Proving Termination of Imperative Programs through Term Rewriting Systems
by
G.D.M. Beerendonk

Supervisor
Prof. Dr. H. Zantema

January 7, 2008, Eindhoven
Abstract

Proving whether an imperative program is terminating is generally considered to be undecidable. In several fields of the computer science before analyzing termination the programs are transformed into a term rewriting system. On these term rewriting systems termination is analyzed and implicates the (non-) termination of the original input.

For an imperative program as input the same kind of tactic can be used. A transformation from an imperative program to a term rewriting system must preserve the termination property, so if the term rewriting system is analyzed to be terminating we should be able to conclude that the original imperative program is terminating.

A direct transformation is unnecessarily difficult, so the use of an intermediate structure within the transformation makes it easier. Such an intermediate structure is the flowchart, which can be used to visualize all kind of processes, including computer programs.

The transformation from the imperative program to the flowchart is a standard transformation and is not described in mathematical detail. The succeeding transformation from the flowchart to the output term rewriting system is not straightforward, but is proved to be a correct and termination preserving transformation. Both transformations, from imperative program to flowchart and from flowchart to term rewriting system were implemented as tools in Java.

Several term rewriting systems, transformed from an imperative input program, were submitted into the annual termination competition database.
Contents

1 Introduction ............................................................................. 4
   1.1 Thesis Outline ................................................................. 5

2 Imperative Language ............................................................ 6
   2.1 Imperative Input Language .................................................. 6

3 Term Rewriting System ........................................................ 10
   3.1 Termination ..................................................................... 12

4 Flowchart ............................................................................. 14
   4.1 Representation ................................................................. 14
   4.2 Termination ..................................................................... 17

5 Transformation ...................................................................... 20
   5.1 Imperative Program to Flowchart ......................................... 20
      5.1.1 Examples .................................................................. 21
      5.1.2 Description ............................................................... 23
   5.2 Flowchart to Term-rewriting System ..................................... 24
      5.2.1 Informal Transformation ............................................. 24
      5.2.2 Formal Transformation ............................................. 26

6 Examples .............................................................................. 36
   6.1 Successful Examples ........................................................ 36
      6.1.1 Beerendonk - 1 ......................................................... 37
      6.1.2 Beerendonk - 4 ......................................................... 38
      6.1.3 Beerendonk - 9 ......................................................... 39
      6.1.4 Beerendonk - 10 ....................................................... 40
      6.1.5 Beerendonk - 15 ....................................................... 41
      6.1.6 Beerendonk - 21 ....................................................... 43
   6.2 Unsuccessful Examples ....................................................... 45
      6.2.1 GCD - Euclid’s Algorithm .......................................... 45
      6.2.2 Simple .................................................................... 46

7 Conclusions .......................................................................... 48
   7.1 Future Work .................................................................... 49

A Proofs .................................................................................. 52
   A.1 Expressions .................................................................... 52
   A.2 Conditions .................................................................... 56
Chapter 1

Introduction

A termination proof is a proof whether a given algorithm is guaranteed to terminate. The termination property plays a critical role in the total correctness proof of an algorithm. In an attempt to find this termination proof a termination analysis is needed. Ideally the analysis of termination on an imperative algorithm is done automatically, however the automatic analysis of termination on these algorithms is generally considered as undecidable. This means that there is assumed to be no way to fully proof termination for all possible algorithms without the interference of user input.

Although automatic verification of termination is undecidable, several (semi-) automatic termination tools in different computer science areas have been implemented. Some of these tools require user input (for instance loop invariants) or only offer a conclusive answer for a subset of the program input. Some examples of areas for where these kind of tools exist are logic programs (e.g. TALP [OCM00]), functional programs (e.g. AProVE [AProVE]), and term rewriting systems (e.g. AProVE [AProVE], Jambox [Jambox]).

Remarkable is that in several of the tools for logic programs and functional programs, the programs are transformed into term rewriting systems. On these term rewriting systems termination is analyzed, and the termination result for the term rewriting system implies the same termination result on the original program.

Since 2004 there exists an annual termination competition [TerComp]. This competition was started to be able to compare and challenge (with new problems) several tools in the term rewriting area. In 2007 this competition between termination tools has in 2007 four kinds of input problems, namely term rewriting systems, string rewrite systems, logic programs and functional programs. The tools which participate in the competition must determine the termination of the input without the interference of a user.

A term rewriting system is a system which consists of a set of variables and a set of rewrite rules over terms. In rewriting you can replace (sub)terms of a formula by other terms, where in a term rewriting system these replacements are restricted to the given rewrite rules. The termination of a term rewriting system is proved if it is not possible to keep rewriting infinitely with the given set of rules. Just like the termination problem in imperative programs, determination of termination on a term rewriting system is undecidable. However there are several mathematical techniques to determine whether there is an infinite sequence of rewrite steps, like semantical, syntactical and transformational methods.

Because of the existence of the mathematical techniques on term rewriting systems and the implementation of these techniques in termination tools it is a plausible idea to research a transformation from an imperative algorithm to a term rewriting system. The transformation must be termination preserving, so if an imperative input is terminating the derived term rewriting system must also be terminating.
In this thesis we want to find a transformation from an imperative program to a term rewriting system to be able to prove termination on the imperative program. The transformation will be divided in two steps, with an intermediate structure in between the imperative program and the term rewriting system. For this intermediate structure we considered options like a parse tree, a state transition diagram \cite{HoU02} or a flowchart \cite{IBM69}, from which we chose the latter. The transformation from an imperative program to a flowchart is trivial, because the flowchart is a graphical visualization of the imperative program. The transformation from the flowchart to the term rewriting system however is complex and should be proved to be a transformation which does not interfere with the termination property (we say the transformation must be termination preserving). The two steps of the transformation are also implemented to be able to automatically transform an imperative program to a term rewriting system.

1.1 Thesis Outline

In Chapter 2 we will discuss the input of our transformation, the imperative language, in the form of a grammar and the restrictions which are the consequence of choosing that grammar. After the input we will discuss the output of our transformation, a term rewriting system, in Chapter 3. We will first formally define what a term rewriting system is and define termination on a term rewriting system in Chapter 3.1. Because we transformed the imperative program to a term rewriting system with the flowchart as an intermediate structure we will look at the flowchart in Chapter 4. We will discuss which parts of the flowchart are used in the transformation and define a textual representation for implementation purposes. In Chapter 4.2 we will define the termination of a flowchart with the use of a transition system of the flowchart. With the three used structures explained in the previous chapters we can define the transformation in Chapter 5. First we will discuss the transformation from an imperative input to a flowchart in Chapter 5.1 by means of some examples. In Chapter 5.2 we will define and prove the transformation from a flowchart to a term rewriting system.

The transformation was described and implemented in Java so in Chapter 6 we will discuss some examples and their results.

We conclude in Chapter 7 with a couple of conclusions on the research and we identify future work.
Chapter 2

Imperative Language

To determine the termination of an imperative program, we first have to define the imperative language we want to work with. We could have chosen an existing imperative language like Java or C++, but that would also include difficult language specific functionality (pointers, packages, classes, library-functions) and many data types (arrays, sets, lists, strings). We chose to take a basic version of a general imperative language, which contains functionality that can be translated into any well-known imperative language and has only two basic data types: the booleans and the natural numbers.

2.1 Imperative Input Language

As our input language we require the used grammar to contain language constructions like a repetition (while), a conditional statement (if), an assignment (:=), the possibility to perform sequential steps (;) and the empty statement (skip). This basic set of language constructions is the minimum set for which we can create interesting programs to transform into a term rewriting system. This set of constructions is extendable with for example function calls. The repetition construction is an essential part of our input language, because it is the only construction within our chosen language which can cause an infinite computation, and thus non-termination.

The chosen language constructions require the existence of conditions, variables and expressions in our input language. The conditions and expressions contain functionality to compare or manipulate the variables within the program. Like the set of language constructions, the set of functionality is not fixed but extendable, and in this thesis contains the boolean connectives \( \land \) (and), \( \lor \) (or), \( \neg \) (not), the operators \( = \) (eq), \( > \) (gr), \( \neq \) (neq), \( \geq \) (geq), even, odd and the manipulators \( +1 \) (s), \( -1 \) (p), \( + \) (add), \( - \) (minus) and \( /2 \) (div2).

The several conditions and expressions mean what could be expected from functions with such a name. So and determines wether both the arguments hold, or determines if one of the arguments hold, not inverts the boolean value of the argument, neq determines the inequality of the two arguments, eq determines the equality of the two arguments, gr determines if the first argument is greater than the second argument, geq determines if the first argument is greater or equal to the second argument, even determines if the argument is an even natural number, odd determines wether the argument is an odd natural number, s determines the successor of a given argument, p determines the predecessor of a given argument (with a minimum of 0), add determines the sum of two arguments, minus determines the difference of the two arguments (with a minimum 0) and div2 determines the division with 2 of the argument.
Definition 2.1. Grammar Imperative Input Language

The chosen language constructions, conditions, expressions and the notion of variables result into the following grammar for our imperative input language:

\[
\begin{align*}
\text{⟨program⟩} & ::= \text{while (cond) do (st) od} \\
\text{⟨st⟩} & ::= (\text{st}); (\text{st}) \\
& | \text{skip} \\
& | \text{⟨var⟩} := (\text{expr}) \\
& | \text{while (cond) do (st) od} \\
& | \text{if (cond) then (st) else (st) fi} \\
\text{⟨cond⟩} & ::= \text{true} \\
& | \text{false} \\
& | \text{gr((expr),(expr))} \\
& | \text{eq((expr),(expr))} \\
& | \text{and((cond),(cond))} \\
& | \text{or((cond),(cond))} \\
& | \text{not((cond))} \\
& | \text{neq((expr),(expr))} \\
& | \text{geq((expr),(expr))} \\
& | \text{even((expr))} \\
& | \text{odd((expr))} \\
\text{⟨expr⟩} & ::= 0 \\
& | \text{⟨var⟩} \\
& | \text{s((expr))} \\
& | \text{p((expr))} \\
& | \text{add((expr),(expr))} \\
& | \text{minus((expr),(expr))} \\
& | \text{div2((expr))} \\
\text{⟨var⟩} & ::= (\text{ID}) \\
\text{⟨ID⟩} & ::= (\text{L}) \\
& | (\text{⟨ID⟩}\text{⟨L⟩}) \\
& | (\text{⟨ID⟩}\text{⟨N⟩}) \\
\text{⟨L⟩} & ::= a | b | ... | z | A | B | ... | Z \\
\text{⟨N⟩} & ::= 0 | 1 | ... | 9
\end{align*}
\]

All the connectives, operators and manipulators are in prefix notation. There is one exception to the prefix notation, the sequential composition (⟨⟩), which is in infix notation.

All natural numbers \( n \) should be presented as the successor function applied \( n \) times on 0. So if you want to use the natural number 2 it should be denoted in our language as \( s(s(0)) \).
Example 2.2. GCD Program

With the grammar we can construct an algorithm which takes two arbitrary natural numbers \((x\) and \(y)\) as input and determines their greatest common divisor (gcd). The result of this algorithm (the gcd of the original values of \(x\) and \(y)\) is the final value of \(y\). This program looks as follows:

\[
\text{while } \text{gr}(x, 0) \\
\text{do} \\
\quad \text{if } \text{gr}(x, y) \text{ then } x := \text{minus}(x, y) \\
\quad \quad \text{else } z := x; x := y; y := z \\
\text{od}
\]

There is a restriction on the root element of the program. This has to be a repetition (while), because the researched transformation does not have the capability of dealing with initializations (like in assignments (:=) and conditional statements (if)). This means that the following programs are not permitted by the grammar.

Example 2.3. Not permitted: Assignment as initialization

Consider the following imperative program:

\[
x := 0; \\
\text{while } \text{gr}(x, 0) \\
\text{do} \\
\quad x := s(x) \\
\text{od}
\]

The repetition will never be started, because the initialization \((x := 0)\) makes sure that the condition of the repetition \((\text{gr}(x, 0))\) directly will be evaluated to \text{false}. This kind of initialization is prohibited in our grammar, because our researched transformation can not deal with these initializations.

Example 2.4. Not permitted: Conditional Statement as root element

Consider the following imperative program:

\[
\text{if } \text{neq}(x, 0) \text{ then } \text{while } \text{eq}(x, 0) \text{ do } x := p(x) \text{ od } \\
\text{else } \text{skip} \\
\text{fi}
\]

The repetition within the conditional statement will never be started, because the condition of the conditional statement \((\text{neq}(x, 0))\) makes sure that the condition of the repetition \((\text{eq}(x, 0))\) directly will be evaluated to \text{false}. This initialization is prohibited in our grammar, because our researched transformation cannot deal with these initializations.
Chapter 3

Term Rewriting System

The term “term rewriting system” contains three distinctive parts, term, rewriting and system. We will look at these three aspects separately before combining them into a definition for a term rewriting system. In this chapter the definitions are taken from [Ter03] unless stated otherwise.

Terms A term is a string of symbols from an alphabet, consisting of a first-order signature and a countably infinite set of variables (Var). The first-order signature is defined as follows:

Definition 3.1. First-Order Signature

A signature Σ consists of a non-empty set of function symbols or operator symbols F, G, . . . , each equipped with a fixed arity. The arity of a function symbol F is a natural number, indicating the number of arguments it is supposed to have. So we (may) have unary, binary, ternary, etc., function symbols. Also nullary (0-ary) functions are allowed: these are also called constant symbols or constants.

The set of variables, (Var), is assumed to be disjoint from the function symbols in the signature Σ (e.g. x, y, x′, y′, x0, x1, y0, y1).

Definition 3.2. Set of Terms

The set of terms over Σ is indicated as Ter(Σ) and is defined inductively:

(i) x ∈ Ter(Σ) for every x ∈ Var.

(ii) If F is a n-ary function symbol (n ≥ 0) and t₁, . . . , tₙ ∈ Ter(Σ), then F(t₁, . . . , tₙ) ∈ Ter(Σ). By this notation it is understood (case n = 0) that the constant symbols are in Ter(Σ).

Rewriting Rewriting is the theory of stepwise transformations of objects (in this research terms). Since computations in computer science are typically stepwise transformations of objects, rewriting is a foundational theory of computing. Rewriting originates from mathematics and mathematical logic. Historically the major source for development of term rewriting is the development of the λ-calculus and its twin combinatory logic (CL) [Hin86] in the 1920s.
Example 3.3. Rewriting of an Arithmetic Term
A simple example of rewriting is to simplify arithmetical expressions for instance:

\[(3 + 5) \cdot (1 + 2) \rightarrow 8 \cdot (1 + 2) \rightarrow 8 \cdot 3 \rightarrow 24\]

The simplification process yields a result in the form of an expression that cannot be simplified any further, which we call a normal form. The relation \(\rightarrow\) is called a rewrite rule, which we will define as follows:

Definition 3.4. Rewrite Rule
A rewrite rule (or reduction rule) for a signature \(\Sigma\) is a pair \(\langle l, r \rangle\) of terms of \(\text{Ter}(\Sigma)\). It will be written as \(l \rightarrow r\). Often a reduction rule will get a name, e.g. \(\rho\), and we write \(\rho: l \rightarrow r\). Two restrictions on rewrite rules will be imposed:

(i) the left-hand side \(l\) is not a variable,

(ii) every variable occurring in the right-hand side \(r\) occurs in \(l\) as well.

System Because we know the arithmetical rewrite rules for addition and multiplication the rewriting in Example 3.3 is taking place without explicit rewrite rules. For automatic rewriting however, we have to define these rewrite rules explicitly into a system.

Example 3.5. Rewrite Rules for Addition and Multiplication

\[
\begin{align*}
0 + y & \rightarrow y \\
\text{s}(x) + y & \rightarrow \text{s}(x + y) \\
0 \cdot y & \rightarrow 0 \\
\text{s}(x) \cdot y & \rightarrow \text{s}(x) + (x \cdot y)
\end{align*}
\]

As can be seen in Example 3.3 we need to be able to substitute terms on the left-hand side with the terms on the right-hand side. Therefore a system needs to know how substitution on terms takes place.

Definition 3.6. Substitution on terms
A substitution is a map from variables to terms. A substitution \(\eta\) can be extended to arbitrary terms and atomic formulas by inductively defining:

(i) \(x \eta = \eta(x)\), for every variable \(x\) and

(ii) \(F(t_1, \ldots, t_n) \eta = F(t_1 \eta, \ldots, t_n \eta)\), for every function/relation symbol \(F\).

Example 3.7. Substitution
We will illustrate the above definition of substitution on terms with the following example. If \(\eta(x) = y\) and \(\eta(y) = g(x)\) then

\[P(f(x), y) \eta = P(f(y), g(x))\]

Term Rewriting System With the notion of terms, rewriting and system we can define a term rewriting system.

Definition 3.8. Term Rewriting System
(i) A term rewriting system is a pair \(\mathcal{R} = (\Sigma, R)\) of a signature \(\Sigma\) and a set of reduction rules \(R\) for \(\Sigma\).

(ii) The one-step reduction relation of \(\mathcal{R}\), denoted by \(\rightarrow\) (or by \(\rightarrow_R\), when we want to be more specific), is defined as the union \(\bigcup \{\rho : t \rightarrow_R s \mid \rho \in R\}\). So we have \(t \rightarrow_R s\) when \(t \rightarrow_\rho s\) for one of the reduction rules \(\rho \in R\).
Definition 3.9. Rewrite Relation

Let $\mathcal{R} = (\Sigma, R)$ be a term rewriting system, then a rewrite relation $\rightarrow_{\mathcal{R}}$ is defined to be the smallest relation $\rightarrow_{\mathcal{R}} \subseteq \text{Ter}(\Sigma) \times \text{Ter}(\Sigma)$ satisfying:

(i) $l\eta \rightarrow_{\mathcal{R}} r\eta$ for every $l \rightarrow r$ in $R$ and every substitution $\eta$.

(ii) if $t_j \rightarrow_{\mathcal{R}} u_j$ and $t_i = u_i$ for every $i \neq j$, then $f(t_1, \ldots, t_n) \rightarrow_{\mathcal{R}} f(u_1, \ldots, u_n)$.

If it is obvious which term rewriting system $\mathcal{R}$ is meant in the rewrite relation, the $\mathcal{R}$ from $\rightarrow_{\mathcal{R}}$ will be omitted.

Example 3.10. Term Rewriting System

Suppose we want to make a term rewriting system which adds two natural numbers. The natural numbers are built up with $0 : \mathbb{N}$ and $s : \mathbb{N} \rightarrow \mathbb{N}$. So we write $0$ for $0$, $s(0)$ for $1$, $s(s(0))$ for $2$, and so on. To create this term rewriting system we need to define the rewrite rules for addition as follows:

\[
\begin{align*}
\text{add}(0, y) & \rightarrow y \\
\text{add}(s(x), y) & \rightarrow s(\text{add}(x, y))
\end{align*}
\]

So if we want to compute the addition of 2 and 1, we will get the following rewrite sequence:

\[
\begin{align*}
\text{add}(s(s(0)), s(0)) & \rightarrow s(\text{add}(s(0), s(0))) \\
& \rightarrow s(s(\text{add}(0, s(0)))) \\
& \rightarrow s(s(s(0)))
\end{align*}
\]

The result of this addition is 3 if we translate $s(s(s(0)))$ to a human readable natural number.

3.1 Termination

Intuitively a term rewriting system is terminating if and only if it doesn’t contain an infinite rewrite sequence. It means that each applied rewrite rule rewrites the term towards the normal form.

Definition 3.11. Termination of a Term Rewriting System [MiZ94]

A $\text{TRS} (\Sigma, R)$ is terminating if no infinite sequence of terms $t_1, t_2, t_3, \ldots$ exists such that $t_i \rightarrow_{\mathcal{R}} t_{i+1}$ for all $i$ or formally:

$\neg (\exists i : (\forall i : t_i \rightarrow_{\mathcal{R}} t_{i+1}))$

Several methods for determining termination of a term rewriting system were developed, like semantical, syntactical and transformational methods (all described in Chapter 6 of [Ter03]). These methods can be used for many term rewriting systems, and many of the methods are implemented in automatic termination provers on term rewriting systems, like $\text{AProVE}$ [AProVE] and $\text{Jambox}$ [Jambox]. Because of the existence of these automatic termination provers on term rewriting systems, we can determine the termination of an imperative program automatically, if we can transform it into a term rewriting system which preserves the termination property.
Chapter 4

Flowchart

To make the transformation from an imperative language into a term rewriting system easier we use an intermediate structure, the flowchart. We use the flowchart because it is a straightforward way of visualizing an imperative program, it is a structure which is already known and defined and it emphasizes the paths in the imperative program, which is closely related to the rewrite sequences in a term rewriting system.

Flowcharts are researched and developed by Herman Goldstine and John von Neumann at Princeton University in late 1946 and early 1947. A flowchart is a graphical formalized structure, which makes it possible to visualize all kind of processes (e.g. computer programs, business processes). A flowchart is a graph built up from several symbols (like rectangles and diamonds) and arrows between these symbols.

4.1 Representation

We use the flowchart to represent computer algorithms, so we only need a subset of all the symbols which are available in flowcharts. Underneath is an enumeration of the symbols we use:

- **Start/End**: A start/end represents the start and end of the process respectively. It is represented by a rounded rectangle with the words start and end in it, to distinguish the start and end.

- **Decision**: A decision represents the conditional statement. It contains a question which can be answered with either yes or no. The decision has one incoming arrow, and two outgoing arrows. These outgoing arrows are labeled with `true`, following the path when the question can be answered positively, and `false`, following the path when the question can be answered negatively. The decision is represented by a diamond-shaped symbol with the question inside.

- **Arrows**: An arrow represents the flow of control from one symbol to another symbol. The symbol the arrow points to, is the symbol which the control will go to. The arrow can be labeled with `true` and `false`, according to the decision it came from.

- **Process**: A process contains a computation step of the system. It has one incoming and one outgoing arrow and is represented by a rectangle.

With these symbols we will be able to represent the algorithms built from the grammar of the imperative language we consider. Even the repetition can be represented by using a decision as the repetitions condition and after the computations in the body of the repetition we can point an arrow backwards to just before that decision.
Example 4.1. Flowchart with used symbols

The underneath flowchart shows an example of a system with all the symbols described above.

In this research, where we have to transform a flowchart into a term rewriting system the graphical representation of a flowchart, such as above, is not very useful. Therefore we will have to define a textual representation of a graphical flowchart.

Although the start symbol is a very important symbol in the flowchart, because of the possibility of initializations, it will be omitted in the textual flowchart. This is due to the fact described in the examples 2.3 and 2.4, that those initializations are not covered within our transformation.

In order to prove termination of a system, we must prove that every repetition in the system terminates. This means we are interested in the decision symbols in the graphical flowchart, the processes after the decision symbols and where the arrows originating from these processes go to.

Concluding, a textual decision will contain a name of the decision, the condition of that decision and two sequences of assignments, one of the process in the true branch and one of the process in the false branch. Furthermore it also contains two names of decisions the control will be given to after the two processes.

A complete flowchart is a, possibly empty, list of these textual decisions.
**Definition 4.2. Grammar Flowchart**

The above description of a textual flowchart is formally defined by the following grammar:

\[
\begin{align*}
\langle fc \rangle & ::= \varepsilon \\
& \quad \mid \langle \text{dec} \rangle \langle fc \rangle \\
\langle \text{dec} \rangle & ::= \langle \text{name} \rangle : \text{if} \langle \text{cond} \rangle \langle \text{process} \rangle \langle \text{name} \rangle \text{else} \langle \text{process} \rangle \langle \text{name} \rangle \\
\langle \text{process} \rangle & ::= \text{skip} \\
& \quad \mid \langle \text{var} \rangle ::= \langle \text{expr} \rangle ; \langle \text{process} \rangle \\
\langle \text{cond} \rangle & ::= \text{true} \\
& \quad \mid \text{false} \\
& \quad \mid \text{gr}(\langle \text{expr} \rangle, \langle \text{expr} \rangle) \\
& \quad \mid \text{eq}(\langle \text{expr} \rangle, \langle \text{expr} \rangle) \\
& \quad \mid \text{and}(\langle \text{cond} \rangle, \langle \text{cond} \rangle) \\
& \quad \mid \text{or}(\langle \text{cond} \rangle, \langle \text{cond} \rangle) \\
& \quad \mid \text{not}(\langle \text{cond} \rangle) \\
& \quad \mid \text{neq}(\langle \text{expr} \rangle, \langle \text{expr} \rangle) \\
& \quad \mid \text{geq}(\langle \text{expr} \rangle, \langle \text{expr} \rangle) \\
& \quad \mid \text{even}(\langle \text{expr} \rangle) \\
& \quad \mid \text{odd}(\langle \text{expr} \rangle) \\
\langle \text{expr} \rangle & ::= 0 \\
& \quad \mid \langle \text{var} \rangle \\
& \quad \mid \text{s}(\langle \text{expr} \rangle) \\
& \quad \mid \text{p}(\langle \text{expr} \rangle) \\
& \quad \mid \text{add}(\langle \text{expr} \rangle, \langle \text{expr} \rangle) \\
& \quad \mid \text{minus}(\langle \text{expr} \rangle, \langle \text{expr} \rangle) \\
& \quad \mid \text{div2}(\langle \text{expr} \rangle) \\
\langle \text{name} \rangle & ::= \langle \text{ID} \rangle \\
\langle \text{var} \rangle & ::= \langle \text{ID} \rangle \\
\langle \text{ID} \rangle & ::= \langle \text{L} \rangle \\
& \quad \mid \langle \text{ID} \rangle \langle \text{L} \rangle \\
& \quad \mid \langle \text{ID} \rangle \langle \text{N} \rangle \\
\langle \text{L} \rangle & ::= a \mid b \mid \ldots \mid z \mid A \mid B \mid \ldots \mid Z \\
\langle \text{N} \rangle & ::= 0 \mid 1 \mid \ldots \mid 9
\end{align*}
\]

where \(\varepsilon\) is the empty assignment and \(\varepsilon\) is the empty flowchart.

Furthermore there are some restrictions on the keyword \text{name}. That name should be the name of a decision and the name should be unique for that decision. A decision can only have the keyword \text{bot} in the \text{else} branch of the textual decision, because of the following two reasons. First, a decision which has \text{bot} in the first branch is easily transformed into a decision with \text{bot} in the \text{else} branch (negate the condition and swap the two processes). Secondly it is not possible for a textual flowchart, transformed from an imperative program (which is restricted to the grammar in Definition 2.1) to have \text{bot} in the first branch.
Example 4.3. Textual Representation

To illustrate the grammar of Definition 4.2, we consider the flowchart of Example 4.1. The textual representation of that flowchart is the following:

Dec1: if Cond1 Process1 Dec2
else bot

Dec2: if Cond2 Process2 Dec1
else Process3 Dec1

4.2 Termination

If we want to talk and reason about the termination of a flowchart, we will have to define termination on a flowchart.

We will use the transition system of the flowchart to be able to define termination of a flowchart. From a flowchart $F$, we can construct a (possibly infinite) transition system $G_F$, which can be defined as a tuple of the set of states $S_F$ and the set of edges $E_F$.

States

The set of states $S_F$ in this transition system $G_F$ can be defined as the combination of the decisions in the flowchart $F$ and the values of the system’s variables in the state.

Definition 4.4. Set of States

Let $F$ be a flowchart, then $S_F$ is the set of states in the flowchart, which is defined as follows:

$S_F = \mathcal{D}_F \times \mathbb{N}^k$

where $\mathcal{D}_F$ is the set of decision descriptions in $F$ and $k$ is the number of variables in $F$.

Thus a state in the system $s \in S_F$ can be denoted as follows:

$s = (d, \overrightarrow{v})$

where $d \in \mathcal{D}_F$ and $\overrightarrow{v}$ is the vector of the variables in $s$.

We need a value function, $Val$, which guarantees the correct functionality of the several expressions and conditions within the grammar of the flowchart given in Definition 4.2. So the value function takes either a condition or an expression and returns the natural number or boolean value corresponding to the meaning of the condition or expression.
Definition 4.5. Valuation Function
For \( t, u \in T(\Sigma) \) where \( \text{Val}(t, \overrightarrow{v}) \rightarrow \mathbb{N} \) and \( \text{Val}(u, \overrightarrow{v}) \rightarrow \mathbb{N} \) and \( a, b \in T(\Sigma) \) where \( \text{Val}(a, \overrightarrow{v}) \rightarrow \mathbb{B} \) and \( \text{Val}(b, \overrightarrow{v}) \rightarrow \mathbb{B} \) the valuation function \( \text{Val} : \text{Expr} \times \mathbb{N}^k \rightarrow \mathbb{N} \) where \( \text{Expr} \subseteq T(\Sigma) \) and \( \text{Expr} = \{0, s, p, \text{div}2, \text{add}, \text{minus}\} \) or \( \text{Val} : \text{Cond} \times \mathbb{N}^k \rightarrow \mathbb{B} \) where \( \text{Cond} \subseteq T(\Sigma) \) and \( \text{Cond} = \{\text{true}, \text{false}, \text{and}, \text{or}, \text{not}, \text{gr}, \text{eq}, \text{neq}, \text{even}, \text{odd}\} \) is defined as follows:

\[
\begin{aligned}
\text{Val}(0, \overrightarrow{v}) &= 0 \\
\text{Val}(\text{true}, \overrightarrow{v}) &= \text{true} \\
\text{Val}(\text{false}, \overrightarrow{v}) &= \text{false} \\
\text{Val}(\text{s}(t), \overrightarrow{v}) &= \text{Val}(t, \overrightarrow{v}) + 1 \\
\text{Val}(\text{p}(t), \overrightarrow{v}) &= 0 \text{ if } \text{Val}(t, \overrightarrow{v}) = 0 \\
&= \text{Val}(t, \overrightarrow{v}) - 1 \text{ if } \text{Val}(t, \overrightarrow{v}) > 0 \\
\text{Val}(\text{div}2(t), \overrightarrow{v}) &= \text{Val}(t, \overrightarrow{v})/2 \\
\text{Val}(\text{add}(t, u), \overrightarrow{v}) &= \text{Val}(t, \overrightarrow{v}) + \text{Val}(u, \overrightarrow{v}) \\
\text{Val}(\text{minus}(t, u), \overrightarrow{v}) &= 0 \text{ if } \text{Val}(t, \overrightarrow{v}) < \text{Val}(u, \overrightarrow{v}) \\
&= \text{Val}(t, \overrightarrow{v}) - \text{Val}(u, \overrightarrow{v}) \text{ if } \text{Val}(t, \overrightarrow{v}) \geq \text{Val}(u, \overrightarrow{v}) \\
\text{Val}(\text{and}(a, b), \overrightarrow{v}) &= \text{Val}(a, \overrightarrow{v}) \land \text{Val}(b, \overrightarrow{v}) \\
\text{Val}(\text{or}(a, b), \overrightarrow{v}) &= \text{Val}(a, \overrightarrow{v}) \lor \text{Val}(b, \overrightarrow{v}) \\
\text{Val}(\text{not}(a), \overrightarrow{v}) &= \neg \text{Val}(a, \overrightarrow{v}) \\
\text{Val}(\text{gr}(t, u), \overrightarrow{v}) &= \text{Val}(t, \overrightarrow{v}) > \text{Val}(u, \overrightarrow{v}) \\
\text{Val}(\text{geq}(t, u), \overrightarrow{v}) &= \text{Val}(t, \overrightarrow{v}) \geq \text{Val}(u, \overrightarrow{v}) \\
\text{Val}(\text{eq}(t, u), \overrightarrow{v}) &= \text{Val}(t, \overrightarrow{v}) = \text{Val}(u, \overrightarrow{v}) \\
\text{Val}(\text{neq}(t, u), \overrightarrow{v}) &= \text{Val}(t, \overrightarrow{v}) \neq \text{Val}(u, \overrightarrow{v}) \\
\text{Val}(\text{even}(t), \overrightarrow{v}) &= \text{Val}(t, \overrightarrow{v}) \mod 2 = 0 \\
\text{Val}(\text{odd}(t), \overrightarrow{v}) &= \text{Val}(t, \overrightarrow{v}) \mod 2 = 1
\end{aligned}
\]

Processes A path in the transition system \( G_F \) is a (possibly infinite) sequence of steps in the transition system. A step in the transition system can be made if a step led to another decision and/or the variables in the system are changed. The variables in the system can change due to a process. A process contains a functional multiple assignment which can change the variables in the system. So a process takes the vector of variables, the remaining assignment(s) are applied. When a sequence of assignments to a functional multiple assignment occurs first we will apply the function belonging to the first assignment, and on the resulting variables the remaining assignment(s) are applied.

Definition 4.6. Set of Processes
Let \( F \) be a flowchart, then \( P_F \) is the set of processes which appear in the flowchart, which is defined as follows:

\[
P_F = \{m : \mathbb{N}^k \rightarrow \mathbb{N}^k \mid m \in F\}
\]

where \( k \) is the number of variables in the system.

According to the definition of processes (Definition 4.6) we can consider a sequence of assignments in our grammar (Definition 4.2) as a function \( m \in P_F ; m(\overrightarrow{v}) = \overrightarrow{v}' \). Thus we need to define a way to get from the sequence of assignments to a functional multiple assignment. When we have a single assignment the variable is altered to the value of the expression which was assigned to that variable. When a sequence of assignments occurs first we will apply the function belonging to the first assignment, and on the resulting variables the remaining assignment(s) are applied.
Definition 4.7. Transformation of sequence of assignments

The transformation function $\phi(p) : \mathbb{N}^k \times \mathbb{N}^k$ with $p \in P_F$ is defined in the following way:

$\phi(x_i := e(a_1, \ldots, a_k)) = a_1, \ldots, a_{i-1}, b_i, a_{i+1}, \ldots, a_k$

where $b_i = Val(ep, a_1, \ldots, a_k)$ and $p(x_i) = s^e(0)$

$\phi(r; x_i := e(a_1, \ldots, a_k)) = \phi(r) \circ \phi(x_i := e(a_1, \ldots, a_k))$

where $r$ is a sequence of assignments.

Edges

An edge in the transition graph connects two states. The set of edges in $G_F$, denoted $E_F$, which describes the behavior of $F$, is a subset of all possible edges in $G_F$: $E_F \subseteq S_F \times S_F$. With Definition 4.4 we can rewrite this to $E_F \subseteq (\mathcal{D}_F \times \mathbb{N}^k) \times (\mathcal{D}_F \times \mathbb{N}^k)$.

Definition 4.8. Edges

The set of edges can be defined as follows:

$$E_F = \bigcup_{d: \text{if } c p_1 d_1 \text{ else } p_2 d_2 \in \mathcal{D}_F} \{(d, v), (d_1, v') \mid Val(c(v)) = \text{true} \land \phi(p_1)(v') = v'\}$$

$$\cup \{(d, v), (d_2, v') \mid Val(c(v)) = \text{false} \land \phi(p_2)(v') = v'\}$$

$$\cup \bigcup_{d: \text{if } c p_1 d_1 \text{ else bot} \in \mathcal{D}_F} \{(d, v), (d_1, v') \mid Val(c(v)) = \text{true} \land \phi(p_1)(v') = v'\}$$

Now we can define a transition system as the tuple of the set of states ($S_F$) and set of edges ($E_F$).

Definition 4.9. Transition System

$G_F = (S_F, E_F)$

With the notion of the transition system $G_F$ we can define termination of a flowchart $F$.

Definition 4.10. Termination of a flowchart

A flowchart $F$ terminates if there is no infinite sequence of edges in $G_F$.

Formally: $\neg (\exists (d, v_i) \in \mathbb{N} : (d, v_i), (d_{i+1}, v_{i+1}) \in E_F))$
Chapter 5

Transformation

In the previous chapters we discussed three concepts (imperative program, term rewriting system and flowchart). The main core of this research was to find a transformation from the imperative program to a term rewriting system in such a way that if termination was proved for the term rewriting system the original imperative program is also proved to be terminating. As an intermediate structure we used the flowchart, so first we transform the imperative program into a flowchart and next that flowchart will be transformed into a term rewriting system.

5.1 Imperative Program to Flowchart

In this section we will discuss the transformation from an imperative program into the intermediate flowchart structure. This transformation is a standard transformation, which preserves the termination property of the system. Because this transformation is standard we will not discuss this transformation in mathematical detail, but we will first illustrate this transformation by some examples and after that briefly describe the transformation.

In our imperative grammar (Definition 2.1) there are two language constructions which will result into a textual decision description, namely the repetition (while) and the conditional statement (if). The grammar obliges us to begin the program with a repetition, so there always will be a transformation possible.
5.1.1 Examples

Example 5.1. Plain repetition
In this example we consider a program with a single repetition. This is the most basic kind of program which can be described by the grammar (Definition 2.1) and transformed into a textual flowchart

\[
\text{while } \text{gr}(x, 0) \\
\text{do} \\
\quad x := \text{p}(x) \\
\text{od}
\]

This imperative program will be transformed into a textual flowchart, giving the decision corresponding to the repetition a unique name (dec1), copying the condition of the repetition to the condition of the decision (\(\text{gr}(x, 0)\)). The body of the repetition will be the first sequence of assignments \((x := \text{p}(x))\) and because there isn’t another repetition or conditional statement in the body of the loop the goto name will be the name of the decision itself (\(\text{dec1}\)). After the repetition there is no sequential composition (\(;\)), which is also not allowed by the grammar of the imperative language, so the second sequence of assignments and the second goto name are replaced with \(\text{bot}\).

This results into the following textual flowchart:

\[
\text{dec1: if } \text{gr}(x, 0) \quad x := \text{p}(x) \quad \text{dec1} \\
\quad \text{else } \quad \text{bot}
\]

Example 5.2. Repetition with nested conditional statement
In this example we consider a program which consists of a repetition and a conditional statement in its body.

\[
\text{while } \text{gr}(x, y) \\
\text{do} \\
\quad \text{if } \text{gr}(x, z) \quad \text{then } y := \text{s}(y) \\
\quad \quad \text{else } x := \text{p}(x) \\
\quad \text{fi} \\
\text{od}
\]

In this example we have no first sequence of assignments, so we fill in the empty assignment (\(\text{skip}\)). In the loop body however we now have a conditional statement which the main repetition should refer to in its first goto name. For the conditional statement there is a separate decision description, with as condition the condition of the conditional statement (\(\text{gr}(x, z)\)) and as first and second sequence of assignments the body of the \(\text{then} \ (y := \text{s}(y))\) and \(\text{else} \ (x := \text{p}(x))\) branch respectively. Both of the goto names are back to the repetition (\(\text{dec1}\)) because its the parent of of this decision. This results into the following textual flowchart:

\[
\text{dec1: if } \text{gr}(x, y) \quad \text{skip} \quad \text{dec2} \\
\quad \text{else } \quad \text{bot} \\
\text{dec2: if } \text{gr}(x, z) \quad y := \text{s}(y) \quad \text{dec1} \\
\quad \text{else } \\
\quad \quad x := \text{p}(x) \\
\quad \quad \text{dec1}
\]
Example 5.3. Repetition with nested a conditional statements and sequentially an assignment

In this example we consider a program which consists of a repetition and a conditional statement in its body with an assignment sequential to the conditional statement.

while $gr(x,y)$
do
  if $gr(x,z)$ then $x := p(x)$
    else skip
  fi;
  $y := s(y)$
end.

The structure of this program is almost identical to the previous example but now the child decision (the one of the conditional statement) has sequentially an assignment ($y := s(y)$). This means that in both cases of the conditional statement this assignment is executed sequentially with the decisions sequence of assignments, forming new sequences. We say that the sequential assignment is distributed into the conditional statement.

This results into the following textual flowchart:

```
dec1: if $gr(x,y)$ skip dec2
  else bot

dec2: if $gr(x,z)$ $x := p(x)$; $y := s(y)$ dec1
  else $y := s(y)$ dec1
```

Example 5.4. Repetition with nested repetition

In this example we consider a program which consists of a repetition and instead of a conditional statement like in the previous examples a repetition in its body.

while $gr(y,x)$
do
  while $gr(y,0)$
do
    $y := p(y)$
  od
end.

The first decision (dec1) still has the second decision (of the second repetition) as the first goto name. The second decision (dec2) has as first goto name the decision itself, because it is a repetition and there is no other repetition or conditional statement in its body. The second sequence of assignments is empty (skip), but the second goto name is not. The second goto name is the name of the decision of the parent loop (dec1).

This results into the following textual flowchart:

```
dec1: if $gr(y,x)$ skip dec2
  else bot

dec2: if $gr(y,0)$ $y := p(y)$ dec2
  else skip dec1
```

22
Example 5.5. Repetition with nested repetition and sequential assignment

In this example we consider a program which consists of a repetition and another repetition in its body and an assignment sequential to that second repetition.

```plaintext
while gr(x, 0)
do
  while gr(y, 0)
do
    y := p(y)
od;
x := p(x)
od
```

This example has the same structure as the previous example, but the sequential assignment \((x := p(x))\) will be the second sequence of assignments in the second decision (dec2).

This results into the following textual flowchart:

```plaintext
dec1: if gr(x, 0) skip dec2
else bot

dec2: if gr(y, 0) y := p(y) dec2
else x := p(x) dec1
```

5.1.2 Description

The examples above illustrate the several possible combinations in the imperative language which have a distinctive influence on the flowchart. We will describe these distinctive influences in detail.

As denoted before, there are two language constructions which cause a decision description within the flowchart, namely the `while` (in this section referred to as repetition) and the `if` (in this section referred to as conditional statement). For both these constructions we have to get a complete textual description in a flowchart, so we will need a unique name, a condition, a process, a goto name (and possibly another process and goto name). We will describe the transformation of these language constructions, with their possibilities, separately.

Repetition

A unique name must be given to each decision description. We will always give a decision description the name `dec` with a unique number. So every first decision description in our transformation will be given the name `dec1` etc.

The condition of the decision description of the repetition is the condition of the repetition in the imperative program.

The first process of the decision description of the repetition is the sequence of assignments in the repetition body. If there is no sequence of assignments in the repetition body, the process will be the empty process (skip).

The main challenge of the transformation is the determination of the goto names. The first goto name is determined by the content of the repetition body. If there is another repetition or conditional statement, it will be the the reference to the decision description of that statement. If there’s no such decision in the repetition body, the first goto name will be the reference to the decision itself, because of the repetition.
The second process and goto name are used when the decision is a child of another decision description, or when there is another decision sequential to this decision. Otherwise the second process and goto name are replaced with bot. If the decision is a child of another decision description, the second process is a (possibly empty) sequence of assignments before returning to the parent decision, and the goto name will be the name of the parent decision description. In case of a sequential decision description the second process is a (possible empty) sequence of assignments between the two decision descriptions, and the goto name will be the name of the decision description which is sequentially to this one.

Conditional Statement

A unique name must be given to each decision description. We will always give a decision description the name dec with a unique number. So every first decision description in our transformation will be given the name dec1 etc.

The condition of the decision description of the conditional statement is the condition of the conditional statement in the imperative program.

The two processes in the decision description are respectively the sequences of assignments in the then and else branch.

The two goto names in the decision description are respectively the names of decision descriptions within the then and else branch. When there is no decision description in one of the branches the goto name is the name of a sequential decision description, or when there is also no sequential decision description the name of the parent of this decision description.

We must remark that (as shown in Example 5.3) when a conditional statement has sequentially a sequence of assignments, it will be distributed into the conditional statement.

The transformation from imperative program to the intermediate flowchart structure was implemented in a java tool imp2fc.

5.2 Flowchart to Term-rewriting System

In the previous section we have shown a transformation from the imperative input to the intermediate flowchart structure. What remains is the transformation from this intermediate flowchart structure into a term rewriting system. We will first illustrate the transformation in an informal way, through an example before we formalize the transformation.

5.2.1 Informal Transformation

We can consider the program of the gcd from Example 2.2 again:

```java
while gr(x, 0)
  do
    if gr(x, y) then x := minus(x, y)
    else z := x; x := y; y := z
  od
```

24
According to the transformation described in the previous section we can transform that into the following flowchart:

![Flowchart Diagram]

Textual this graph will be represented according to Definition 4.2 as follows:

```
dec1: if gr(x,0) then skip else bot
dec2: if gr(x,y) then x := minus(x,y) else z := x, x := y, y := z
```

The transformation from a textual flowchart into a term rewriting system means that for almost each decision description there will be two rewrite rules within the term rewriting system. One for the case where the condition is true and one for the case where the condition is false. An exception is when a bot is encountered, because that rule will be omitted from the term rewriting system. Furthermore there needs to be rewrite rules for every defined symbol (expression or condition) within the flowchart. These rules are fixed per defined symbol.
The flowchart of the gcd can therefore be transformed into the following rewrite rules of a term rewriting system:

\[
\begin{align*}
gr(0, x) & \rightarrow \text{false} \\
gr(s(x), 0) & \rightarrow \text{true} \\
gr(s(x), s(y)) & \rightarrow gr(x, y) \\
\text{minus}(0, x) & \rightarrow 0 \\
\text{minus}(x, 0) & \rightarrow x \\
\text{minus}(s(x), s(y)) & \rightarrow \text{minus}(x, y) \\
dec1(\text{true}, x, y, z) & \rightarrow dec2(gr(x, y), x, y, z) \\
dec2(\text{true}, x, y, z) & \rightarrow dec1(gr(x, 0), \text{minus}(x, y), y, z) \\
dec2(\text{false}, x, y, z) & \rightarrow dec1(gr(x, 0), y, x, x)
\end{align*}
\]

We see that there are 3 rules for the condition \(gr\), 3 rules for the expression \(\text{minus}\) and 3 rules for the decisions in the flowchart.

The transformation from the intermediate flowchart structure to a term rewriting system was implemented in the java tool fc2trs.

5.2.2 Formal Transformation

We will first consider the fixed translations, for the expressions \((Expr)\) and conditions \((Cond)\) given by the grammar of the flowchart (See Definition 4.2) as defined symbols in the term rewriting system. Exceptions are \(\text{true}, \text{false}, 0\) and \(s\), which are the constructor symbols. The constructor symbols will not be in the set of defined symbols, because they are used to construct the variables of the flowchart. This means the set of defined symbols, \(\text{DefSymbols}\), will be the set of expressions \(Expr\) and conditions \(Cond\) except for the symbols \(\text{true}, \text{false}, 0\) and \(s\).

**Definition 5.6.** Set of Defined Symbols

\[\text{DefSymbols} = (Expr \cup Cond) \setminus \{s\} \setminus \{0\} \setminus \{\text{true}\} \setminus \{\text{false}\}\]

where \(Expr\) is the set of Expressions and \(Cond\) is the set of Conditions defined in the grammar of the flowchart (Definition 4.2).

The transformation of the set \(\text{DefSymbols}\) into rules for the term rewriting system is straightforward. For each defined symbol there is a separate set of rules to be added to the term rewriting system.
Definition 5.7. Transformation of Defined Symbols
In our formal transformation \( \sigma \) is a function from the defined symbols to a term rewriting system. So: \( \sigma : \text{DefSymbols} \rightarrow \text{TRS} \). Where \( x, y \in \text{Var} \)

\[
\begin{align*}
\sigma(\text{p}) &= \{ \text{p}(0) \rightarrow 0 \, , \\
&\quad \text{p}(s(x)) \rightarrow x \} \\
\sigma(\text{add}) &= \{ \text{add}(0,y) \rightarrow y \, , \\
&\quad \text{add}(s(x),y) \rightarrow s(\text{add}(x,y)) \} \\
\sigma(\text{min}) &= \{ \text{min}(0,y) \rightarrow 0 \, , \\
&\quad \text{minus}(x,0) \rightarrow x \, , \\
&\quad \text{minus}(s(x),s(y)) \rightarrow \text{minus}(x,y) \} \\
\sigma(\text{div2}) &= \{ \text{div2}(0) \rightarrow 0 \, , \\
&\quad \text{div2}(s(0)) \rightarrow 0 \, , \\
&\quad \text{div2}(s(s(x))) \rightarrow s(\text{div2}(x)) \} \\
\sigma(\text{and}) &= \{ \text{and}(\text{true},\text{true}) \rightarrow \text{true} \, , \\
&\quad \text{and}(\text{false},x) \rightarrow \text{false} \, , \\
&\quad \text{and}(x,\text{false}) \rightarrow \text{false} \} \\
\sigma(\text{or}) &= \{ \text{or}(\text{false},\text{false}) \rightarrow \text{false} \, , \\
&\quad \text{or}(\text{true},x) \rightarrow \text{true} \, , \\
&\quad \text{or}(x,\text{true}) \rightarrow \text{true} \} \\
\sigma(\text{not}) &= \{ \text{not}(\text{true}) \rightarrow \text{false} \, , \\
&\quad \text{not}(\text{false}) \rightarrow \text{true} \} \\
\sigma(\text{eq}) &= \{ \text{eq}(0,0) \rightarrow \text{true} \, , \\
&\quad \text{eq}(s(x),0) \rightarrow \text{false} \, , \\
&\quad \text{eq}(0,s(x)) \rightarrow \text{false} \, , \\
&\quad \text{eq}(s(x),s(y)) \rightarrow \text{eq}(x,y) \} \\
\sigma(\text{neq}) &= \{ \text{neq}(0,0) \rightarrow \text{false} \, , \\
&\quad \text{neq}(s(x),0) \rightarrow \text{true} \, , \\
&\quad \text{neq}(0,s(x)) \rightarrow \text{true} \, , \\
&\quad \text{neq}(s(x),s(y)) \rightarrow \text{neq}(x,y) \} \\
\sigma(\text{geq}) &= \{ \text{geq}(0,0) \rightarrow \text{false} \, , \\
&\quad \text{geq}(s(x),0) \rightarrow \text{true} \, , \\
&\quad \text{geq}(0,s(x)) \rightarrow \text{false} \, , \\
&\quad \text{geq}(s(x),s(y)) \rightarrow \text{geq}(x,y) \} \\
\sigma(\text{gr}) &= \{ \text{gr}(0,x) \rightarrow \text{false} \, , \\
&\quad \text{gr}(s(x),0) \rightarrow \text{true} \, , \\
&\quad \text{gr}(s(x),s(y)) \rightarrow \text{gr}(x,y) \} \\
\sigma(\text{even}) &= \{ \text{even}(0) \rightarrow \text{true} \, , \\
&\quad \text{even}(s(0)) \rightarrow \text{false} \, , \\
&\quad \text{even}(s(s(x))) \rightarrow \text{even}(x) \} \\
\sigma(\text{odd}) &= \{ \text{odd}(0) \rightarrow \text{false} \, , \\
&\quad \text{odd}(s(0)) \rightarrow \text{true} \, , \\
&\quad \text{odd}(s(s(x))) \rightarrow \text{odd}(x) \}
\end{align*}
\]
Of course it is important to prove that the above transformation of the defined symbols provide the right rules and functionality required by the meaning of the defined symbol. In this chapter we will only see proofs for the following terms: 0, s, p, add, gr, eq. The proofs of the other expression within the set DefSymbols can be found in Appendix A.1 while the other conditions within the set DefSymbols can be found in Appendix A.2. We will first prove the expressions of the defined symbols, for which it is important they represent the correct natural number or boolean value.

Through the following lemmas \( \rightarrow \rightarrow_{F} \), where \( F = \bigcup \sigma(f) \) and \( f \in \text{DefSymbols}_{F} \), \( F \) is our original flowchart and \( \vec{v} \) is the vector of variables in \( F \). A natural number \( n \) will be represented as \( s^{n}(0) \), the successor applied \( n \) times on 0.

The first lemma, is a lemma to prove a property on the defined symbol add.

**Lemma 5.8.** Let \( m, n \in \mathbb{N} \), then \( \text{add}(s^{m}(0), s^{n}(0)) \rightarrow s^{m+n}(0) \)

**Proof.** With induction to \( m \).

**Case** \( m = 0 \)

\[
\text{add}(s^{0}(0), s^{n}(0)) = \begin{cases} s^{0}(0) = 0 \end{cases}
\]

\[
\text{add}(0, s^{n}(0)) = \begin{cases} \text{Rewrite Rule 1 of } \sigma(\text{add}) \end{cases}
\]

\[
s^{n}(0) = \begin{cases} 0 \text{ is unit type of } + \end{cases}
\]

\[
s^{0+n}(0)
\]

**Case** \( m > 0 \)

Induction Hypothesis: \( \text{add}(s^{m}(0), s^{n}(0)) \rightarrow s^{m+n}(0) \)

\[
\text{add}(s^{m+1}(0), s^{n}(0)) = \begin{cases} \text{Rewrite Rule 2 of } \sigma(\text{add}) \end{cases}
\]

\[
\text{s}(\text{add}(s^{m}(0), s^{n}(0))) = \begin{cases} \text{Induction Hypothesis} \end{cases}
\]

\[
s(s^{m+n}(0)) = \begin{cases} \text{Property s} \end{cases}
\]

\[
s^{m+1+n}(0)
\]
The following lemma is a lemma to prove that the defined symbols in $\Delta$ represent the correct natural number.

**Lemma 5.9.** Let $\Delta = \{0, s, p, add\}$, $t \in \Delta$ and $n = Val(t, v)$ then $t \rightarrow^* s^n(0)$.

**Proof.** The proof will use induction to the elements of $\Sigma'$ to prove the lemma.

**Case $t = 0$**

\[
\begin{align*}
  \text{Let } & t = 0, \\
  & s^0(0) \\
  \Rightarrow & [\text{Definition 4.5}] Val(0, v) \\
  \Rightarrow & s_{Val(0, v)}^0(0) \\
  \Rightarrow & [\text{Case } t = 0] s_{Val(0, v)}^0(0)
\end{align*}
\]

**Case $t = s(u)$**

Induction Hypothesis: $u \rightarrow^* s^{Val(u, v)}(0)$

\[
\begin{align*}
  \text{Let } & t = s(u), \\
  s(u) \rightarrow^* & [\text{Induction Hypothesis}] s(s_{Val(u, v)}(0)) \\
  \Rightarrow & [\text{Property } s] s_{Val(u, v)+1}(0) \\
  \Rightarrow & [\text{Definition 4.5}] Val(s(u), v) \\
  \Rightarrow & s_{Val(s(u), v)}(0) \\
  \Rightarrow & [\text{Case } t = s(u)] s_{Val(s(u), v)}(0)
\end{align*}
\]

**Case $t = \text{add}(u, v)$**

Induction Hypothesis: $u \rightarrow^* s^{Val(u, v)}(0) \land v \rightarrow^* s^{Val(v, v)}(0)$

\[
\begin{align*}
  \text{Let } & t = \text{add}(u, v), \\
  \text{add}(u,v) \rightarrow^* & [\text{Induction Hypothesis}] s(s^{Val(u, v)}(0), s^{Val(v, v)}(0)) \\
  \Rightarrow & [\text{Lemma 5.8}] s_{Val(u, v) + Val(v, v)}(0) \\
  \Rightarrow & [\text{Definition 4.5}] Val(\text{add}(u, v), v) \\
  \Rightarrow & s_{Val(\text{add}(u, v), v)}(0) \\
  \Rightarrow & [\text{Case } t = \text{add}(u, v)] s_{Val(\text{add}(u, v), v)}(0)
\end{align*}
\]
Case \( t = p(u) \)

Induction Hypothesis: \( u \rightarrow^* s^{\text{Val}(u, \overrightarrow{v})}(0) \).

Case distinction on \( \text{Val}(u, \overrightarrow{v}) \).

Case \( \text{Val}(u, \overrightarrow{v}) = 0 \)

\[
t = \begin{cases} 
\text{Case } t = p(u) \mid \text{Induction Hypothesis} \\
p(u) \rightarrow^* \begin{cases} 
\text{Induction Hypothesis} \\
p(s^{\text{Val}(u, \overrightarrow{v})}(0)) \begin{cases} 
\text{Val}(u, \overrightarrow{v}) = 0 \\
p(0) \begin{cases} 
\text{Definition } p \mid 0 \\
\text{Property } s \mid s^0(0) \\
\text{Val}(u, \overrightarrow{v}) = 0 \\
s_{\text{Val}(u, \overrightarrow{v})}(0) \begin{cases} 
\text{Definition } 4.5 \text{ Val}(p(u), \overrightarrow{v}) \mid s_{\text{Val}(u, \overrightarrow{v})}(0) \begin{cases} 
\text{Case } t = p(u) \mid s_{\text{Val}(u, \overrightarrow{v})}(0) \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases}
\]

Case \( \text{Val}(u, \overrightarrow{v}) > 0 \)

\[
t = \begin{cases} 
\text{Case } t = p(u) \mid \text{Induction Hypothesis} \\
p(u) \rightarrow^* \begin{cases} 
\text{Induction Hypothesis} \\
p(s^{\text{Val}(u, \overrightarrow{v})}(0)) \begin{cases} 
\text{Definition } 4.5 \text{ Val}(p(u), \overrightarrow{v}) \mid s_{\text{Val}(u, \overrightarrow{v})}(0) \begin{cases} 
\text{Definition } p \mid s_{\text{Val}(u, \overrightarrow{v})}(0) \begin{cases} 
\text{Case } t = p(u) \mid s_{\text{Val}(u, \overrightarrow{v})}(0) \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\n\]

\( \square \)
Next we will look to the conditions of the defined symbols. For the conditions $C$ we have to prove that for each condition $c \in C$ it holds that $c \rightarrow \mathbb{B}$. The conditions do not just evaluate to any boolean, but the boolean corresponding to the meaning of $c$. The following lemma will prove this property for the conditions $eq, gr \in C$.

**Lemma 5.10.** Take $t, u \in T(\Sigma)$, then

1. $eq(t, u) \rightarrow^* false$ if $Val(t, \overline{v}) \neq Val(u, \overline{v})$

2. $eq(t, u) \rightarrow^* true$ if $Val(t, \overline{v}) = Val(u, \overline{v})$

3. $gr(t, u) \rightarrow^* false$ if $Val(t, \overline{v}) \leq Val(u, \overline{v})$

4. $gr(t, u) \rightarrow^* true$ if $Val(t, \overline{v}) > Val(u, \overline{v})$

**Proof.** Take $n, m \in \mathbb{N}$, $n = Val(t, \overline{v})$ and $m = Val(u, \overline{v})$, then we can prove the statements with induction to these naturals. Due to Lemma 5.9 and the proofs within Appendix A.1 we have $t \rightarrow^* s^0(0)$ and $u \rightarrow^* s^0(0)$.

1. $eq(s^n(0), s^m(0)) \rightarrow^* false$ if $n \neq m$

   Proof with induction to $n$.

   **case $n = 0$**

   $eq(s^0(0), s^m(0))$

   $= \begin{cases} 
   s^0(0) = 0 \quad \text{if } n \neq m \\
   eq(0, s^m(0)) \quad \text{if } n = m > 0
   \end{cases}$

   $\rightarrow \quad \begin{cases} 
   \text{Rewrite Rule 3 of } \sigma(eq) \\
   false
   \end{cases}$

   **case $m = 0$**

   $eq(s^{n+1}(0), s^0(0))$

   $= \begin{cases} 
   s^0(0) = 0 \quad \text{if } n \neq m \\
   eq(s^{n+1}(0), 0) \quad \text{if } n = m > 0
   \end{cases}$

   $\rightarrow \quad \begin{cases} 
   \text{Rewrite Rule 2 of } \sigma(eq) \\
   false
   \end{cases}$

   **case $n > 0 \land m > 0$**

   Induction Hypothesis: ($\forall i : i < n \land i \neq j : eq(s^i(0), s^j(0)) \rightarrow^* false$)

   $eq(s^{n+1}(0), s^{m+1}(0))$

   $\rightarrow \quad \begin{cases} 
   \text{Rewrite Rule 4 of } \sigma(eq) \\
   eq(s^n(0), s^m(0)) \quad \text{if } n \neq m \\
   false \quad \text{if } n = m
   \end{cases}$

   $\rightarrow \quad \begin{cases} 
   \text{Induction Hypothesis} \\
   eq(s^n(0), s^m(0)) \rightarrow^* false
   \end{cases}$
2. \( \text{eq}(s^n(0), s^m(0)) \rightarrow^* \text{true} \) if \( n = m \)

Proof with induction to \( n \).

\textit{case} \( n = 0 \)

\[
\text{eq}(s^0(0), s^0(0)) \\
= \quad \{ s^0(0) = 0 \} \\
\rightarrow \quad \{ \text{Rewrite Rule 1 of } \sigma(\text{eq}) \} \\
\text{true}
\]

\textit{case} \( n > 0 \)

Induction Hypothesis: \( \text{eq}(s^n(0), s^m(0)) = \text{true} \)

\[
\text{eq}(s^{n+1}(0), s^{n+1}(0)) \\
\rightarrow \quad \{ \text{Rewrite Rule 4 of } \sigma(\text{eq}) \} \\
\text{eq}(s^n(0), s^m(0)) \\
= \quad \{ \text{Induction Hypothesis} \} \\
\text{true}
\]

3. \( \text{gr}(t, u) \rightarrow^* \text{false} \) if \( n \leq m \)

4. \( \text{gr}(t, u) \rightarrow^* \text{true} \) if \( n > m \)

These two statements we will prove together with induction to \( n \).

\textit{case} \( n = 0 \) (\( n \leq m \))

\[
\text{gr}(s^0(0), s^m(0)) \\
= \quad \{ s^0(0) = 0 \} \\
\text{eq}(0, s^m(0)) \\
\rightarrow \quad \{ \text{Rewrite Rule 1 of } \sigma(\text{gr}) \} \\
\text{false}
\]

\textit{case} \( n > 0 \land m = 0 \) (\( n > m \))

\[
\text{gr}(s^{n+1}(0), s^0(0)) \\
= \quad \{ s^0(0) = 0 \} \\
\text{gr}(s^{n+1}(0), 0) \\
= \quad \{ \text{Property s} \} \\
\text{gr}(s(s^0(0)), 0) \\
\rightarrow \quad \{ \text{Rewrite Rule 2 of } \sigma(\text{gr}) \} \\
\text{true}
\]
\[ \text{case } n > 0 \land m > 0 \]

Induction Hypothesis: \((\forall m : n \leq m : \text{gr}(s^n(0), s^m(0)) \rightarrow \text{false}) \land (\forall m : n > m : \text{gr}(s^n(0), s^m(0)) \rightarrow \text{true})\]

\[
\text{gr}(s^{n+1}(0), s^{m+1}(0)) \\
= \begin{cases} 
\text{Property } s & \\
\text{Rewrite Rule 3 of } \sigma(\text{gr}) & \\
\text{Induction Hypothesis} & \\
\text{false if } n \leq m & \\
\text{true if } n > m &
\end{cases}
\]

\[\square\]

**Decisions**

As described in the informal description there are two possibilities when a decision symbol has to be translated into a term rewriting system. The set \(D_F\) is the set of decision descriptions in the flowchart. The function \(\theta :: d \rightarrow \text{TRS}\) is the translation from \(d \in D_F\) to a term rewriting system.

**Definition 5.11. Transformation of Decisions**

In this transformation \(c_1, c_2, c_3 \in \text{Cond}, p_1 = x_1, \ldots, x_n := e_1, \ldots, e_n, p_2 = x_1, \ldots, x_n := f_1, \ldots, f_n\) and \(st_1, st_2, st_3 \in D_F\). Also \(c_1\) is the condition belonging to \(st_1\), \(c_2\) the condition belonging to \(st_2\) and \(c_3\) the condition belonging to \(st_3\).

\[
\theta(st_1 : \text{if } c_1 \ p_1 \ st_2 \ \text{else } p_2 \ st_3) = \\
\{sr1(true, x_1, \ldots, x_n) \rightarrow st2(c_2, e_1, \ldots, e_n),sr1(false, x_1, \ldots, x_n) \rightarrow st3(c_3, f_1, \ldots, f_n)\}
\]

And when the else branch is empty:

\[
\theta(st_1 : \text{if } c_1 \ p_1 \ st_2 \ \text{else } \text{bot}) = \\
\{sr1(true, x_1, \ldots, x_n) \rightarrow st2(c_2, e_1, \ldots, e_n)\}
\]

From the transformations in Definition 5.7 and Definition 5.11 we can now construct the total term rewriting system.

**Definition 5.12. Term Rewriting System of a flowchart**

The complete term rewriting system of a flowchart \(F\), \((\Pi_F)\) consists of the rules generated by the transformation of the decisions which appear in the flowchart combined with the transformation of the defined symbols which appear in the flowchart:

\[
\Pi_F = \{\theta(r) | r \in D_F\} \cup \sigma(f) \text{ where } f \in\text{DefSymbols}_F
\]

Unlike the transformation from an imperative program to a flowchart this transformation does not already exist, and so we will have to prove the transformation indeed preserves termination. So if the term rewriting system is concluded to be terminating also the original flowchart is terminating.
Lemma 5.13. Let $E_F$ be the set of edges within a transition system $G_F$ of a flowchart $F$ and $\Pi_F$ be a term rewriting system. If $(d, \vec{v}), (d', \vec{v'}) \in E_F$, then $\text{ter}(d, \vec{v}) \rightarrow^*_{\Pi_F} \text{ter}(d', \vec{v'})$

Proof. Let $(d, \vec{v})(d', \vec{v'}) \in E_F$ then there are three possibilities for how the decision description in the original flowchart looks like. Either $d : \text{if } c \text{ then } p_1 \text{ else } p_2 \text{ else } \text{bot} \in D_F$, $d : \text{if } c \text{ then } p_1 \text{ else } p_2 \text{ else } p_2' \in D_F$ or $d : \text{if } c \text{ then } d_1 \text{ else } p_2 \text{ else } p_2' \in D_F$.

When the originating decision description from $D_F$ is $d : \text{if } c \text{ then } p_1 \text{ else } p_2 \text{ else } \text{bot}$, we transform it into rules for $\Pi_F$ according to Definition 5.11 like $d(\text{true}, x_1, \ldots, x_n) \rightarrow d'(c_\varphi, e_1, \ldots, e_n)$ where $c_\varphi$ is the condition of $d' \in D_F$. The true of this rewrite rule which originates from the evaluation of the condition of $d$, $\text{Val}(c, \vec{v}) = \text{true}$. The evaluation of the condition is transformed into rules for $\Pi_F$ according to Definition 5.11 and proved to be correct in Lemma 5.10 and the proofs in Appendix A.2. The $x_1, \ldots, x_n$ is equal to $\vec{v}$ and $e_1, \ldots, e_n$ equals the expressions belonging to $\vec{v}$ after the assignments in $p_1$. The evaluation of these expressions result in $\vec{v'}, \vec{v'} = \phi(p_1, \vec{v})$. This evaluation is transformed into rules in $\Pi_F$ according to Definition 5.11 and proved to be correct in Lemma 5.10 and the proofs in Appendix A.2.

When the originating decision description from $D_F$ is $d : \text{if } c \text{ then } p_1 \text{ else } p_2 \text{ else } p_2'$, we transform it into rules for $\Pi_F$ according to Definition 5.11 like $d(\text{false}, x_1, \ldots, x_n) \rightarrow d'(c_\varphi, e_1, \ldots, e_n)$ where the first rule is explained as the rule in the previous case. In the second rule $c_{d_2}$ is the condition of $d_2 \in D_F$, the false originates from the evaluation of the condition of $d$, $\text{Val}(c, \vec{v}) = \text{false}$. The evaluation of the condition is transformed into rules for $\Pi_F$ according to Definition 5.11 and proved to be correct in Lemma 5.10 and the proofs in Appendix A.2. The $x_1, \ldots, x_n$ is equal to $\vec{v}$ and $e_1, \ldots, e_n$ equals the expressions belonging to $\vec{v}$ after the assignments in $p_2$. The evaluation of these expressions result in $\vec{v'}, \vec{v'} = \phi(p_1, \vec{v})$. This evaluation is transformed into rules in $\Pi_F$ according to Definition 5.11 and proved to be correct in Lemma 5.10 and the proofs in Appendix A.2.

When the originating decision description from $D_F$ is $d : \text{if } c \text{ then } d_1 \text{ else } p_2 \text{ else } p_2'$, we transform it into rules for $\Pi_F$ according to Definition 5.11 like $d(\text{true}, x_1, \ldots, x_n) \rightarrow d'_1(c_\varphi, f_1, \ldots, f_n)$, $d(\text{false}, x_1, \ldots, x_n) \rightarrow d'_2(c_\varphi, f_1, \ldots, f_n)$, where $c_{d_1}$ is the condition of $d_1 \in D_F$. The rest of this rule can be explained similar to the previous case.

So if we have a $(d, \vec{v})(d', \vec{v'}) \in E_F$ then we have a rewrite sequence in $\Pi_F$ ($\rightarrow^*_{\Pi_F}$) from $\text{ter}(d, \vec{v})$ to $\text{ter}(d', \vec{v'})$ which contains the evaluation of the condition and the evaluation of the expressions. □

Theorem 5.14. Termination Preservation Transformation

If $\Pi_F$ is terminating, then also the original $F$ is terminating.

Proof. Assume the original $F$ is not terminating. So there is an infinite sequence of edges in $G_F$, thus $\forall i :: (d_i, \vec{v}_i), (d_{i+1}, \vec{v}_{i+1}) \in E_F$, see Definition 4.10. According to Lemma 5.13 we then have $\forall i :: \text{ter}(d_i, \vec{v}_i) \rightarrow^*_{\Pi_F} \text{ter}(d_{i+1}, \vec{v}_{i+1})$, contradicting the termination of $\Pi_F$ (Definition 3.11). □
Chapter 6

Examples

As a result of the transformation we described several imperative programs and transformed them into term rewriting systems. A total of 24 examples were submitted into the problem database of the Termination Competition 2007 [TerComp] under the name Beerendonk - 1 to Beerendonk - 24 in the TRS sub-category Standard. In this chapter we will first discuss some successful examples, which were all submitted to the competition database. Also we will show some examples which were not successful.

6.1 Successful Examples

All the examples in this section are successful which means that AProVE could find a termination proof for the term rewriting systems constructed by our transformation from an imperative program. All other termination tools could not find a conclusive proof for termination. Due to Theorem 5.14, with the termination proof for the term rewriting system also the original imperative program was proved to be terminating.
6.1.1 Beerendonk - 1

This example is a program where we compare two variables and while the first argument is greater than the second argument we decrease the first argument.

**Imperative Program**

```plaintext
while gr(x, y)
do
    x := p(x)
od
```

**Flowchart**  If we construct a graphical flowchart of this program it will look like this:

![Flowchart Diagram](image)

This flowchart can be denoted textual as:

```plaintext
dec1: if gr(x, y) x := p(x) dec1
else ⊥
```

**Term Rewriting System**  This flowchart can be, according to the transformations of the defined symbols (Definition 5.7) and the decisions (Definition 5.11) to the following rewrite rules:

- \( \text{dec}(\text{true}, x, y) \rightarrow \text{dec}(\text{gr}(x, y), p(x), y) \)
- \( \text{gr}(0, x) \rightarrow \text{false} \)
- \( \text{gr}(s(x), 0) \rightarrow \text{true} \)
- \( \text{gr}(s(x), s(y)) \rightarrow \text{gr}(x, y) \)
- \( p(0) \rightarrow 0 \)
- \( p(s(x)) \rightarrow x \)
6.1.2 Beerendonk - 4

This example is a program where the loop only will be run once, because we loop while the first argument is greater than the second argument. Inside the loop body we swap the two arguments.

**Imperative Program**

```
while gr(x, y)
  do
    x, y := y, x;
  od
```

**Flowchart** If we construct a graphical flowchart of this program it will look like this:

![Flowchart](chart.png)

This flowchart can be denoted textual as:

```
dcl: if gr(x, y) x, y := y, x dcl
else ⊥
```

**Term Rewriting System** This flowchart can be, according to the transformations of the defined symbols (Definition 5.7) and the decisions (Definition 5.11) to the following rewrite rules:

- `dec(true, x, y) → dec(gr(x, y), y, x)`
- `gr(0, x) → false`
- `gr(s(x), 0) → true`
- `gr(s(x), s(y)) → gr(x, y)`

Note that according to the grammar of our imperative input language the construction `x, y := y, z` is not allowed. If we follow the grammar we have to introduce a temporary variable `z`. Like this: `z := x; x := y; y := z`. If we input a term rewriting system with this loop body AProVE also determines the term rewriting system to be terminating.
This example shows that not only comparing to 0, but also comparing to an arbitrary variable (in this case $z$) can be used. The example is a program which compares if two variables are greater than a given value $z$. If both are greater than the value, they are decreased within the loop body.

**Imperative Program**

```plaintext
while and(gr(x, z), gr(y, z))
  do
    x := p(x);
    y := p(y)
  od
```

**Flowchart**  If we construct a graphical flowchart of this program it will look like this:

![Flowchart Diagram](image.png)

This flowchart can be denoted textual as:

```plaintext
dec1: if and(gr(x, z), gr(y, z))
  then x := p(x); y := p(y)
else⊥
```

**Term Rewriting System**  This flowchart can be, according to the transformations of the defined symbols (Definition 5.7) and the decisions (Definition 5.11) to the following rewrite rules:

- $dec(true, x, y, z) \rightarrow dec(and(gr(x, z), gr(y, z)), p(x), p(y), z)$
- $and(true, true) \rightarrow true$
- $and(x, false) \rightarrow false$
- $and(false, x) \rightarrow false$
- $gr(0, 0) \rightarrow false$
- $gr(0, x) \rightarrow false$
- $gr(s(x), 0) \rightarrow true$
- $gr(s(x), s(y)) \rightarrow gr(x, y)$
- $p(0) \rightarrow 0$
- $p(s(x)) \rightarrow x$
6.1.4 Beerendonk - 10

This example shows that nesting of a conditional statement within a loop is also a possibility. This program loops while the argument is unequal to 0. In the loop body if the argument is an even number we divide the number by 2, otherwise we decrease the value of the argument.

**Imperative Program**

while neq(x,0)
    do
        if even(x) then x := div2(x)
            else x := p(x)
    od

**Flowchart** If we construct a graphical flowchart of this program it will look like this:

This flowchart can be denoted textual as:

```
dec1: if neq(x,0) then skip dec2
    else ⊥

dec2: if even(x) then x := div2(x) dec1
    else x := p(x) dec1
```
Term Rewriting System  This flowchart can be, according to the transformations of the defined symbols (Definition 5.7) and the decisions (Definition 5.11) to the following rewrite rules:

\[
\begin{align*}
&\text{dec1}(\text{true}, x) \Rightarrow \text{dec2}(\text{even}(x), x) \\
&\text{dec2}(\text{true}, x) \Rightarrow \text{dec1}(\text{neq}(x, 0), \text{div2}(x)) \\
&\text{dec2}(\text{false}, x) \Rightarrow \text{dec1}(\text{neq}(x, 0), \text{p}(x)) \\
&\text{neq}(0, 0) \Rightarrow \text{false} \\
&\text{neq}(0, \text{s}(x)) \Rightarrow \text{true} \\
&\text{neq}(\text{s}(x), 0) \Rightarrow \text{true} \\
&\text{neq}(\text{s}(x), \text{s}(y)) \Rightarrow \text{neq}(x, y) \\
&\text{even}(0) \Rightarrow \text{true} \\
&\text{even}(\text{s}(0)) \Rightarrow \text{false} \\
&\text{even}(\text{s}(\text{s}(x))) \Rightarrow \text{even}(x) \\
&\text{div2}(0) \Rightarrow 0 \\
&\text{div2}(\text{s}(0)) \Rightarrow 0 \\
&\text{div2}(\text{s}(\text{s}(x))) \Rightarrow \text{s}(\text{div2}(x)) \\
&\text{p}(0) \Rightarrow 0 \\
&\text{p}(\text{s}(x)) \Rightarrow x
\end{align*}
\]

6.1.5 Beerendonk - 15

This example shows that multiple nesting is also covered by the transformation

**Imperative Program**

```plaintext
while gr(add(x, y), 0)
do
    if gr(x, 0) then x := p(x)
    else if gr(y, 0) then y := p(y)
    else skip
    fi
fi
od
```
**Flowchart**  If we construct a graphical flowchart of this program it will look like this:

![Flowchart Diagram]

This flowchart can be denoted textual as:

```plaintext
dec1: if gr(add(x, y), 0) skip dec2
    else ⊥

dec2: if gr(x, 0) x := p(x) dec1
    else skip dec3

dec3: if gr(y, 0) y := p(y) dec1
    else skip
```

**Term Rewriting System**  This flowchart can be, according to the transformations of the defined symbols (Definition 5.7) and the decisions (Definition 5.11) to the following rewrite rules:

```plaintext
dec1(true, x, y) → dec2(gr(x), 0), x, y)
dec2(true, x, y) → dec1(gr(add(x, y), 0), p(x), y)
dec2(false, x, y) → dec3(gr(y), 0), x, y)
dec3(true, x, y) → dec1(gr(add(x, y), 0), x, p(y))
dec3(false, x, y) → dec1(gr(add(x, y), 0), x, y)

gr(0, x) → false
gr(s(x), 0) → true
gr(s(x), s(y)) → gr(x, y)
add(0, x) → x
add(s(x), y) → s(add(x, y))
p(0) → 0
p(s(x)) → x
```
6.1.6 Beerendonk - 21

This example shows nesting of loops within loops.

**Imperative Program**

```
while and(eq(x, y), gr(x, z))
do
  while gr(y, z)
do
    y := p(y);
x := p(x)
  od
od
```

**Flowchart**  
If we construct a graphical flowchart of this program it will look like this:

![Flowchart Image]

This flowchart can be denoted textual as:

```
dec1: if and(eq(x, y), gr(x, z)) skip dec2 else ⊥
dec2: if gr(y, z) y := p(y); x := p(x) dec1 else skip dec1
```
Term Rewriting System  This flowchart can be, according to the transformations of the defined symbols (Definition 5.7) and the decisions (Definition 5.11) to the following rewrite rules:

\[
\begin{align*}
decl & (true, x, y, z) \rightarrow \decl (gr(y, z), x, y, z) \\
decl & (true, x, y, z) \rightarrow \decl (gr(y, z), p(x), p(y), z) \\
decl & (false, x, y, z) \rightarrow \decl (and(eq(x, y), gr(x, z)), x, y, z) \\
gr & (0, x) \rightarrow false \\
gr & (s(x), 0) \rightarrow true \\
gr & (s(x), s(y)) \rightarrow gr(x, y) \\
p & (0) \rightarrow 0 \\
p & (s(x)) \rightarrow x \\
eq & (0, 0) \rightarrow true \\
eq & (s(x), 0) \rightarrow false \\
eq & (0, s(x)) \rightarrow false \\
eq & (s(x), s(y)) \rightarrow eq(x, y) \\
and & (true, true) \rightarrow true \\
and & (false, x) \rightarrow false \\
and & (x, false) \rightarrow false
\end{align*}
\]
6.2 Unsuccessful Examples

For the examples in this section AProVE was not able to find a conclusive termination proof. Hence there is no conclusion on termination of our imperative program, while to a programmer it is obvious these imperative programs will terminate.

6.2.1 GCD - Euclid’s Algorithm

This example shows the complete transformation of Euclid’s GCD Algorithm introduced in 2.2.

Imperative Program  We recall that the imperative program looks like this:

while gr(x, 0)
  do
    if gr(x, y) then x := minus(x, y)
      else z := x; x := y; y := z
  od

Flowchart  If we construct a graphical flowchart of this algorithm it looks like this:

This flowchart can be denoted textual as follows:

decl1: if gr(x, 0) skip dec2
else bot
decl2: if gr(x, y) x := minus(x, y) dec1
else z := x; x := y; y := z dec1

*For these tests we used the web-interface on the site of AProVE [AProVE]
**Term Rewriting System**  This flowchart can be, according to the transformations of the defined symbols (Definition 5.7) and the decisions (Definition 5.11) to the following rewrite rules:

\[
\begin{align*}
gr(0, x) & \rightarrow \text{false} \\
gr(s(x), 0) & \rightarrow \text{true} \\
gr(s(x), s(y)) & \rightarrow gr(x, y) \\
\text{minus}(0, x) & \rightarrow 0 \\
\text{minus}(x, 0) & \rightarrow x \\
\text{minus}(s(x), s(y)) & \rightarrow \text{minus}(x, y) \\
\text{dec1}(\text{true}, x, y, z) & \rightarrow \text{dec2}(\text{gr}(x, y), x, y, z) \\
\text{dec2}(\text{true}, x, y, z) & \rightarrow \text{dec1}(\text{gr}(x, 0), \text{minus}(x, y), y, z) \\
\text{dec2}(\text{false}, x, y, z) & \rightarrow \text{dec1}(\text{gr}(x, 0), y, x, x)
\end{align*}
\]

**6.2.2 Simple**

After the conclusion of the previous example, that there still are imperative programs for which termination could not be shown, the question arises what the smallest problem is, which can not be evaluated to be terminating. We found a very simple program which still cannot be proved:

```plaintext
while gr(add(x, y), 0) 
do 
  x := p(x); 
y := p(y) 
od
```

**Flowchart**  If we construct a graphical flowchart of this algorithm it looks like this:

![Flowchart](image_url)

This flowchart can be denoted textual as follows:

```plaintext
\text{dec1}: \text{if} \quad \text{gr}(\text{add}(x, y), 0) \quad x := p(x); y := p(y) \quad \text{dec1}
\text{else} \quad \text{bot}
```
**Term Rewriting System**  This flowchart can be, according to the transformations of the defined symbols (Definition 5.7) and the decisions (Definition 5.11) to the following rewrite rules:

\[
\begin{align*}
\text{add}(0, y) & \rightarrow y \\
\text{add}(s(x), y) & \rightarrow s(\text{add}(x, y)) \\
\text{gr}(0, x) & \rightarrow \text{false} \\
\text{gr}(s(x), 0) & \rightarrow \text{true} \\
\text{gr}(s(x), s(y)) & \rightarrow \text{gr}(x, y) \\
p(0) & \rightarrow 0 \\
p(s(x)) & \rightarrow x \\
\text{dec1}(\text{true}, x, y) & \rightarrow \text{dec1}(\text{gr}(\text{add}(x, y), 0), p(x), p(y))
\end{align*}
\]
Chapter 7

Conclusions

To verify whether an imperative program is correct among other things a proof for termination is needed. A way to automatically prove termination of an imperative program is to transform it into a term rewriting system which preserves the termination property. Proving termination of that term rewriting system implies the termination of the original program. Just like proving termination on the original program, proving termination of a term rewriting system is considered to be undecidable. However, techniques for proving termination (and tools which implement these techniques) exist for a great range of term rewriting systems.

For two reasons the transformation from an imperative program to a term rewriting system has a flowchart as an intermediate structure. One is because the flowchart is a straightforward visualization of an imperative program and secondly because the traces within the flowchart are similar to the rewrite sequences within a term rewriting system. With the researched transformation only a subset of the imperative programs can be transformed into a term rewriting system, for example initializations are still prohibited. The grammar of the input program is also extendable with more constructions, functionality and data types.

The transformation from an imperative program to an intermediate flowchart is a standard and straightforward transformation which was implemented in a java tool named imp2fc. The transformation from the flowchart to a term rewriting system was not standard and thus we described this transformation in mathematical detail and we have proved this transformation to be termination preserving. For proving this property of the transformation we had to define a transition system of the flowchart from which we can prove whether a flowchart terminates. Because of this transition system we can prove that if there is an edge in that transition system of the flowchart we have to have one or more rewrite steps in the term rewriting system of the flowchart, which leads to the final proof for termination preservation. This transformation was also implemented in a java tool named fc2trs.

The examples have shown that (using AProVE) it is possible to prove termination on several kinds of imperative programs. However a practical example like the gcd algorithm has shown us that there are still programs for which no conclusive termination proof can be found. Also AProVE is, up to now, the only termination tool on term rewriting systems which can conclude termination of these generated term rewriting systems. All other tools in the termination competition were unable to find a termination proof.
7.1 Future Work

The subset of imperative programs which can be transformed into a term rewriting system can be extended by implementing more possible structures, more language constructions, more functionality and extra data types.

- The structures of the imperative language can be extended by defining a transformation (or altering the current transformation) which allows for example initializations and consecutive execution of language constructions.

- The language constructions in the imperative language can be extended by adding for example function calls and for-loops to the imperative grammar and the transformation.

- The functionality in the imperative language can be extended by adding expressions and conditions like multiplication and equality on the booleans to the imperative grammar and the transformation.

- The set of data types in the imperative language can be extended by adding for example strings and arrays to the imperative grammar and the transformation.

The subset of term rewriting systems for which the termination tools (like AProVE and Jambox) can prove termination is still growing. For example the examples in the competition database could not be proved by the previous version of AProVE. Still new implementations for the termination proof techniques are being researched, which can contribute in proving more transformed imperative programs.
Bibliography

[AProVE]  \textit{AProVE}. \url{http://aprove.informatik.rwth-aachen.de/}.


[Jambox] \textit{Jambox}. \url{http://joerg.endrullis.de/}.


Appendix A

Proofs

A.1 Expressions

In this section we give the proofs for the expressions within the DefSymbols which are not discussed within the main report. Through this appendix $\rightarrow_{z} \rightarrow_{F}$, where $F = \bigcup \sigma(f)$ and $f \in \text{DefSymbols}_{F}$, $F$ is our original flowchart and $v$ is the vector of variables in $F$. A natural number $n$ will be represented as $s^{n}(0)$, the successor applied $n$ times on 0.

**div2**

The next lemma is a lemma to prove a property on the defined symbol $\text{div2}$. It will be needed to prove that $\text{div2}$ will represent the correct natural number.

**Lemma A.1.** Let $m \in \mathbb{N}$, then $\text{div2}(s^{m}(0)) \rightarrow^{*} s^{m/2}(0)$

*Proof.* With induction to $m$.

**Case $m = 0$**

$$ \text{div2}(s^{0}(0)) $$

\[
= \{ s^{0}(0) = 0 \} \text{div2}(0)
\]

\[
= \{ \text{Rewrite Rule 1 of } \sigma(\text{div2}) \}
\]

\[
0
\]

\[
= \{ s^{0}(0) = 0 \}
\]

\[
= \text{s}^{0}(0)
\]

\[
= \{ 0/2 = 0 \}
\]

\[
= \text{s}^{0/2}(0)
\]

**Case $m = 1$**

$$ \text{div2}(s^{1}(0)) $$

\[
= \{ s^{1}(0) = s(0) \} \text{div2}(s(0))
\]

\[
= \{ \text{Rewrite Rule 2 of } \sigma(\text{div2}) \}
\]

\[
0
\]

\[
= \{ s^{0}(0) = 0 \}
\]

\[
= \text{s}^{0}(0)
\]

\[
= \{ 1/2 = 0 \}
\]

\[
= \text{s}^{1/2}(0)
\]

\[
= \{ 1 = s(0) \}
\]

\[
= \text{s}^{s(0)/2}(0)
\]
Case \( m > 1 \)

Induction Hypothesis: \( \text{div2}(s^m(0)) \rightarrow^* s^{m/2}(0) \)

\[
\text{div2}(s^{m+2}(0)) = \begin{cases} s^{m+2}(0) = s(s(s^m(0))) \end{cases}
\]

\[
\text{div2}(s(s(s^m(0)))) = \begin{cases} \text{Rewrite Rule 3 of } \sigma(\text{div2}) \end{cases}
\]

\[
s(\text{div2}(s^m(0))) = \begin{cases} \text{Induction Hypothesis} \end{cases}
\]

\[
s(s^{m/2}(0)) = \begin{cases} \text{Property } s \end{cases}
\]

\[
s^{(m/2)+1}(0) = \begin{cases} \text{Algebra} \end{cases}
\]

\[
s^{(m+2)/2}(0)
\]

\[
\square
\]

The next lemma proves that for the keyword \( \text{div2} \) the result will always be the natural number it represents.

**Lemma A.2.** Let \( \Sigma' = \{ \text{div2} \}, t \in T(\Sigma') \) and \( n = V(t, \overline{t}) \) then \( t \rightarrow^* s^n(0) \).

**Proof.** Case \( t = \text{div2}(u) \)

Induction Hypothesis: \( u \rightarrow^* s^{V(u, \overline{t})}(0) \)

\[
t = \begin{cases} \text{Case } t = \text{div2}(u) \end{cases}
\]

\[
\text{div2}(u) \rightarrow^* \begin{cases} \text{Induction Hypothesis} \end{cases}
\]

\[
(s^{V(u, \overline{t})}(0)) = \begin{cases} \text{Lemma A.1} \end{cases}
\]

\[
s^{V(u, \overline{t})/2}(0) = \begin{cases} \text{Definition 4.5} \end{cases}
\]

\[
s^{V(\text{div2}(u), \overline{t})/2}(0) = \begin{cases} i = \text{div2}(u) \end{cases}
\]

\[
s^{V(i, \overline{t})}(0)
\]

\[
\square
\]
Lemma A.3. Let \( m \in \mathbb{N} \), then \( \text{minus}(s^m(0), s^n(0)) \rightarrow^* s^{m-n}(0) \)

Proof. With induction to \( m \).

Case \( m = 0 \)

\[
\text{minus}(s^0(0), s^n(0)) \equiv s^0(0) = 0 \\
\text{by rewrite rule 1 of } \sigma(\text{minus}) \\
= 0 \\
\text{by rewrite rule 2 of } \sigma(\text{minus}) \\
s^0(0) \\
= \begin{cases} 
0/2 = 0 \\
s^0-\cdot-n(0)
\end{cases}
\]

Case \( m > 0 \)

Induction Hypothesis: \( \text{minus}(s^m(0), s^n(0)) \rightarrow^* s^{m-n}(0) \)

Case distinction on \( n \).

Case \( n = 0 \)

\[
\text{minus}(s^{m+1}(0), s^0(0)) \equiv s^{m+1}(0) \\
\text{by rewrite rule 2 of } \sigma(\text{minus}) \\
= s^{m+1-0}(0) \\
\]

Case \( n > 0 \)

\[
\text{minus}(s^{m+1}(0), s^{n+1}(0)) \equiv s^{m+1}(0), s^{n}(0) \\
\text{by rewrite rule 3 of } \sigma(\text{minus}) \\
= \text{induction hypothesis} \\
= s^{m+1-n-1}(0) \\
\text{by algebra} \\
= s^{m+1-\cdot-(n+1)}(0)
\]
The next lemma proves that for the keyword \textit{minus} the result will always be the natural number it represents.

**Lemma A.4.** Let $\Sigma' = \{\text{minus}\}$, $t \in T'($\Sigma$')$ and $n = V(t, \vec{v})$ then $t \rightarrow^n s^n(0)$.

**Proof.** Case $t = \text{minus}(u, v)$

Induction Hypothesis: $u \rightarrow^s s^{V(u, \vec{v})}(0) \land v \rightarrow^s s^{V(v, \vec{v})}(0)$

\[
\begin{align*}
t & = \text{Case } t = \text{minus}(u, v) \\Rightarrow \text{[ Induction Hypothesis ]} \\Rightarrow \text{[ Lemma A.3 ]} \\
& = \text{[ Definition 4.5 $V(\text{minus}(t, \vec{v}))$ ]} \\
& = \text{[ $t = \text{minus}(u, v)$ ]} \\
& = s^{V(u, \vec{v})}(0)
\end{align*}
\]
A.2 Conditions

This section contains the proofs of the conditions within DefSymbols which are not discussed within the main report.

**neq**

The following lemma will prove this property for the condition \( \text{neq} \in C \).

**Lemma A.5.** Take \( t, u \in T(\Sigma) \), then
1. \( \text{neq}(t, u) \rightarrow^* \text{true} \) if \( V(t, \vec{v}) \neq V(u, \vec{v}) \)
2. \( \text{neq}(t, u) \rightarrow^* \text{false} \) if \( V(t, \vec{v}) = V(u, \vec{v}) \)

**Proof.** Take \( n, m \in \mathbb{N}, n = V(t, \vec{v}) \) and \( m = V(u, \vec{v}) \), then we can prove the statements with induction to these naturals. Due to Lemma 5.9, we have \( t \rightarrow^* \text{s}^n(0) \) and \( u \rightarrow^* \text{s}^m(0) \).

1. \( \text{neq} (\text{s}^n(0), \text{s}^m(0)) \rightarrow^* \text{true} \) if \( n \neq m \)

Proof with induction to \( n \).

**case** \( n = 0 \)

\[
\text{neq}(\text{s}^0(0), \text{s}^m(0)) = \begin{cases} \text{true} \quad \text{if } n \neq m \end{cases}
\]

**case** \( m = 0 \)

\[
\text{neq}(\text{s}^n(0), \text{s}^0(0)) = \begin{cases} \text{true} \quad \text{if } n \neq m \end{cases}
\]

**case** \( n > 0 \land m > 0 \)

Induction Hypothesis: \( (\forall i : i < n \land i \neq j : \text{neq}(\text{s}^i(0), \text{s}^j(0)) \rightarrow^* \text{true}) \)

\[
\text{neq}(\text{s}^{n+1}(0), \text{s}^{m+1}(0)) = \begin{cases} \text{true} \quad \text{Induction Hypothesis} \end{cases}
\]

56
2. \textit{neq}(s^n(0), s^m(0)) \rightarrow^* \textbf{false} \textbf{ if } n = m \\
Proof with induction to \( n \).

\textit{case} \( n = 0 \)

\textit{neq}(s^0(0), s^m(0))

\begin{align*}
\textbf{false} & \\
\textit{neq}(s^0(0), s^0(0)) & \\
\textbf{false} & \\
\textit{false} & \\
\textit{Rewrite Rule 1 of } \sigma(\textit{neq}) & \\
\textbf{false} & \\
\textit{false} & \\
\end{align*}

\textit{case} \( n > 0 \)

Induction Hypothesis: \( (\forall i : i < n \land i = j : \textit{neq}(s^i(0), s^j(0)) \rightarrow^* \textbf{false}) \)

\textit{neq}(s^{i+1}(0), s^{m+1}(0))

\begin{align*}
\textbf{false} & \\
\textit{Rewrite Rule 4 of } \sigma(\textit{neq}) & \\
\textit{false} & \\
\textit{Induction Hypothesis} & \\
\textbf{false} & \\
\end{align*}

\( \square \)

\textbf{geq}

The following lemma will prove this property for the condition \( \textbf{geq} \in C \).

\textbf{Lemma A.6.} \textbf{Take } t, u \in T(\Sigma), \textbf{ then}

1. \( \textbf{geq}(t, u) \rightarrow^* \textbf{true} \textbf{ if } V(t, \overline{v}) \geq V(u, \overline{v}) \)

2. \( \textbf{geq}(t, u) \rightarrow^* \textbf{false} \textbf{ if } V(t, \overline{v}) < V(u, \overline{v}) \)

\textbf{Proof.} \textbf{Take } n, m \in \mathbb{N}, n = V(t, \overline{v}) \textbf{ and } m = V(u, \overline{v}), \textbf{ then we can prove the statements with induction to these naturals. Due to Lemma[5,9] we have } t \rightarrow^* s^n(0) \textbf{ and } u \rightarrow^* s^m(0). \)

1. \( \textbf{geq}(s^n(0), s^m(0)) \rightarrow^* \textbf{true} \textbf{ if } n \geq m \)

\textbf{Proof with induction to } \( n \).

\textit{case} \( n = 0 \)

\textit{geq}(s^0(0), s^m(0))

\begin{align*}
\textbf{true} & \\
\textit{geq}(s^0(0), s^0(0)) & \\
\textbf{true} & \\
\textit{true} & \\
\textit{Rewrite Rule 1 of } \sigma(\textit{geq}) & \\
\textbf{true} & \\
\textit{false} & \\
\end{align*}

\textit{case} \( n > 0 \land m = 0 \)

\textit{geq}(s^{i+1}(0), s^0(0))

\begin{align*}
\textbf{false} & \\
\textit{true} & \\
\textit{Property s} & \\
\textit{true} & \\
\textit{Rewrite Rule 2 of } \sigma(\textit{geq}) & \\
\textbf{true} & \\
\textit{false} & \\
\end{align*}

57
\[\text{case } n > 0 \land m > 0\]

Induction Hypothesis: (\(\forall i : i < n \land i \geq j : \text{geq}(s^i(0), s^j(0)) \rightarrow^* \text{true}\))

\[
\text{geq}(s^{n+1}(0), s^{m+1}(0))
\rightarrow \quad \{	ext{Rewrite Rule 4 of } \sigma(\text{geq})\}
\text{geq}(s^n(0), s^m(0))
= \quad \{	ext{Induction Hypothesis}\}
\text{true}
\]

2. \(\text{geq}(s^n(0), s^m(0)) \rightarrow^* \text{false}\) if \(n < m\)

Proof with induction to \(n\).

\text{case } n = 0

\[
\text{geq}(s^0(0), s^m(0))
= \quad \{ s^0(0) = 0 \}
\text{geq}(0, s^m(0))
= \quad \{ n > m \rightarrow m > 0 \}
\text{geq}(0, s(s^m-1)(0))
\rightarrow \quad \{	ext{Rewrite Rule 3 of } \sigma(\text{geq})\}
\text{false}
\]

\text{case } n > 0

Induction Hypothesis: (\(\forall i : i < n \land i < j : \text{geq}(s^i(0), s^j(0)) \rightarrow^* \text{false}\))

\[
\text{geq}(s^{n+1}(0), s^{m+1}(0))
\rightarrow \quad \{	ext{Rewrite Rule 4 of } \sigma(\text{geq})\}
\text{geq}(s^n(0), s^m(0))
= \quad \{	ext{Induction Hypothesis}\}
\text{false}\]

\(\Box\)

\textbf{and, or, not}

The proof for the correctness of the defined symbols and, or and not directly can be concluded from the rewrite rules of \(\sigma(\text{and})\), \(\sigma(\text{or})\) and \(\sigma(\text{not})\) by pattern matching.