sigma(l_1) = l_1$ $\sigma(str_2) = str_1, \sigma(l_2) = l_1$	$\langle str_1, str_1, and(true, true)\rangle$
$R_4 \times R_4$	$tail(consI(i_1, l_1))$	$tail(consI(i_2, l_2))$	$tail(consI(i_2, l_2))$	$\sigma(i_1) = i_1, \sigma(l_1) = l_1$ $\sigma(i_2) = i_1, \sigma(l_2) = l_1$	$\langle l_1, l_1, and(true, true)\rangle$
$R_5 \times R_5$	$tail(consB(b_1, l_1))$	$tail(consB(b_2, l_2))$	$tail(consB(b_2, l_2))$	$\sigma(b_1) = b_1, \sigma(l_1) = l_1$ $\sigma(b_2) = b_1, \sigma(l_2) = l_1$	$\langle l_1, l_1, and(true, true)\rangle$
$R_6 \times R_6$	$tail(consS(str_1, l_1))$	$tail(consS(str_2, l_2))$	$tail(consS(str_2, l_2))$	$\sigma(str_1) = str_1, \sigma(l_1) = l_1$ $\sigma(str_2) = str_1, \sigma(l_2) = l_1$	$\langle l_1, l_1, and(true, true)\rangle$

All candidate critical pairs computed for the set of LCTRS rules $\mathcal{R}_{list}$ are trivial critical pairs.
We proved that the left-side of each rule in $\mathcal{R}_{list}$ is linear, and all critical pairs generated using the set $\mathcal{R}_{list}$ are trivial.
Now, we will consider the translation function $\mathcal{T}$. We will prove that the LCTRS generated using this function is weakly orthogonal. The proof will be based on the structural induction on the form of the statement S:

- $S = (\texttt{skip})$

  Consider the translation of the statement S with the initial vector of variables $\vec{x}$ and the initial identifier id:

$$[\![\texttt{skip}]\!]_{\vec{x},id} = \begin{pmatrix} \begin{pmatrix} \{stm_{id+1} : signature(\mathcal{VT}(\vec{x}), list), \\ stm_{id+2} : signature(\mathcal{VT}(\vec{x}), list)\}, \\ \{stm_{id+1}[\mathcal{VT}(\vec{x})] \to stm_{id+2}[\mathcal{VT}(\vec{x})]\ [true]\} \end{pmatrix}, \\ \vec{x}, \\ id + 2 \end{pmatrix}$$

  Now, let's consider the set of LCTRS rules generated by the translation procedure:

$$\mathcal{R}_{\text{skip}} = \{stm_{id+1}[\mathcal{VT}(\vec{x})] \to stm_{id+2}[\mathcal{VT}(\vec{x})]\ [true]\}$$

  We need to show that:

  – The left-hand side of each rule is linear.

– For every pair of rules $\rho_1, \rho_2 \in \mathcal{R}_{list} \cup \mathcal{R}_{skip}$: every critical pair between ρ_1 and the variable-renamed copy of ρ_2 is trivial.

Consider the only rule in set of the LCTRS rules $\mathcal{R}_{skip}$:

$$R : stm_{id+1}[\mathcal{VT}(\vec{x})] \to stm_{id+2}[\mathcal{VT}(\vec{x})]\ [true]$$

The left side of the rule R is the term $stm_{id+1}[\mathcal{VT}(\vec{x})]$. Because we assume that the vector of variables $\mathcal{VT}(\vec{x})$ consists of the unique variables, we can conclude that the left side of the rule is linear.

Because none of rules in the set $\mathcal{R}_{list}$ has stm_{id+1} function symbol in the left-hand side of the rule, we can't find any critical pairs between the rules from the sets $\mathcal{R}_{list}$ and $\mathcal{R}_{\text{skip}}$. That means we need to look for the critical pairs only within the rules of the set $\mathcal{R}_{skip}$.

We need to show that all critical pairs computed using the rules set $\mathcal{R}_{\text{skip}}$ are trivial:

Assume that: $\mathcal{VT}(\vec{x}) = \vec{v}$.

Consider candidate critical pairs:

Rule pairs	l_1	l_2	t	σ	Candidate Critical Pair
$R \times R$	$stm_{id+1}[\vec{v}]$	$stm_{id+1}[\vec{w}]$	$stm_{id+1}[\vec{w}]$	$\forall v_i \in \vec{v} : \sigma(v_i) = v_i,$ $\forall w_i \in \vec{w} : \sigma(w_i) = v_i$	$\langle stm_{id+2}[\vec{v}], stm_{id+2}[\vec{v}], and(true, true)\rangle$

All candidate critical pairs generated using the set $\mathcal{R}_{skip}$ are trivial critical pairs.

We can conclude that the left-side of each rule in $\mathcal{R}_{list} \cup \mathcal{R}_{skip}$ is linear, and all critical pairs generated using the set $\mathcal{R}_{list} \cup \mathcal{R}_{skip}$ are trivial.

- $S = (type\ var\ \texttt{:=}\ expr)$

  The translation of the statement results in:

$$\frac{var \notin \vec{x}}{[\![type\ var\ \texttt{:=}\ expr]\!]_{\vec{x},id} = \begin{pmatrix} \begin{pmatrix} \{stm_{id+1} : signature(\mathcal{VT}(\vec{x}), list), \\ stm_{id+2} : signature(\mathcal{VT}(\vec{x}) \cdot [\mathcal{ET}(expr) : type], list)\}, \\ \{stm_{id+1}[\mathcal{VT}(\vec{x})] \to stm_{id+2}[\mathcal{VT}(\vec{x}) \cdot [\mathcal{ET}(expr) : type]]\ [true]\} \end{pmatrix}, \\ \vec{x} \cdot [var : type], \\ id + 2 \end{pmatrix}}$$

  Now, let's consider the set of LCTRS rules generated by the translation procedure:

$$\mathcal{R}_{\text{declare}} = \{stm_{id+1}[\mathcal{VT}(\vec{x})] \to stm_{id+2}[\mathcal{VT}(\vec{x}) \cdot (\mathcal{ET}(expr) : type)]\ [true]\}$$

  We need to show that:

  – The left-hand side of each rule is linear.
  – For every pair of rules $\rho_1, \rho_2 \in \mathcal{R}_{list} \cup \mathcal{R}_{declare}$: every critical pair between ρ_1 and the variable-renamed copy of ρ_2 is trivial.

  Consider the only rule in set of the LCTRS rules $\mathcal{R}_{declare}$:

$$R : stm_{id+1}[\mathcal{VT}(\vec{x})] \to stm_{id+2}[\mathcal{VT}(\vec{x}) \cdot (\mathcal{ET}(expr) : type)]\ [true]$$

  The left side of the rule R is the term $stm_{id+1}[\mathcal{VT}(\vec{x})]$. Because we assume that the vector of variables $\mathcal{VT}(\vec{x})$ consists of the unique variables, we can conclude that the left side of the rule is linear.

Because none of rules in the set $\mathcal{R}_{list}$ has stm_{id+1} function symbol in the left-hand side of the rule, we can't find any critical pairs between the rules from the sets $\mathcal{R}_{list}$ and $\mathcal{R}_{\text{declare}}$. That means we need to look for the critical pairs only within the rules of the set $\mathcal{R}_{declare}$.

We need to show that all critical pairs computed using the rules set $\mathcal{R}_{declare}$ are trivial.

Assume that: $\mathcal{VT}(\vec{x}) = \vec{v}$.

Consider candidate critical pairs:

Rule pairs	l_1	l_2	t	σ	Candidate Critical Pair
$R \times R$	$stm_{id+1}[\vec{v}]$	$stm_{id+1}[\vec{w}]$	$stm_{id+1}[\vec{w}]$	$\forall v_i \in \vec{v} : \sigma(v_i) = v_i,$ $\forall w_i \in \vec{w} : \sigma(w_i) = v_i$	$\left\langle \begin{matrix} stm_{id+2}[\vec{v} \cdot (\mathcal{ET}(expr) : type)], \\ stm_{id+2}[\vec{v} \cdot (\mathcal{ET}(expr) : type)], \\ and(true, true) \end{matrix} \right\rangle$

All candidate critical pairs generated using the set $\mathcal{R}_{declare}$ are trivial critical pairs.

We can conclude that the left-side of each rule in $\mathcal{R}_{list} \cup \mathcal{R}_{declare}$ is linear, and all critical pairs generated using the set $\mathcal{R}_{list} \cup \mathcal{R}_{declare}$ are trivial.

- $S = (var \texttt{ := } expr)$

  The translation of the statement results in:

$$\frac{\begin{matrix} (|\vec{x}| = |\vec{w}|) \wedge \\ \exists p \in \{0, |\vec{x}| - 1\} : \\ \begin{pmatrix} ((x_p = var) \wedge (w_p = \mathcal{ET}(expr))) \quad \wedge \\ (\forall i \in \{0, |\vec{x}| - 1\} : (i \neq p) \implies (w_i = \mathcal{ET}(x_i))) \end{pmatrix} \end{matrix}}{[\![var \texttt{ := } expr]\!]_{\vec{x},id} = \left(\begin{pmatrix} \{stm_{id+1} : signature(\mathcal{VT}(\vec{x}), list), \\ stm_{id+2} : signature(\mathcal{VT}(\vec{x}), list)\}, \\ \{stm_{id+1}[\mathcal{VT}(\vec{x})] \to stm_{id+2}[\vec{w}]\ [true]\} \end{pmatrix}, \vec{x}, id + 2 \right)}$$

  Now, let's consider the set of LCTRS rules generated by the translation procedure:

$$\mathcal{R}_{\text{ass}} = \{stm_{id+1}[\mathcal{VT}(\vec{x})] \to stm_{id+2}[\vec{w}]\ [true]\}$$

  We need to show that:

  - The left-hand side of each rule is linear.
  - For every pair of rules $\rho_1, \rho_2 \in \mathcal{R}_{list} \cup \mathcal{R}_{ass}$: every critical pair between ρ_1 and the variable-renamed copy of ρ_2 is trivial.

  Consider the only rule in set of the LCTRS rules $\mathcal{R}_{ass}$:

$$R : stm_{id+1}[\mathcal{VT}(\vec{x})] \to stm_{id+2}[\vec{w}]\ [true]$$

  The left side of the rule R is the term $stm_{id+1}[\vec{x}]$. Because we assume that the vector of variables $\vec{x}$ consists of the unique variables, we can conclude that the left side of the rule is linear.

  Because none of rules in the set $\mathcal{R}_{list}$ has stm_{id+1} function symbol in the left-hand side of the rule, we can't find any critical pairs between the rules from the sets $\mathcal{R}_{list}$ and $\mathcal{R}_{\text{ass}}$. That means we need to look for the critical pairs only within the rules of the set $\mathcal{R}_{ass}$.

  Now we need to show that all critical pairs from the set $\mathcal{R}_{ass}$ are trivial.

Assume that: $\mathcal{VT}(\vec{x}) = \vec{v}$.

Consider candidate critical pairs:

Rule pairs	l_1	l_2	t	σ	Candidate Critical Pair
$R \times R$	$stm_{id+1}[\vec{v}]$	$stm_{id+1}[\vec{y}]$	$stm_{id+1}[\vec{y}]$	$\forall v_i \in \vec{v} : \sigma(v_i) = v_i$, $\forall y_i \in \vec{y} : \sigma(y_i) = v_i$	$\langle stm_{id+2}[\vec{w}], stm_{id+2}[\vec{w}], and(true, true)\rangle$

All candidate critical pairs generated using the set $\mathcal{R}_{ass}$ are trivial critical pairs.

We can conclude that the left-side of each rule in $\mathcal{R}_{list} \cup \mathcal{R}_{ass}$ is linear, and all critical pairs generated using the set $\mathcal{R}_{list} \cup \mathcal{R}_{ass}$ are trivial.

- $S = (\texttt{print}Type\texttt{(}expr\texttt{)})$

  The translation of the statement results in:

$$\frac{\begin{array}{l}(|\vec{x}| = |\vec{w}|)\wedge \\ \exists p \in \{0, |\vec{x}| - 1\} : \\ \quad \left(\begin{array}{l}(x_p = \texttt{Out}) \wedge (w_p = consT(\mathcal{ET}(expr), \mathcal{VT}(\texttt{Out})))\wedge \\ (\forall i \in \{0, |\vec{x}| - 1\} : (i \neq p) \implies (w_i = \mathcal{ET}(x_i)))\end{array}\right)\end{array}}{[\![\texttt{printType(}expr\texttt{)}]\!]_{\vec{x},id} = \left(\left(\begin{array}{l}\{stm_{id+1} : signature(\mathcal{VT}(\vec{x}), list), \\ stm_{id+2} : signature(\mathcal{VT}(\vec{x}), list)\}, \\ \{stm_{id+1}[\mathcal{VT}(\vec{x})] \to stm_{id+2}[\vec{w}]\ [true]\}\end{array}\right), \vec{x}, id + 2\right)}$$

  Now, let's consider the set of LCTRS rules generated by the translation procedure:

$$\mathcal{R}_{\text{print}} = \{stm_{id+1}[\mathcal{VT}(\vec{x})] \to stm_{id+2}[\vec{w}]\ [true]\}$$

  We need to show that:

  - The left-hand side of each rule is linear.
  - For every pair of rules $\rho_1, \rho_2 \in \mathcal{R}_{list} \cup \mathcal{R}_{print}$: every critical pair between ρ_1 and the variable-renamed copy of ρ_2 is trivial.

  Consider the only rule in set of the LCTRS rules $\mathcal{R}_{print}$:

$$R : stm_{id+1}[\mathcal{VT}(\vec{x})] \to stm_{id+2}[\vec{w}]\ [true]$$

  The left side of the rule R is the term $stm_{id+1}[\mathcal{VT}(\vec{x})]$. Because we assume that the vector of variables $\mathcal{VT}(\vec{x})$ consists of the unique variables, we can conclude that the left side of the rule is linear.

  Because none of rules in the set $\mathcal{R}_{list}$ has stm_{id+1} function symbol in the left-hand side of the rule, we can't find any critical pairs between the rules from the sets $\mathcal{R}_{list}$ and $\mathcal{R}_{\text{print}}$. That means we need to look for the critical pairs only within the rules of the set $\mathcal{R}_{print}$.

  We need to show that all critical pairs from the set $\mathcal{R}_{print}$ are trivial.

  Assume that: $\mathcal{VT}(\vec{x}) = \vec{v}$.

  Consider candidate critical pairs:

Rule pairs	l_1	l_2	t	σ	Candidate Critical Pair
$R \times R$	$stm_{id+1}[\vec{v}]$	$stm_{id+1}[\vec{y}]$	$stm_{id+1}[\vec{y}]$	$\forall v_i \in \vec{v} : \sigma(v_i) = v_i$, $\forall y_i \in \vec{y} : \sigma(y_i) = v_i$	$\langle stm_{id+2}[\vec{w}], stm_{id+2}[\vec{w}], and(true, true)\rangle$

  All candidate critical pairs generated using the set $\mathcal{R}_{print}$ are trivial critical pairs.

  We can conclude that the left-side of each rule in $\mathcal{R}_{list} \cup \mathcal{R}_{print}$ is linear, and all critical pairs generated using the set $\mathcal{R}_{list} \cup \mathcal{R}_{print}$ are trivial.

- $S = (\texttt{read}Type(var))$

  The translation of the statement results in:

$$\frac{\begin{array}{l}(|\vec{x}| = |\vec{w}|) \wedge \\ \exists p_1, p_2 \in \{0, |\vec{x}| - 1\} : \\ \quad \left(\begin{array}{l}(p_1 \neq p_2) \wedge (x_{p_1} = var) \wedge (x_{p_2} = \texttt{In}) \quad \wedge \\ (w_{p_1} = headT(\mathcal{VT}(\texttt{In}))) \wedge (w_{p_2} = tailT(\mathcal{VT}(\texttt{In}))) \quad \wedge \\ (\forall i \in \{0, |\vec{x}| - 1\} : (i \neq p_1 \wedge i \neq p_2) \implies (w_i = \mathcal{ET}(x_i)))\end{array}\right)\end{array}}{[\![\texttt{readType}(var)]\!]_{\vec{x},id} = \left(\left(\begin{array}{l}\{stm_{id+1} : signature(\mathcal{VT}(\vec{x}), list), \\ stm_{id+2} : signature(\mathcal{VT}(\vec{x}), list)\}, \\ \{stm_{id+1}[\mathcal{VT}(\vec{x})] \to stm_{id+2}[\vec{w}]\ [true]\}\end{array}\right), \vec{x}, id + 2\right)}$$

  Now, let's consider the set of LCTRS rules generated by the translation procedure:

$$\mathcal{R}_{read} = \{stm_{id+1}[\mathcal{VT}(\vec{x})] \to stm_{id+2}[\vec{w}]\ [true]\}$$

  We need to show that:

  - The left-hand side of each rule is linear.
  - For every pair of rules $\rho_1, \rho_2 \in \mathcal{R}_{list} \cup \mathcal{R}_{read}$: every critical pair between ρ_1 and the variable-renamed copy of ρ_2 is trivial.

  Consider the only rule of the LCTRS $\mathcal{R}_{read}$:

$$R : stm_{id+1}[\mathcal{VT}(\vec{x})] \to stm_{id+2}[\vec{w}]\ [true]$$

  The left side of the rule R is the term $stm_{id+1}[\mathcal{VT}(\vec{x})]$. Because we assume that the vector of variables $\mathcal{VT}(\vec{x})$ consists of the unique variables, we can conclude that the left side of the rule is linear.

  Because none of rules in the set $\mathcal{R}_{list}$ has stm_{id+1} function symbol in the left-hand side of the rule, we can't find any critical pairs between the rules from the sets $\mathcal{R}_{list}$ and $\mathcal{R}_{\text{read}}$. That means we need to look for the critical pairs only within the rules of the set $\mathcal{R}_{read}$.

  We need to show that all critical pairs from the set $\mathcal{R}_{read}$ are trivial.

  Assume that: $\mathcal{VT}(\vec{x}) = \vec{v}$.

  Consider candidate critical pairs:

Rule pairs	l_1	l_2	t	σ	Candidate Critical Pair
$R \times R$	$stm_{id+1}[\vec{v}]$	$stm_{id+1}[\vec{y}]$	$stm_{id+1}[\vec{y}]$	$\forall v_i \in \vec{v} : \sigma(v_i) = v_i$, $\forall y_i \in \vec{y} : \sigma(y_i) = v_i$	$\langle stm_{id+2}[\vec{w}], stm_{id+2}[\vec{w}], and(true, true)\rangle$

  All candidate critical pairs generated using the set $\mathcal{R}_{read}$ are trivial critical pairs.

  We can conclude that the left-side of each rule in $\mathcal{R}_{list} \cup \mathcal{R}_{read}$ is linear, and all critical pairs generated using the set $\mathcal{R}_{list} \cup \mathcal{R}_{read}$ are trivial.

- $S = (S_1\,;\ S_2)$

The translation of the statement results in:

$$\frac{[\![S_1]\!]_{\vec{x},id} = \left(\begin{pmatrix}\Sigma_{\text{comp}_1},\\ \mathcal{R}_{\text{comp}_1}\end{pmatrix}, \vec{x}\cdot\vec{y}, id'\right) \qquad [\![S_2]\!]_{\vec{x}\cdot\vec{y},id'} = \left(\begin{pmatrix}\Sigma_{\text{comp}_2},\\ \mathcal{R}_{\text{comp}_2}\end{pmatrix}, \vec{x}\cdot\vec{y}\cdot\vec{z}, id''\right)}{[\![S_1\,;\ S_2]\!]_{\vec{x},id} = \left(\begin{pmatrix}\Sigma_{\text{comp}_1}\cup\Sigma_{\text{comp}_2},\\ \mathcal{R}_{\text{comp}_1}\cup\\ \{stm_{id'}[\mathcal{VT}(\vec{x}\cdot\vec{y})] \to stm_{id'+1}[\mathcal{VT}(\vec{x}\cdot\vec{y})]\ [true]\}\cup\\ \mathcal{R}_{\text{comp}_2}\end{pmatrix}, \vec{x}\cdot\vec{y}\cdot\vec{z}, id''\right)}$$

Now let's consider the set of LCTRS rules generated by the translation procedure:

$$\mathcal{R}_{\text{comp}} = \mathcal{R}_{\text{comp}_1} \cup \mathcal{R}'_{\text{comp}} \cup \mathcal{R}_{\text{comp}_2}$$

Where:

$$\mathcal{R}'_{\text{comp}} = \{stm_{id'}[\mathcal{VT}(\vec{x}\cdot\vec{y})] \to stm_{id'+1}[\mathcal{VT}(\vec{x}\cdot\vec{y})]\ [true]\}$$

Induction hypothesis (IH1):

Consider the application of the translation function $\mathcal{T}$ to the sub-statement S_1:

$$[\![S_1]\!]_{id,\vec{x}} = ((\Sigma_{\text{comp}_1}, \mathcal{R}_{\text{comp}_1}), \vec{x}\cdot\vec{y}, id')$$

Assume that for the set of the LCTRS rules $\mathcal{R}_{\text{comp}_1}$, the following holds:

- The left-hand side of each rule is linear.
- For every pair of rules $\rho_1, \rho_2 \in \mathcal{R}_{list} \cup \mathcal{R}_{\text{comp}_1}$: every critical pair between ρ_1 and the variable-renamed copy of ρ_2 is trivial.

Induction hypothesis (IH2):

Consider the application of the translation function $\mathcal{T}$ to the sub-statement S_2:

$$[\![S_2]\!]_{id',\vec{x}\cdot\vec{y}} = ((\Sigma_{\text{comp}_2}, \mathcal{R}_{\text{comp}_2}), \vec{x}\cdot\vec{y}\cdot\vec{z}, id'')$$

Assume that for the set of the LCTRS rules $\mathcal{R}_{\text{comp}_2}$, the following holds:

- The left-hand side of each rule is linear.
- For every pair of rules $\rho_1, \rho_2 \in \mathcal{R}_{list} \cup \mathcal{R}_{\text{comp}_2}$: every critical pair between ρ_1 and the variable-renamed copy of ρ_2 is trivial.

Induction step:

Using the induction hypotheses, we need to prove that for the set of LCTRS rules $\mathcal{R}_{comp}$ it holds that:

- The left-hand side of each rule is linear.
- For every pair of rules $\rho_1, \rho_2 \in \mathcal{R}_{list} \cup \mathcal{R}_{comp}$: every critical pair between ρ_1 and the variable-renamed copy of ρ_2 is trivial.

From the **IH1** and **IH2**, it follows that the left-hand sides of the rules in the sets of LCTRS rules $\mathcal{R}_{list} \cup \mathcal{R}_{\text{comp}_1}$ and $\mathcal{R}_{list} \cup \mathcal{R}_{\text{comp}_2}$ are linear.

Consider the additional rule generated while translating the composition of two statements (rules from $\mathcal{R}'_{comp}$):

$$R : stm_{id'}[\mathcal{VT}(\vec{x}\cdot\vec{y})] \to stm_{id'+1}[\mathcal{VT}(\vec{x}\cdot\vec{y})]\ [true]$$

The left side of the rule R is the term $stm_{id'}[\mathcal{VT}(\vec{x} \cdot \vec{y})]$. The vector of variables $\vec{x}$ consists of the unique variables, the new variables from vector $\vec{y}$ (potentially declared in the sub-statement S_1) are unique, and there are no variables in common in vectors $\vec{x}$ and $\vec{y}$ (caused by restrictions of variable declaration rule). We can conclude that the left side of the rule is linear.

That means all left sides of the rules from a set of LCTRS rules $\mathcal{R}_{comp}$ are linear.

We need to show that all critical pairs generated using the rules from the set $\mathcal{R}_{comp}$ are trivial.

All the LCTRS rules generated by the translation function $\mathcal{T}$ have the following structure:

$$stm_{id}[\vec{x}] \to stm_{id'}[\vec{x'}]\ [\varphi]$$

Where id, id' are integer identifiers, $stm_{id}, stm_{id'}$ are function symbols and $\vec{x}, \vec{x'}$ are variable vectors. So, two LCTRS rules generated by the translation function $\mathcal{T}$ can have a critical pair if the left-hand side function symbols have the same identifier and variables.

Consider the translation of the statement S_1:

$$[\![S_1]\!]_{id,\vec{x}} = ((\Sigma_{\text{comp}_1}, \mathcal{R}_{\text{comp}_1}), \vec{x} \cdot \vec{y}, id')$$

For all LCTRS rules $stm_{id_1}[\vec{x}] \to stm_{id_2}[\vec{x'}]\ [\varphi] \in \mathcal{R}_{\text{comp}_1}$ the following holds: $id_1 \in \{id + 1, id' - 1\}$.

Consider the translation of the statement S_2:

$$[\![S_2]\!]_{id',\vec{x}\cdot\vec{y}} = ((\Sigma_{\text{comp}_2}, \mathcal{R}_{\text{comp}_2}), \vec{x} \cdot \vec{y} \cdot \vec{z}, id'')$$

For all LCTRS rules $stm_{id_1}[\vec{x}] \to stm_{id_2}[\vec{x'}]\ [\varphi] \in \mathcal{R}_2$ the following holds $id_1 \in \{id' + 1, id'' - 1\}$.

And for the additional rule generated for the composition $S_1\,;\ S_2$: the left side of the rule has the identifier id'. That means for the sets of LCTRS rules $\mathcal{R}_1$, $\mathcal{R}_2$ and $\mathcal{R}'_{comp}$ it holds that there are no pairs of rules from two sets of reduction rules with critical pairs.

That means, if there are critical pairs, they can exist exclusively within the rules of the smaller sets $(\mathcal{R}_1, \mathcal{R}_2, \mathcal{R}'_{comp})$.

According to the **IH1**: for every pair of rules $\rho_1, \rho_2 \in \mathcal{R}_{list} \cup \mathcal{R}_{\text{comp}_1}$: every critical pair between ρ_1 and the variable-renamed copy of ρ_2 is trivial.

According to the **IH2**: for every pair of rules $\rho_1, \rho_2 \in \mathcal{R}_{list} \cup \mathcal{R}_{\text{comp}_2}$: every critical pair between ρ_1 and the variable-renamed copy of ρ_2 is trivial.

It is possible to derive only trivial critical pairs from the sets of LCTRS rules $\mathcal{R}_{list} \cup \mathcal{R}_{\text{comp}_1}$ and $\mathcal{R}_{list} \cup \mathcal{R}_{\text{comp}_2}$.

Because none of rules in the set $\mathcal{R}_{list}$ has stm_{id+1} function symbol in the left-hand side of the rule, we can't find any critical pairs between the rules from the sets $\mathcal{R}_{list}$ and $\mathcal{R}'_{comp}$. That means we need to look for the critical pairs only within the rules of the set $\mathcal{R}'_{comp}$.

We need to show that all critical pairs derived from the set $\mathcal{R}'_{comp}$ are trivial.

Assume that $\mathcal{VT}(\vec{x} \cdot \vec{y}) = \vec{v}$.

Consider candidate critical pairs:

Rule pairs	l_1	l_2	t	σ	Candidate Critical Pair
$R \times R$	$stm_{id'}[\vec{v}]$	$stm_{id'}[\vec{v'}]$	$stm_{id'}[\vec{v'}]$	$\forall v_i \in \vec{v} : \sigma(v_i) = v_i,$ $\forall v'_i \in \vec{v'} : \sigma(v'_i) = v_i$	$\langle stm_{id'+1}[\vec{v}], stm_{id'+1}[\vec{v}], and(true, true)\rangle$

All candidate critical pairs generated using the set $\mathcal{R}'_{comp}$ are trivial critical pairs.

We can conclude that the left-side of each rule in $\mathcal{R}_{list} \cup \mathcal{R}_{comp}$ is linear, and all critical pairs generated using the set $\mathcal{R}_{list} \cup \mathcal{R}_{comp}$ are trivial.

- $S = (\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2)$

  The translation of the statement results in:

$$[\![S_1]\!]_{\vec{x},id+1} = \left(\begin{pmatrix}\Sigma_{\text{if}_1},\\ \mathcal{R}_{\text{if}_1}\end{pmatrix}, \vec{x}\cdot\vec{y}, id'\right) \qquad [\![S_2]\!]_{\vec{x},id'} = \left(\begin{pmatrix}\Sigma_{\text{if}_2},\\ \mathcal{R}_{\text{if}_2}\end{pmatrix}, \vec{x}\cdot\vec{z}, id''\right)$$

$$[\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!]_{\vec{x},id} =$$

$$\left(\begin{pmatrix}\Sigma_{\text{if}_1} \cup \Sigma_{\text{if}_2} \cup \\ \{stm_{id+1} : signature(\mathcal{VT}(\vec{x}), list),\\ stm_{id''+1} : signature(\mathcal{VT}(\vec{x}), list)\},\\ \{stm_{id+1}[\mathcal{VT}(\vec{x})] \to stm_{id+2}[\mathcal{VT}(\vec{x})]\ [\mathcal{BT}(b)]\} \cup \mathcal{R}_{\text{if}_1} \cup\\ \{stm_{id+1}[\mathcal{VT}(\vec{x})] \to stm_{id'+1}[\mathcal{VT}(\vec{x})]\ [not(\mathcal{BT}(b))]\} \cup \mathcal{R}_{\text{if}_2} \cup\\ \{stm_{id'}[\mathcal{VT}(\vec{x}\cdot\vec{y})] \to stm_{id''+1}[\mathcal{VT}(\vec{x})]\ [true],\\ stm_{id''}[\mathcal{VT}(\vec{x}\cdot\vec{z})] \to stm_{id''+1}[\mathcal{VT}(\vec{x})]\ [true]\}\end{pmatrix}, \vec{x}, id''+1\right)$$

Now let's consider the set of LCTRS rules generated by the translation procedure:

$$\mathcal{R}_{\text{if}} = \mathcal{R}_{\text{if}_1} \cup \mathcal{R}'_{\text{if}} \cup \mathcal{R}_{\text{if}_2}$$

Where:

$$\mathcal{R}'_{\text{if}} = \left\{\begin{array}{l} stm_{id+1}[\mathcal{VT}(\vec{x})] \to stm_{id+2}[\mathcal{VT}(\vec{x})]\ [\mathcal{BT}(b)],\\ stm_{id+1}[\mathcal{VT}(\vec{x})] \to stm_{id'+1}[\mathcal{VT}(\vec{x})]\ [not(\mathcal{BT}(b))],\\ stm_{id'}[\mathcal{VT}(\vec{x}\cdot\vec{y})] \to stm_{id''+1}[\mathcal{VT}(\vec{x})]\ [true],\\ stm_{id''}[\mathcal{VT}(\vec{x}\cdot\vec{z})] \to stm_{id''+1}[\mathcal{VT}(\vec{x})]\ [true]\end{array}\right\}$$

Induction hypothesis (IH1):

Consider the application of the translation function $\mathcal{T}$ to the sub-statement S_1:

$$[\![S_1]\!]_{id,\vec{x}} = ((\Sigma_{\text{if}_1}, \mathcal{R}_{\text{if}_1}), \vec{x}\cdot\vec{y}, id')$$

Assume that for the set of the LCTRS rules $\mathcal{R}_{\text{if}_1}$, the following holds:

- The left-hand side of each rule is linear.
- For every pair of rules $\rho_1, \rho_2 \in \mathcal{R}_{list} \cup \mathcal{R}_{\text{if}_1}$: every critical pair between ρ_1 and the variable-renamed copy of ρ_2 is trivial.

Induction hypothesis (IH2):

Consider the application of the translation function $\mathcal{T}$ to the sub-statement S_2:

$$[\![S_2]\!]_{id,\vec{x}} = ((\Sigma_{\text{if}_2}, \mathcal{R}_{\text{if}_2}), \vec{x}\cdot\vec{z}, id'')$$

Assume that for the set of the LCTRS rules $\mathcal{R}_{\text{if}_2}$, the following holds:

- The left-hand side of each rule is linear.

– For every pair of rules $\rho_1, \rho_2 \in \mathcal{R}_{list} \cup \mathcal{R}_{\text{if}_2}$: every critical pair between ρ_1 and the variable-renamed copy of ρ_2 is trivial.

Induction step:

Using the induction hypotheses we need to prove that for the set of LCTRS rules $\mathcal{R}_{if}$ it holds that:

– The left-hand side of each rule is linear.
– For every pair of rules $\rho_1, \rho_2 \in \mathcal{R}_{list} \cup \mathcal{R}_{if}$: every critical pair between ρ_1 and the variable-renamed copy of ρ_2 is trivial.

From the **IH1** and **IH2** it follows that the left-hand sides of the rules in the sets of LCTRS rules $\mathcal{R}_{list} \cup \mathcal{R}_{\text{if}_1}$ and $\mathcal{R}_{list} \cup \mathcal{R}_{\text{if}_2}$ are linear.

Consider the additional rules generated while the translation of the "if" statement (rules from the $\mathcal{R}'_{if}$):

– $R_1 : stm_{id+1}[\mathcal{VT}(\vec{x})] \to stm_{id+2}[\mathcal{VT}(\vec{x})]\ [\mathcal{BT}(b)]$
The left side of the rule R_1 is term $stm_{id+1}[\mathcal{VT}(\vec{x})]$. Because we assumed that the vector of variables $\vec{x}$ consists of the unique variables, we can conclude that the left side of the rule is linear.
– $R_2 : stm_{id+1}[\mathcal{VT}(\vec{x})] \to stm_{id'+1}[\mathcal{VT}(\vec{x})]\ [not(\mathcal{BT}(b))]$
The left side of the rule R_2 is term $stm_{id+1}[\mathcal{VT}(\vec{x})]$. Because we assumed that the vector of variables $\vec{x}$ consists of the unique variables, we can conclude that the left side of the rule is linear.
– $R_3 : stm_{id'}[\mathcal{VT}(\vec{x} \cdot \vec{y})] \to stm_{id''+1}[\mathcal{VT}(\vec{x})]\ [true]$
The left side of the rule R_3 is term $stm_{id'}[\mathcal{VT}(\vec{x} \cdot \vec{y})]$. Because we assumed that the vector of variables $\vec{x}$ consists of unique variables, the new variables from vector $\vec{y}$ (potentially declared in the sub-statement S_1) are unique, and there are no variables in common in vectors $\vec{x}$ and $\vec{y}$ (caused by restrictions of variable declaration rule)
We can conclude that the left side of the rule R_3 is linear.
– $R_4 : stm_{id''}[\mathcal{VT}(\vec{x} \cdot \vec{z})] \to stm_{id''+1}[\mathcal{VT}(\vec{x})]\ [true]$
The left side of the rule R_4 is term $stm_{id''}[\mathcal{VT}(\vec{x} \cdot \vec{z})]$. The vector of variables $\vec{x}$ consists of unique variables, the new variables from vector $\vec{z}$ (potentially declared in the sub-statement S_2) are unique, and there are no variables in common in vectors $\vec{x}$ and $\vec{z}$ (caused by restrictions of variable declaration rule).
We can conclude that the left side of the rule R_4 is linear.

That means all left-hand sides of the LCTRS rules from $\mathcal{R}_{if}$ are linear.

We need to show that all critical pairs derived from the set $\mathcal{R}_{if}$ are trivial.

Consider the translation of the statement S_1:

$$[\![S_1]\!]_{id+1,\vec{x}} = ((\Sigma_{\text{if}_1}, \mathcal{R}_{\text{if}_1}), \vec{x} \cdot \vec{y}, id')$$

For all LCTRS rules $stm_{id_1}[\vec{x}] \to stm_{id_2}[\vec{x'}]\ [\varphi] \in \mathcal{R}_1$ the following holds: $id_1 \in \{id + 2, id' - 1\}$.

Consider the translation of the statement S_2:

$$[\![S_2]\!]_{id',\vec{x}} = ((\Sigma_{\text{if}_2}, \mathcal{R}_{\text{if}_2}), \vec{x} \cdot \vec{z}, id'')$$

For all LCTRS rules $stm_{id_1}[\vec{x}] \to stm_{id_2}[\vec{x'}]\ [\varphi] \in \mathcal{R}_2$ the following holds: $id_1 \in \{id' + 1, id'' - 1\}$.

And for the additional LCTRS rules from $\mathcal{R}'_{if}$, the left sides of rules have the following identifiers: $\{id+1, id', id''\}$.

The sets of identifiers related to the rules of different LCTRS do not intersect. That means for the sets $\mathcal{R}_1, \mathcal{R}_2$, and $\mathcal{R}'_{if}$ it holds that there are no pairs of rules from two different sets such that they have critical pairs.

So, if there are critical pairs, they can exist exclusively within the rules of the smaller sets $(\mathcal{R}_{\text{if}_1}, \mathcal{R}_{\text{if}_2}, \mathcal{R}'_{if})$.

According to the **IH1**: for every pair of rules $\rho_1, \rho_2 \in \mathcal{R}_{list} \cup \mathcal{R}_{\text{if}_1}$: every critical pair between ρ_1 and the variable-renamed copy of ρ_2 is trivial.

According to the **IH2**: for every pair of rules $\rho_1, \rho_2 \in \mathcal{R}_{list} \cup \mathcal{R}_{\text{if}_2}$: every critical pair between ρ_1 and the variable-renamed copy of ρ_2 is trivial.

Because none of rules in the set $\mathcal{R}_{list}$ has stm_{id+1}, $stm_{id'}$, or $stm_{id''}$ function symbol in the left-hand side of the rule, we can't find any critical pairs between the rules from the sets $\mathcal{R}_{list}$ and $\mathcal{R}'_{if}$. That means we need to look for the critical pairs only within the rules of the set $\mathcal{R}'_{if}$.

We need to show that all critical pairs derived from the set $\mathcal{R}'_{if}$ are trivial.

Assume that $\mathcal{VT}(\vec{x}) = \vec{v}$, $\mathcal{VT}(\vec{x} \cdot \vec{y}) = \vec{w}$, and $\mathcal{VT}(\vec{x} \cdot \vec{z}) = \vec{u}$.

Consider candidate critical pairs:

Rule pairs	l_1	l_2	t	σ	Candidate Critical Pair
$R_1 \times R_1$	$stm_{id+1}[\vec{v}]$	$stm_{id+1}[\vec{v'}]$	$stm_{id+1}[\vec{v'}]$	$\forall v_i \in \vec{v} : \sigma(v_i) = v_i,$ $\forall v'_i \in \vec{v'} : \sigma(v'_i) = v_i$	$\langle stm_{id+2}[\vec{v}], stm_{id+2}[\vec{v}], and(\mathcal{BT}(b), \mathcal{BT}(b))\rangle$
$R_2 \times R_2$	$stm_{id+1}[\vec{v}]$	$stm_{id+1}[\vec{v'}]$	$stm_{id+1}[\vec{v'}]$	$\forall v_i \in \vec{v} : \sigma(v_i) = v_i,$ $\forall v'_i \in \vec{v'} : \sigma(v'_i) = v_i$	$\langle stm_{id'+1}[\vec{v}], stm_{id'+1}[\vec{v}], and(not(\mathcal{BT}(b)), not(\mathcal{BT}(b)))\rangle$
$R_1 \times R_2$	$stm_{id+1}[\vec{v}]$	$stm_{id+1}[\vec{v'}]$	$stm_{id+1}[\vec{v'}]$	$\forall v_i \in \vec{v} : \sigma(v_i) = v_i,$ $\forall v'_i \in \vec{v'} : \sigma(v'_i) = v_i$	$\langle stm_{id+2}[\vec{v}], stm_{id'+1}[\vec{v}], and(\mathcal{BT}(b), not(\mathcal{BT}(b)))\rangle$
$R_2 \times R_1$	$stm_{id+1}[\vec{v}]$	$stm_{id+1}[\vec{v'}]$	$stm_{id+1}[\vec{v'}]$	$\forall v_i \in \vec{v} : \sigma(v_i) = v_i,$ $\forall v'_i \in \vec{v'} : \sigma(v'_i) = v_i$	$\langle stm_{id'+1}[\vec{v}], stm_{id+2}[\vec{v}], and(not(\mathcal{BT}(b)), \mathcal{BT}(b))\rangle$
$R_3 \times R_3$	$stm_{id'}[\vec{w}]$	$stm_{id'}[\vec{w'}]$	$stm_{id'}[\vec{w'}]$	$\forall w_i \in \vec{w} : \sigma(w_i) = w_i,$ $\forall w'_i \in \vec{w'} : \sigma(w') = w_1$	$\langle stm_{id'}[\vec{v}], stm_{id'}[\vec{v}], and(true, true)\rangle$
$R_4 \times R_4$	$stm_{id''}[\vec{u}]$	$stm_{id''}[\vec{u'}]$	$stm_{id''}[\vec{u'}]$	$\forall u_i \in \vec{u} : \sigma(u_i) = u_i,$ $\forall u'_i \in \vec{u'} : \sigma(u'_i) = u_i$	$\langle stm_{id''+1}[\vec{v}], stm_{id''+1}[\vec{v}], and(true, true)\rangle$

We can clearly see that the following candidate critical pairs are trivial critical pairs:

$$\begin{pmatrix} \langle stm_{id+2}[\vec{v}], stm_{id+2}[\vec{v}], and(\mathcal{BT}(b), \mathcal{BT}(b))\rangle, \\ \langle stm_{id'+1}[\vec{v}], stm_{id'+1}[\vec{v}], and(not(\mathcal{BT}(b)), not(\mathcal{BT}(b)))\rangle, \\ \langle stm_{id'}[\vec{v}], stm_{id'}[\vec{v}], and(true, true)\rangle, \\ \langle stm_{id''+1}[\vec{v}], stm_{id''+1}[\vec{v}], and(true, true)\rangle \end{pmatrix}$$

However, it is not that obvious with the following candidate critical pairs:

$$\begin{pmatrix} \langle stm_{id+2}[\vec{v}], stm_{id'+1}[\vec{v}], and(\mathcal{BT}(b), not(\mathcal{BT}(b)))\rangle, \\ \langle stm_{id'+1}[\vec{v}], stm_{id+2}[\vec{v}], and(not(\mathcal{BT}(b)), \mathcal{BT}(b))\rangle \end{pmatrix}$$

We need to verify whether these candidate critical pairs satisfy the definition of the critical pair (Definition 6.1.4) and whether they are trivial (Definition 6.1.5).

– Consider the candidate critical pair:

$$\langle stm_{id+2}[\vec{v}], stm_{id'+1}[\vec{v}], and(\mathcal{BT}(b), not(\mathcal{BT}(b)))\rangle$$

For the terms $s = stm_{id+2}[\vec{v}]$ and $t = stm_{id'+1}[\vec{v}]$ it would never hold that for every substitution γ we will have $s\gamma = t\gamma$.

Consider the candidate pair condition with some substitution γ:

$$and(\mathcal{BT}(b), not(\mathcal{BT}(b)))\gamma$$

This condition is a contradiction:

$$\mathcal{J}(and(\mathcal{BT}(b), not(\mathcal{BT}(b)))\gamma) = \mathcal{J}(\mathcal{BT}(b)\gamma) \wedge \neg\mathcal{J}(\mathcal{BT}(b)\gamma) = \bot$$

That means there does not exist a substitution which respects:

$$and(\mathcal{BT}(b), not(\mathcal{BT}(b)))$$

From this, it follows that the candidate critical pair

$$\langle stm_{id+2}[\vec{v}], stm_{id'+1}[\vec{v}], and(\mathcal{BT}(b), not(\mathcal{BT}(b)))\rangle$$

doesn't satisfy the definition of the critical pair (Definition 6.1.4).

– Consider the candidate critical pair:

$$\langle stm_{id'+1}[\vec{v}], stm_{id+2}[\vec{v}], and(not(\mathcal{BT}(b)), \mathcal{BT}(b))\rangle$$

For the terms $s = stm_{id+2}[\vec{v}]$ and $t = stm_{id+2}[\vec{v}]$ it would never hold that for every substitution γ we will have $s\gamma = t\gamma$.
Consider the candidate critical pair condition with some substitution γ:

$$and(not(\mathcal{BT}(b)), \mathcal{BT}(b))\gamma$$

This condition is a contradiction:

$$\mathcal{J}(and(not(\mathcal{BT}(b)), \mathcal{BT}(b))\gamma) = \neg\mathcal{J}(\mathcal{BT}(b)\gamma) \wedge \mathcal{J}(\mathcal{BT}(b)\gamma) = \bot$$

That means there does not exist a substitution which respects:

$$and(not(\mathcal{BT}(b)), \mathcal{BT}(b))$$

From this, it follows that the candidate critical pair:

$$\langle stm_{id'+1}[\vec{v}], stm_{id+2}[\vec{v}], and(not(\mathcal{BT}(b)), \mathcal{BT}(b))\rangle$$

doesn't satisfy the definition of the critical pair (Definition 6.1.4).

All critical pairs generated using the set $\mathcal{R}_{if}$ are trivial.

We can conclude that the left-side of each rule in $\mathcal{R}_{list} \cup \mathcal{R}_{if}$ is linear, and all critical pairs generated using the set $\mathcal{R}_{list} \cup \mathcal{R}_{if}$ are trivial.

- $S = (\texttt{while}\ b\ \texttt{do}\ S)$

The translation of the statement results in:

$$[\![S]\!]_{\vec{x},id+1} = \left(\begin{pmatrix}\Sigma_{\text{while}_1},\\ \mathcal{R}_{\text{while}_1}\end{pmatrix}, \vec{x}\cdot\vec{y}, id'\right)$$

$$[\![\texttt{while}\ b\ \texttt{do}\ S]\!]_{\vec{x},id} =$$

$$\left(\begin{pmatrix}\Sigma_{\text{while}_1} \cup\\ \{stm_{id+1} : signature(\mathcal{VT}(\vec{x}), list),\\ stm_{id'+1} : signature(\mathcal{VT}(\vec{x}), list)\},\\ \{stm_{id+1}[\mathcal{VT}(\vec{x})] \to stm_{id+2}[\mathcal{VT}(\vec{x})]\ [\mathcal{BT}(b)]\} \cup \mathcal{R}_{\text{while}_1} \cup\\ \{stm_{id'}[\mathcal{VT}(\vec{x}\cdot\vec{y})] \to stm_{id+1}[\mathcal{VT}(\vec{x})]\ [true]\\ stm_{id+1}[\mathcal{VT}(\vec{x})] \to stm_{id'+1}[\mathcal{VT}(\vec{x})]\ [not(\mathcal{BT}(b))]\}\end{pmatrix}, \vec{x}, id'+1\right)$$

Now let's consider the set of LCTRS rules generated by the translation procedure:

$$\mathcal{R}_{\text{while}} = \mathcal{R}_{\text{while}_1} \cup \mathcal{R}'_{\text{while}}$$

Where:

$$\mathcal{R}'_{\text{while}} = \left\{ \begin{array}{l} stm_{id+1}[\mathcal{VT}(\vec{x})] \to stm_{id+2}[\mathcal{VT}(\vec{x})]\ [\mathcal{BT}(b)], \\ stm_{id'}[\mathcal{VT}(\vec{x} \cdot \vec{y})] \to stm_{id+1}[\mathcal{VT}(\vec{x})]\ [true], \\ stm_{id+1}[\mathcal{VT}(\vec{x})] \to stm_{id'+1}[\mathcal{VT}(\vec{x})]\ [not(\mathcal{BT}(b))] \end{array} \right\}$$

Induction hypothesis (IH):

Consider the application of the translation function $\mathcal{T}$ to the sub-statement S:

$$[\![S]\!]_{id+1,\vec{x}} = ((\Sigma_{\text{while}_1}, \mathcal{R}_{\text{while}_1}), \vec{x} \cdot \vec{y}, id')$$

Assume that for the set of the LCTRS rules $\mathcal{R}_{\text{while}}$, the following holds:

- The left-hand side of each rule is linear.
- For every pair of rules $\rho_1, \rho_2 \in \mathcal{R}_{list} \cup \mathcal{R}_{\text{while}}$: every critical pair between ρ_1 and the variable-renamed copy of ρ_2 is trivial.

Induction step:

Using the induction hypotheses we need to prove that for the set of LCTRS rules $\mathcal{R}_{while}$ it holds that:

- The left-hand side of each rule is linear.
- For every pair of rules $\rho_1, \rho_2 \in \mathcal{R}_{list} \cup \mathcal{R}_{while}$: every critical pair between ρ_1 and the variable-renamed copy of ρ_2 is trivial.

From the **IH**, it follows that the left-hand sides of the rules in the sets of LCTRS rules $\mathcal{R}_{list} \cup \mathcal{R}_{\text{while}_1}$ are linear.

Consider the additional rules generated while the translation of the "while" statement (rules from the $\mathcal{R}'_{while}$):

- $R_1 : stm_{id+1}[\mathcal{VT}(\vec{x})] \to stm_{id+2}[\mathcal{VT}(\vec{x})]\ [\mathcal{BT}(b)]$
  The left side of the rule R_1 is the term $stm_{id+1}[\vec{x}]$. Because we assumed that the vector of variables x consists of the unique variables, we can conclude that the left side of the rule is linear.
- $R_2 : stm_{id'}[\mathcal{VT}(\vec{x} \cdot \vec{y})] \to stm_{id+1}[\mathcal{VT}(\vec{x})]\ [true]$
  The left side of the rule R_2 is term $stm_{id'}[\mathcal{VT}(\vec{x} \cdot \vec{y})]$. The vector of variables $\vec{x}$ consists of unique variables, the new variables from vector $\vec{y}$ (potentially declared in the sub-statement S) are unique, and there are no variables in common in vectors $\vec{x}$ and $\vec{y}$ (caused by restrictions of variable declaration rule).
- $R_3 : stm_{id+1}[\mathcal{VT}(\vec{x})] \to stm_{id'+1}[\mathcal{VT}(\vec{x})]\ [not(\mathcal{BT}(b))]$
  The left side of the rule R_3 is the term $stm_{id+1}[\vec{x}]$. Because we assumed that the vector of variables x consists of the unique variables, we can conclude that the left side of the rule is linear.

That means all left sides of the LCTRS rules form $\mathcal{R}_{while}$ are linear.

We need to show that all critical pairs derived from the set $\mathcal{R}_{while}$ are trivial.

Consider the translation of the statement S:

$$[\![S]\!]_{id+1,\vec{x}} = ((\Sigma_{\text{while}_1}, \mathcal{R}_{\text{while}_1}), \vec{x} \cdot \vec{y}, id')$$

For all LCTRS rules $stm_{id_1}[\vec{x}] \to stm_{id_2}[\vec{x'}]\ [\varphi]$ the following holds: $id_1 \in \{id+2, id'-1\}$.

And for the additional rules from $\mathcal{R}'_{while}$, the left sides of rules have the following identifiers: $\{id+1, id'\}$. The sets of identifiers related to the rules of different LCTRS do not intersect.

That means for the sets $\mathcal{R}, \mathcal{R}'_{while}$, it holds that there are no pairs of rules from two different sets of rules that have critical pairs. So, if there are critical pairs, they can exist exclusively within the rules of the smaller sets $(\mathcal{R}_{\text{while}_1}, \mathcal{R}'_{while})$.

According to the **IH**, for every pair of rules $\rho_1, \rho_2 \in \mathcal{R}_{list} \cup \mathcal{R}_{\text{while}_1}$: every critical pair between ρ_1 and the variable-renamed copy of ρ_2 is trivial.

Because none of rules in the set $\mathcal{R}_{list}$ has stm_{id+1}, $stm_{id'}$, or $stm_{id''}$ function symbol in the left-hand side of the rule, we can't find any critical pairs between the rules from the sets $\mathcal{R}_{list}$ and $\mathcal{R}'_{while}$. That means we need to look for the critical pairs only within the rules of the set $\mathcal{R}'_{while}$.

We need to show that all critical pairs from the set $CR(\mathcal{R}'_{while})$ are trivial.

Assume that $\mathcal{VT}(\vec{x}) = \vec{v}$, and $\mathcal{VT}(\vec{x} \cdot \vec{y}) = \vec{w}$.

Consider candidate critical pairs:

Rule pairs	l_1	l_2	t	σ	Candidate Critical Pair
$R_1 \times R_1$	$stm_{id+1}[\vec{v}]$	$stm_{id+1}[\vec{v'}]$	$stm_{id+1}[\vec{v'}]$	$\forall v_i \in \vec{v} : \sigma(v_i) = v_i,$ $\forall v'_i \in \vec{v'} : \sigma(v'_i) = v_i$	$\langle stm_{id+2}[\vec{v}], stm_{id+2}[\vec{v}], and(\mathcal{BT}(b), \mathcal{BT}(b))\rangle$
$R_2 \times R_2$	$stm_{id'}[\vec{w}]$	$stm_{id'}[\vec{w'}]$	$stm_{id'}[\vec{w'}]$	$\forall w_i \in \vec{w} : \sigma(w_i) = w_i,$ $\forall w'_i \in \vec{w'} : \sigma(w'_i) = w_i$	$\langle stm_{id+1}[\vec{v}], stm_{id+1}[\vec{v}], and(true, true)\rangle$
$R_3 \times R_3$	$stm_{id+1}[\vec{v}]$	$stm_{id+1}[\vec{v'}]$	$stm_{id+1}[\vec{v'}]$	$\forall v_i \in \vec{v} : \sigma(v_i) = v_i,$ $\forall v'_i \in \vec{v'} : \sigma(v'_i) = v_i$	$\langle stm_{id'+1}[\vec{v}], stm_{id'+1}[\vec{v}], and(not(\mathcal{BT}(b)), not(\mathcal{BT}(b)))\rangle$
$R_1 \times R_3$	$stm_{id+1}[\vec{v}]$	$stm_{id+1}[\vec{v'}]$	$stm_{id+1}[\vec{v'}]$	$\forall v_i \in \vec{v} : \sigma(v_i) = v_i,$ $\forall v'_i \in \vec{v'} : \sigma(v'_i) = v_i$	$\langle stm_{id+2}[\vec{v}], stm_{id'+1}[\vec{v}], and(\mathcal{BT}(b), not(\mathcal{BT}(b)))\rangle$
$R_3 \times R_1$	$stm_{id+1}[\vec{v}]$	$stm_{id+1}[\vec{v'}]$	$stm_{id+1}[\vec{v'}]$	$\forall v_i \in \vec{v} : \sigma(v_i) = v_i,$ $\forall v'_i \in \vec{v'} : \sigma(v'_i) = v_i$	$\langle stm_{id'+1}[\vec{v}], stm_{id+2}[\vec{v}], and(not(\mathcal{BT}(b)), \mathcal{BT}(b))\rangle$

We can clearly see that the following candidate critical pairs are trivial critical pairs:

$$\begin{pmatrix} \langle stm_{id+2}[\vec{v}], stm_{id+2}[\vec{v}], and(\mathcal{BT}(b), \mathcal{BT}(b))\rangle, \\ \langle stm_{id+1}[\vec{v}], stm_{id+1}[\vec{v}], and(true, true)\rangle, \\ \langle stm_{id'+1}[\vec{v}], stm_{id'+1}[\vec{v}], and(not(\mathcal{BT}(b)), not(\mathcal{BT}(b)))\rangle \end{pmatrix}$$

However, it is not that obvious with the following candidate critical pairs:

$$\begin{pmatrix} \langle stm_{id+2}[\vec{v}], stm_{id'+1}[\vec{v}], and(\mathcal{BT}(b), not(\mathcal{BT}(b)))\rangle, \\ \langle stm_{id'+1}[\vec{v}], stm_{id+2}[\vec{v}], and(not(\mathcal{BT}(b)), \mathcal{BT}(b))\rangle \end{pmatrix}$$

We need to verify whether these candidate critical pairs satisfy the definition of the critical pair (Definition 6.1.4) and whether they are trivial (Definition 6.1.5).

– Consider the candidate critical pair:

$$\langle stm_{id+2}[\vec{v}], stm_{id'+1}[\vec{v}], and(\mathcal{BT}(b), not(\mathcal{BT}(b)))\rangle$$

For the terms $s = stm_{id+2}[\vec{v}]$ and $t = stm_{id'+1}[\vec{v}]$ it would never hold that for every substitution γ we will have $s\gamma = t\gamma$.

Consider the candidate critical pair condition with some substitution γ:

$$and(\mathcal{BT}(b), not(\mathcal{BT}(b)))\gamma$$

This condition is a contradiction:

$$\mathcal{J}(and(\mathcal{BT}(b), not(\mathcal{BT}(b)))\gamma) = \mathcal{J}(\mathcal{BT}(b)\gamma) \wedge \neg\mathcal{J}(not(\mathcal{BT}(b)\gamma))$$

That means there does not exist a substitution which respects:

$$and(\mathcal{BT}(b), not(\mathcal{BT}(b)))$$

From this it follows that the candidate critical pair:

$$\langle stm_{id+2}[\vec{v}], stm_{id'+1}[\vec{v}], and(\mathcal{BT}(b), not(\mathcal{BT}(b)))\rangle$$

doesn't satisfy the definition of the critical pair (Definition 6.1.4).

– Consider the critical pair $\langle stm_{id'+1}[\vec{v}], stm_{id+2}[\vec{v}], and(not(\mathcal{BT}(b)), \mathcal{BT}(b))\rangle$.
For the terms $s = stm_{id'+1}[\vec{v}]$ and $t = stm_{id+2}[\vec{v}]$ it would never hold that for every substitution γ we will have $s\gamma = t\gamma$.
Consider the candidate critical pair condition with some substitution γ:

$$and(not(\mathcal{BT}(b)), \mathcal{BT}(b))\gamma$$

This condition is a contradiction:

$$\mathcal{J}(and(not(\mathcal{BT}(b)), \mathcal{BT}(b))\gamma) = \neg\mathcal{J}(\mathcal{BT}(b)\gamma) \wedge \mathcal{J}(\mathcal{BT}(b)\gamma)$$

That means there does not exist a substitution which respects

$$and(not(\mathcal{BT}(b)), \mathcal{BT}(b))$$

From this it follows that the candidate critical pair:

$$\langle stm_{id'+1}[\vec{v}], stm_{id+2}[\vec{v}], and(not(\mathcal{BT}(b)), \mathcal{BT}(b))\rangle$$

doesn't satisfy the definition of the critical pair (Definition 6.1.4).

All critical pairs generated using the set $\mathcal{R}_{while}$ are trivial.

We can conclude that the left-side of each rule in $\mathcal{R}_{list} \cup \mathcal{R}_{while}$ is linear, and all critical pairs generated using the set $\mathcal{R}_{list} \cup \mathcal{R}_{while}$ are trivial.

We proved that for the arbitrary While++ program S it holds that if we will apply the translation function $\mathcal{T}$ to this program S, starting identifier id and starting vector of unique variables $\vec{x}$:

$$[\![S]\!]_{id,\vec{x}} = ((\Sigma_S, \mathcal{R}_S), \vec{x'}, id')$$

Then for the resulting set of LCTRS rules $\mathcal{R}_S$, it holds that:

- The left-hand side of each rule $\rho \in \mathcal{R}_S$ is linear.
- For every pair of rules $\rho_1, \rho_2 \in \mathcal{R}_{list} \cup \mathcal{R}_S$: every critical pair between ρ_1 and the variable-renamed copy of ρ_2 is trivial.

Consider the LCTRS:

$$(\Sigma_{theory} \cup \Sigma_{list} \cup \Sigma_S, \mathcal{R}_{list} \cup \mathcal{R}_S, \mathcal{I}, \mathcal{J})$$

According to the Definition 6.1.6, it is weakly orthogonal.

We proved that the LCTRS constructed using the translation function $\mathcal{T}$ is weakly orthogonal. □

Bibliography

[1] IEEE Spectrum. *The Top Programming Languages 2023: Python and SQL are on top, but old languages shouldn't be forgotten*. https://spectrum.ieee.org/the-top-programming-languages-2023. 2023.

[2] Hanne Riis Nielson and Flemming Nielson. *Semantics With Applications: A Formal Introduction*. John Wiley & Sons, 1991.

[3] Cynthia Kop. *Cora*. https://github.com/hezzel/cora. 2024.

[4] Yoshiaki Kanazawa and Naoki Nishida. "On Transforming Functions Accessing Global Variables into Logically Constrained Term Rewriting Systems". In: *Electronic Proceedings in Theoretical Computer Science* 289 (Feb. 2019), pp. 34–52. ISSN: 2075-2180. DOI: 10.4204/eptcs.289.3. URL: http://dx.doi.org/10.4204/EPTCS.289.3.

[5] Naoki Nishida, Misaki Kojima, and Takumi Kato. *On Transforming Imperative Programs into Logically Constrained Term Rewrite Systems via Injective Functions from Configurations to Terms*. EasyChair Preprint no. 8673. EasyChair, 2022.

[6] Stephan Falke and Deepak Kapur. "A Term Rewriting Approach to the Automated Termination Analysis of Imperative Programs". In: *Automated Deduction – CADE-22*. Ed. by Renate A. Schmidt. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 277–293. ISBN: 978-3-642-02959-2.

[7] Stephan Falke, Deepak Kapur, and Carsten Sinz. "Termination Analysis of C Programs Using Compiler Intermediate Languages". In: *22nd International Conference on Rewriting Techniques and Applications (RTA'11)*. Ed. by Manfred Schmidt-Schauss. Vol. 10. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2011, pp. 41–50. ISBN: 978-3-939897-30-9. DOI: 10.4230/LIPIcs.RTA.2011.41. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.RTA.2011.41.

[8] Jürgen Giesl et al. "Proving Termination of Programs Automatically with AProVE". In: *Automated Reasoning*. Ed. by Stéphane Demri, Deepak Kapur, and Christoph Weidenbach. Cham: Springer International Publishing, 2014, pp. 184–191. ISBN: 978-3-319-08587-6.

[9] Carsten Fuhs, Cynthia Kop, and Naoki Nishida. "Verifying Procedural Programs via Constrained Rewriting Induction". In: *ACM Trans. Comput. Logic* 18.2 (June 2017). ISSN: 1529-3785. DOI: 10.1145/3060143. URL: https://doi.org/10.1145/3060143.

[10] Raymond T. Boute. "The Euclidean definition of the functions div and mod". In: *ACM Transactions on Programming Languages and Systems* 14 (1992).

[11] Enno Ohlebusch. *Advanced Topics in Term Rewriting*. Springer, 2002.

[12] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[13] Cynthia Kop and Naoki Nishida. *Term Rewriting with Logical Constraints*. 2013.

[14] Terence Parr. *The Definitive ANTLR 4 Reference*. 2nd ed. Raleigh, NC: Pragmatic Bookshelf, 2013. ISBN: 978-1-93435-699-9. URL: `https://www.safaribooksonline.com/library/view/the-definitive-antlr/9781941222621/`.