Program Specialization as a Tool for Solving Word Equations

Antonina Nepeivoda

Program Systems Institute of Russian Academy of Sciences*
Pereslavl-Zalessky, Russia
a_nevod@mail.ru

The paper focuses on the automatic generating of the witnesses for the word equation satisfiability problem by means of specializing an interpreter $\mathrm{WI}(\sigma, \mathscr{E}qs)$, which tests whether a substitution σ of variables of a given word equation system $\mathscr{E}qs$ produces its solution. We specialize such an interpreter *w.r.t.* $\mathscr{E}qs$, while σ is unknown. We show that several variants of such interpreters, when specialized using the basic unfold/fold specialization methods, are able to decide the satisfiability problem for some sets of the word equations whose left- and right-hand sides share variables. We prove that the specialization process *w.r.t.* the constructed interpreters is sound, *i.e.* gives a simple syntactic criterion of the satisfiability, and compare the results of the suggested approach with the results produced by `Z3str3` and `CVC4`, the widely-used SMT-solvers.

1 Introduction

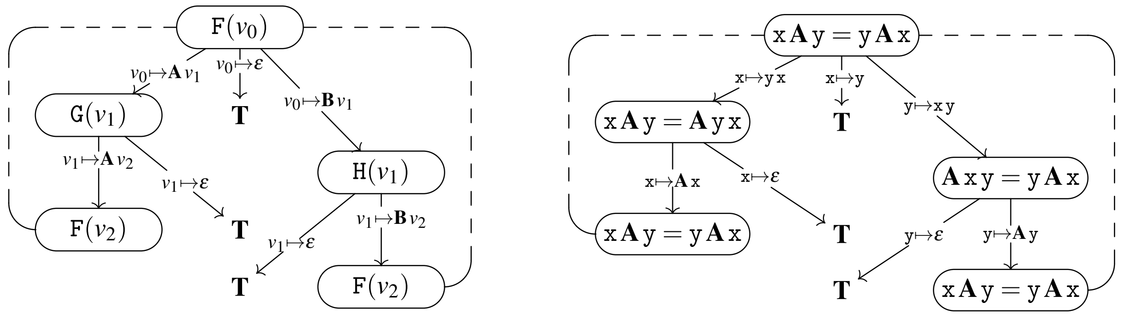
In recent decades, program transformation techniques were applied to verification and analysis of properties of several models, including cache-coherence and cryptographic protocols, constrained Horn clauses, Petri nets, deductive databases, control-flow analysis, and others [2, 10, 11, 13, 22, 31, 42]. On the other hand, the number of works on verification of string manipulating programs and string constraint solvers is rapidly growing during last years [1, 4, 6, 8, 18, 19, 20, 23, 34, 38, 43]. As far as we know, there are few interactions between the two research areas, although some of their methods exploit similar concepts.

One approach to the verification is to apply an unfold/fold algorithm [5] to a deterministic program modelling a nondeterministic system behaviour via introducing an additional path parameter [22, 21]. That is, given an unfold/fold algorithm `Spec` and a non-deterministic program $f(\mathtt{x})$, the algorithm `Spec` solves the specialization task $f'(v, \mathtt{x})$ satisfying the condition[1] $\forall c\,(\exists f(c) \Rightarrow \exists p\,(f'(p,c) = f(c)))$. Here the parameter v ranges over the paths determining the ways to compute $f(\mathtt{x})$.

This idea has many applications in computer science. In particular, methods to solve word equations, starting from Matyiasevich's [25], Hmelevskij's [14], Makanin's [24] algorithms in 1970s, and including Plandowski's [33] and Jez's algorithms [15] designed in the recent two decades, all use the non-deterministic search. A word equation is an equation $\Phi = \Psi$, where Φ and Ψ are words in the joint alphabet $\mathscr{A} \cup \mathscr{V}$ of letters and variables, its solution is a substitution $\sigma : \mathscr{V} \mapsto \mathscr{A}^*$ *s.t.* $\Phi\sigma$ is textually equal to $\Psi\sigma$. The algorithms provide transformation steps, which, applied iteratively to a given equation, generate its solution set. Thus, a (partial) solution tree of the given equation is produced. Some paths of the solution tree may be infinite, and are to be either pruned or represented as loops. For example, given

*The reported study was partially supported by Russian Academy of Sciences, research project No. AAAA-A16-116021760039-0.

[1]We use the assumption that only the elements c belonging to the function domain are considered, which is expressed by the condition $\exists f(c)$.


F(v_0)
$v_0 \mapsto \mathbf{A}\, v_1$
$v_0 \mapsto \varepsilon$
$v_0 \mapsto \mathbf{B}\, v_1$
G(v_1)
T
H(v_1)
$v_1 \mapsto \mathbf{A}\, v_2$
$v_1 \mapsto \varepsilon$
$v_1 \mapsto \mathbf{B}\, v_2$
F(v_2)
T
$v_1 \mapsto \varepsilon$
T
F(v_2)
x **A** y = y **A** x
x↦y x
x↦y
y↦x y
x **A** y = **A** y x
T
A x y = y **A** x
x↦**A** x
x↦ε
x **A** y = y **A** x
T
y↦ε
y↦**A** y
T
x **A** y = y **A** x

(a) Program transformation

(b) Solving a word equation

Figure 1: Unfold/fold method for solving the word equations and program transformation.

a path in the solution tree and a node along this path, Matyiasevich's algorithm constructs an arc leading to this node from its descendant when the two equations marking these nodes coincide modulo the variable names. Figure 1 shows a solution graph for a simple word equation constructed by Matyiasevich's algorithm and a graph of states and the state transitions of a functional program constructed by a basic unfold/fold algorithm. Henceforth, we refer to such graphs as (partial) process trees [32, 36]. One may find that the two graphs coincide modulo the node names. This paper focuses on the mentioned similarity of the methods and aims at adjusting the unfold/fold program transformation algorithm to solving the word equations.

Our contributions are the following. We study three interpreters, specialization of which using a general-purpose tool constructs a residual program that presents the complete solutions set of a given word equation. We prove that an analogue of Jones-optimality holds for the interpreters considered [3, 16], which guarantees that the satisfiability of equation systems analysed can be easily restored by checking a simple syntactic property of the residual programs. Surprisingly, the naive unfolding plus some basic optimisations generate a solution algorithm, for example, for the set of the one-variable word equations, and the algorithm differs from the well-known one given by Hmelevskij [14]. We also reveal several other sets of equations sharing variables in right- and left-hand sides, for which the specialization is shown to be sound. To the best our knowledge, these equation sets were uncovered by the published works on the string constraint solvers applied to the unbounded-length case. Finally, we show the application results of the presented approach to the benchmark equation sets for the solver `Woorpje` [8], and to a new benchmark of 50 equation systems, and compare the results with the results of the application of the SMT-solvers `Z3str3` and `CVC4` to these benchmarks. The systems were generated by the authors of the benchmarks randomly, and the complete solution sets are constructed by our algorithm for the most of the equation systems considered in the tests.

The remainder of this paper is structured as follows. In Sec. 2, we introduce the presentation syntax. We describe the interpreters used in Sec. 3, and the verification task in Sec. 4. The general unfold/fold scheme is given in Sec. 5. In Sec. 6, we discuss the optimality of the specialization and present results of the verification, in particular, for a number of sets of the word equations we show that every equation in any of the sets can be solved via the verification method. Sec. 7 considers related work, and Sec. 8 concludes the paper. The proofs of the main statements on properties of the presented algorithms are given in Appendix, as well as the source code of the interpreter models used in the specialization.

2 Preliminaries

We denote the set of the string variables with $\mathscr{V}$, the set of the letters with $\mathscr{A}$. A term is an element of $\mathscr{A} \cup \mathscr{V}$. A word equation is an equation $\Phi = \Psi$, where $\Phi, \Psi \in \{\mathscr{A} \cup \mathscr{V}\}^*$. A word equation is said to be *reduced* iff its sides neither start nor end with the same terms [7].

We write an application of substitution $\xi : \mathscr{V} \to \{\mathscr{A} \cup \mathscr{V}\}^*$ to a word Φ as $\Phi\xi$. A solution of the equation $\Phi = \Psi$ is a substitution $\xi : \mathscr{V} \to \mathscr{A}^*$ *s.t.* $\Phi\xi$ coincides with $\Psi\xi$ textually. Given an equation $E : \Phi = \Psi$ and substitution ξ, $E\xi$ is $\Phi\xi = \Psi\xi$ (by default, in the reduced form). We denote the number of occurrences of the term t in a word Φ with $|\Phi|_t$. The word equation length is $|\Phi| + |\Psi|$.

2.1 Presentation Language

In this paper we use the following pseudocode for functional programs manipulating the strings and based on the pattern matching. The programs are written as lists of term rewriting rules. The rules in the definitions are applied using the top-down matching order. The syntax is given in Figure 2. Here ε is the empty word, $++$ stands for the associative concatenation sign (both may be omitted). The set of the constants used as the letters[2] is Σ, elements of which are given in bold.

$$\begin{array}{rcllllll}
\text{Rule} & ::= & \multicolumn{6}{l}{\text{FName}\,(\text{Pattern}, \ldots, \text{Pattern}) \ = \ \text{Exp}} \\
\text{Pattern} & ::= & \varepsilon & \mid \text{Variable} & \mid \text{Letter} & \mid (\text{Pattern}) & \mid \text{Pattern} ++ \text{Pattern} & \\
\text{Exp} & ::= & \varepsilon & \mid \text{Variable} & \mid \text{Letter} & \mid (\text{Exp}) & \mid \text{FName}\,(\text{Exp}, \ldots, \text{Exp}) & \mid \text{Exp} ++ \text{Exp} \\
\text{Variable} & ::= & \mathtt{x}\text{Name} & \mid \mathtt{s}\text{Name} & & & &
\end{array}$$

Figure 2: The syntax of the program pseudocode.

An *object* expression is either a string in Σ^*, catenation of two object expressions, or (Expr), where Expr is an object expression. The variables with the first letter `x` range over object expressions; the variables with the first letter `s` range over symbols in Σ. We denote the set of the variables occurring in the expression Exp with $\mathscr{V}ar(\text{Exp})$. Given a program, the function `Go` serves as its entry function. The programming language uses the call-by-value evaluation strategy.

3 Word Equations Interpreters

In this section we introduce a class of simple interpreters transforming the word equations. Actually the interpreters generate finite paths in an equation solution tree in a way being similar to Matiyasevich's approach [25]. The nodes in the tree are labeled with configurations, which are equation lists[3]. Given a list of the word equations $\langle \Phi_1 = \Psi_1, \ldots, \Phi_n = \Psi_n \rangle$, its solution tree is a (possibly infinite) directed graph representing a non-deterministic unfolding process using the rules listed in Figure 3 (a). We see the narrowing rules in the first column. Given a narrowing rule, the second column provides the constraint imposed on the first equation in the list required for applying the rule. Following the classical approach [7, 14, 25], no fresh variables and constants are introduced in the narrowing rules.

We use a slightly modified version of Matiyasevich's algorithm. The Nielsen transformation [9], which is the base of the algorithm, states that given $\mathtt{x}\Phi_1 = \mathtt{y}\Psi_1$ either the length of the value of the

[2] This set is wider than the set $\mathscr{A}$, Sec. 2, because it contains also the letters used in the inner encoding of the equations.

[3] In order to emphasize that the the graph depends on the order of the equations in the equation system, we use "the list of the equations" instead of "the system ...".

$\mathtt{x} \mapsto \varepsilon$	$\mathtt{x}\Phi_1 = \Psi_1$ or $\Phi_1 = \mathtt{x}\Psi_1$
$\mathtt{x} \mapsto t\,\mathtt{x}$ $(t \in \mathscr{A})$	$\mathtt{x}\Phi_1 = t\Psi_1$ or $t\Phi_1 = \mathtt{x}\Psi_1$
$\mathtt{x} \mapsto \mathtt{y}\,\mathtt{x}$	$\mathtt{x}\Phi_1 = \mathtt{y}\Psi_1$ or $\mathtt{y}\Phi_1 = \mathtt{x}\Psi_1$

(a) The rewriting rules and equations required to apply them.

$$\begin{tikzcd}
 & & \boxed{\mathbf{A}\,\mathtt{x}\,\mathtt{y} = \mathtt{x}\,\mathtt{y}\,\mathbf{A}} \arrow[dl, "\mathtt{x}\mapsto\varepsilon"'] \arrow[loop right, "\mathtt{x}\mapsto\mathbf{A}\,\mathtt{x}"] \\
 & \boxed{\mathbf{A}\,\mathtt{y} = \mathtt{y}\,\mathbf{A}} \arrow[dl, "\mathtt{y}\mapsto\varepsilon"'] \arrow[loop right, "\mathtt{y}\mapsto\mathbf{A}\,\mathtt{y}"] & \\
\mathbf{T} & &
\end{tikzcd}$$

(b) The solution graph of $\mathbf{A}\,\mathtt{x}\,\mathtt{y} = \mathtt{x}\,\mathtt{y}\,\mathbf{A}$.

Figure 3: The rewriting rules and an example of a solution graph.

variable $\mathtt{x}$ is greater than the length of the value of $\mathtt{y}$, or vice versa, or their lengths are equal (and, consequently, $\mathtt{x} = \mathtt{y}$). Figure 3 (a) does not show the third case: it is a composition of $\mathtt{x} \mapsto \mathtt{y}\,\mathtt{x}$ and $\mathtt{x} \mapsto \varepsilon$, and we allow the rule $\mathtt{x} \mapsto \varepsilon$ to be applied to any equation whose left- or right-hand side starts with $\mathtt{x}$. This modification guarantees that any solution of the equation $\Phi_1 = \Psi_1$ can be generated[4] as a composition of the substitutions given in Figure 3 (a). The following example shows that for the classical form of the algorithm, this statement does not hold.

Example 1. *Let* $\mathtt{x}\sigma = \mathbf{A}$, $\mathtt{y}\sigma = \varepsilon$, *and the equation* $\mathtt{x}\,\mathtt{y} = \mathtt{y}\,\mathtt{x}$ *be considered. If we use the narrowings* $\mathtt{x} \mapsto \mathtt{y}\,\mathtt{x} \vee \mathtt{y} \mapsto \mathtt{x}\,\mathtt{y} \vee \mathtt{x} \mapsto \mathtt{y}$ *provided by the classic form of the Nielsen transformation then the solution generated by* σ *cannot be obtained by any finite number of such substitutions.*

We assume that an infinite path in a solution tree is to be folded iff it contains nodes N_1 and N_2 *s.t.* N_1 is an ancestor of N_2 and their configurations textually coincide. Then the subtree with the root N_2 repeats the subtree with the root N_1, and the infinite path can be represented as a cycle. Henceforth we use the notions of the solution tree and of the solution graph almost interchangeably. An example of a graph representing all solutions of the equation $\mathbf{A}\,\mathtt{x}\,\mathtt{y} = \mathtt{x}\,\mathtt{y}\,\mathbf{A}$ is given in Figure 3 (b). The cyclic arcs in the graph show the folding operation. For the sake of brevity, the nodes along the folded paths are not shown.

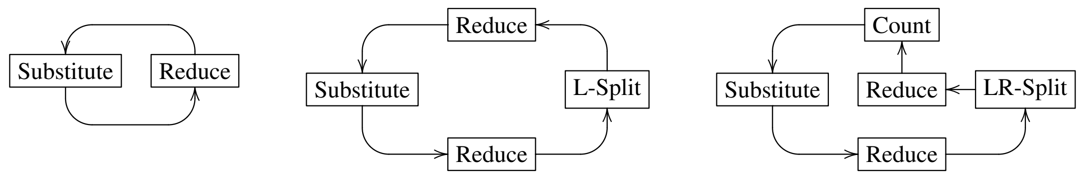
$$\begin{array}{rcccccc}
\text{Program} & ::= & \varepsilon & \mid & \text{Subst; Program} & & \\
\text{Subst} & ::= & \text{Var} \mapsto \varepsilon & \mid & \text{Var} \mapsto \text{Letter Var} & \mid & \text{Var} \mapsto \text{Var' Var}
\end{array}$$

Figure 4: The syntax of the simple logical programs.

An interpreter $\mathrm{WI}(P, \langle E_1 \ldots, E_n\rangle)$ takes a list of the word equations $E_1, \ldots, E_n$ and a linear logical program P being a list of the variable substitutions that have to be successively applied to the equations. The syntax of such programs is given[5] in Figure 4. If the substitutions given in P transform all the equations in the list to the tautologies, then $\mathrm{WI}(P, \langle E_1 \ldots, E_n\rangle)$ returns the value $\mathbf{T}$. If a substitution is invalid (cannot be applied soundly) *w.r.t.* the current equations list or the list of substitutions is empty whereas the equations are not tautologies then $\mathrm{WI}(P, \langle E_1 \ldots, E_n\rangle)$ results in the value $\mathbf{F}$. Whatever the input list of equations is, the interpreter does at most m steps, where m is the number of substitutions in P, and always returns either the value $\mathbf{T}$ or the value $\mathbf{F}$.

[4]The idea behind the algorithm originated by Matiyasevich is aimed at deciding the solvability of an equation, rather than at constructing the whole set of the solutions.

[5]In the paper, we use some syntactic sugar to simplify the programs encoding (*e.g.* we replace the string `'EMPTY'` with ε, the string `'TO'` with $\mapsto$, and use the semicolon instead of parentheses to separate the substitutions in the program).



Interpreter WIBase
Scheme $\text{AlgWE}_{\text{Base}}$

Interpreter WISplit
Scheme $\text{AlgWE}_{\text{Split}}$

Interpreter WICount
Scheme $\text{AlgWE}_{\text{Count}}$

Figure 5: Schemes for solving the word equations.

We use the notion of a parameter value (or a dynamic variable) for a datum which is already given, but it is unknown to us; while a variable value is undefined and is to be assigned. Thus, the variables are matched against data, while the parameters are to be narrowed. The parameters are used in this paper in order to represent possible paths in the solution tree. Thus, if the program P in $\text{WI}(P, \langle E_1 \ldots, E_n \rangle)$ is replaced by a parameter, then the unfolding of this call has to construct all the possible programs that can be applied to the list of the equations. We denote the parameters with the letters v (maybe subscripted). We say an expression is *ground* if it does not contain function calls, but may contain parameter occurrences. The set of the parameters is denoted with $\mathscr{Par}$, and the set of the ground expressions with $\mathscr{Grd}$.

Generally we distinguish the configurations in a *process* tree of $\text{WI}(v, \mathscr{Eqs})$ (where v is a parameter) from the configurations in a *solution* tree of $\mathscr{Eqs}$ generated by a non-deterministic application of the rules given in Figure 3. However, the structure of the interpreters given below allows us to refer to the solution tree of an equation instead of a more complex process tree of the program specialized by the equation (see Lemma 1).

The general structure of the three interpreters studied in this paper is given in Figure 5. The transformations shown in this figure are also used for constructing the corresponding solution graphs. According to Figure 5 we say that the interpreter WIBase models paths in a solution tree constructed by means of the algorithm $\text{AlgWE}_{\text{Base}}$; and so on. See Figure 6 for examples of the corresponding solution graphs.

3.1 Basic Interpreter WIBase

The basic interpreter $\text{WI}(P, E)$ manipulates with a single equation E and accepts as the first input a list of the variable substitutions presented in Figure 3 (a). When a substitution σ is applied, the interpreter WIBase immediately reduces $E\sigma$. In fact, this basic interpreter models the classic algorithm for solving the word equations suggested by Matiyasevich and used in the works [18, 20].

3.2 Splitting Interpreter WISplit

The interpreter $\text{WISplit}(P, \langle E_1, \ldots, E_n \rangle)$ manipulates with the lists of equations rather than with a single equation. Thus, the variable substitutions and reductions are applied to every equation in the list. If a reduction results in an equation $t_1\,\Phi_i = t_2\,\Psi_i$, where $t_1 \in \mathscr{A}$, $t_2 \in \mathscr{A}$, $t_1 \neq t_2$, the equation is immediately replaced with the trivial contradictory equation $t_1 = t_2$, and all the other equations in the list are removed. We assume that the operation looking for trivial contradictions is also a part of the reduction algorithm.

A natural way to simplify an equation manually is an attempt to split it using the length argument [7]. *E.g.* given equation $\mathtt{x}\,\mathbf{A}\,\mathtt{x}\,\mathbf{B}\,\mathtt{x} = \mathbf{B}\,\mathtt{x}\,\mathtt{x}\,\mathtt{x}\,\mathbf{A}$ we can split it into the list of $E_1 : \mathtt{x}\,\mathbf{A} = \mathbf{B}\,\mathtt{x}$ and $E_2 : \mathtt{x}\,\mathbf{B}\,\mathtt{x} = \mathtt{x}\,\mathtt{x}\,\mathbf{A}$, where E_2 is reduced to E_1.

Let us describe the mentioned method more formally. Given Φ and Ψ in $\{\mathscr{A} \cup \mathscr{V}\}^+$, *s.t.* $|\Phi| = |\Psi|$ and for every $\mathtt{x} \in \mathscr{V}$ the equality $|\Phi|_{\mathtt{x}} = |\Psi|_{\mathtt{x}}$ holds, we say that the words Φ and Ψ are *variable-permutated* (briefly, var-permutated). The following proposition is trivial.

Proposition 1. *Let an equation be of the form* $\mathrm{Pr}_1\,\mathrm{S}_1 = \mathrm{Pr}_2\,\mathrm{S}_2$ *where the prefixes* Pr_1 *and* Pr_2 *are non-empty and var-permutated. Then the equation is equivalent to the system* $\mathrm{Pr}_1 = \mathrm{Pr}_2 \;\&\; \mathrm{S}_1 = \mathrm{S}_2$.

The same proposition holds for var-permutated suffixes. If it is applied to the prefixes of an equation we say that the equation is left-split; if Proposition 1 is applied to the suffixes we say the equation is right-split.

The interpreter WISplit uses Proposition 1 as follows. Given a call $\mathrm{WISplit}(\langle\sigma_1,\dots\rangle,\langle E_1\dots,E_n\rangle)$, the interpreter first reduces all the equations in the list $\langle E_1\sigma_1,\dots,E_n\sigma_1\rangle$. For every resulting equation E'_i, WISplit tries then to find the shortest non-empty var-permutated prefixes $\mathrm{Pr}_{1,i,1}$ and $\mathrm{Pr}_{2,i,1}$ of its left- and right-hand sides. In the case of success, WIBase splits the equation E'_i into the two equations $\mathrm{Pr}_{1,i,1} = \mathrm{Pr}_{2,i,1}$ and $\mathrm{S}_{1,i,1} = \mathrm{S}_{2,i,1}$, reduces the equation $\mathrm{S}_{1,i,1} = \mathrm{S}_{2,i,1}$, and tries to left-split it, *etc.* until $\mathrm{S}_{1,i,j}$ and $\mathrm{S}_{2,i,j}$ have no non-empty var-permutated prefixes. The generated equations resulting in k successful left-split operations are placed instead of the initial equation E'_i in the list. The order of the new equations is as follows: $\mathrm{S}_{1,i,k} = \mathrm{S}_{2,i,k}, \mathrm{Pr}_{1,i,1} = \mathrm{Pr}_{2,i,1}, \dots, \mathrm{Pr}_{1,i,k} = \mathrm{Pr}_{2,i,k}$. The interpreter WISplit does not reduce the resulting equations $\mathrm{Pr}_{1,i,k} = \mathrm{Pr}_{2,i,k}$, because their construction guarantees that they are already reduced.

3.3 Counting Interpreter WICount

The third variant of our interpreter is based on the following simple observation.

Proposition 2. *Given an equation* $\Phi = \Psi$*, let for every* $\mathtt{x} \in \mathscr{V}$ $|\Phi|_{\mathtt{x}} \geq |\Psi|_{\mathtt{x}}$*, and* $\sum\limits_{t_i \in \mathscr{A}} |\Phi|_{t_i} > \sum\limits_{t_i \in \mathscr{A}} |\Psi|_{t_i}$. *Then the equation* $\Phi = \Psi$ *has no solution.*

After reducing the equations, the interpreter WICount tries to construct their left-splits (as WISplit does), and then to construct the right-splits of their suffixes resulted from the left-splits. Finally, WICount checks the resulting equations with non-var-permutated sides according to Proposition 2.

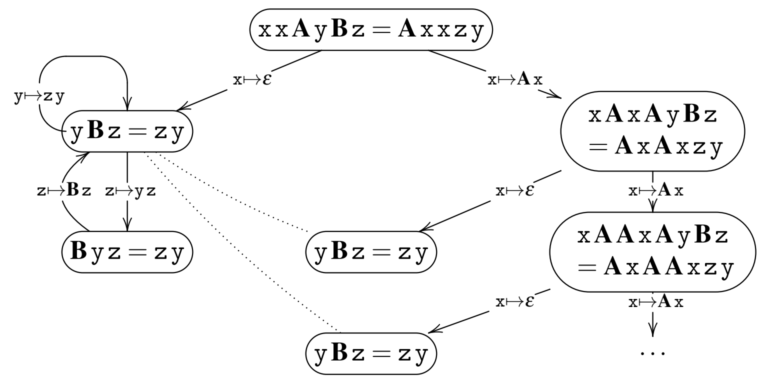
Figure 6 demonstrates the difference between the algorithms used in the presented interpreters. The dotted edges in the graph for the algorithm $\mathrm{AlgWE}_{\mathrm{Base}}$ show the equality of the configurations; but the configurations are not folded there, because they are not along the same path. The edges given in the two parallel lines show splits. The graph for $\mathrm{AlgWE}_{\mathrm{Base}}$ is infinite, whereas the other two graphs show that the equation has no solution.

4 Verification Task

Below we use the underlining sign to show encoded structures of the data and program to be specialized. Given a program transformation tool `Spec`, a list of the word equations $\mathscr{E}qs$ and an encoded sources $\underline{\mathrm{WI}}$ of an interpreter WI, the specialization task is as follows.

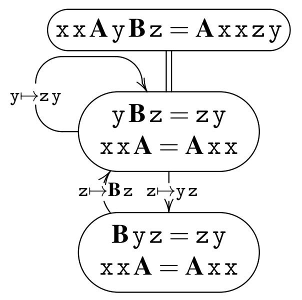
$$\mathfrak{M}(\mathrm{WI},\mathscr{E}qs) = \mathtt{Spec}(\underline{\mathrm{WI}}, \underline{\mathtt{Go}}(v, \underline{\mathscr{E}qs})),$$

where `Go` is the name of the entry function of WI. Here v ranges over the set of the encoded programs that can be interpreted by WI, namely all the possible encoded sequences of the narrowing rules shown in Figure 3 (a).

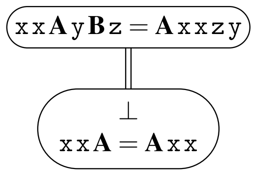


xx**A**y**B**z = **A**xxzy
x↦ε
x↦**A**x
y↦zy
y**B**z = zy
x**A**x**A**y**B**z = **A**x**A**xzy
z↦**B**z
z↦yz
x↦ε
x↦**A**x
Byz = zy
y**B**z = zy
x**AA**x**A**y**B**z = **A**x**AA**xzy
x↦ε
x↦**A**x
y**B**z = zy
…

(a) Algorithm $\text{AlgWE}_{\text{Base}}$



(b) Algorithm $\text{AlgWE}_{\text{Split}}$



(c) Algorithm $\text{AlgWE}_{\text{Count}}$

Figure 6: Solution graphs for $\mathtt{xx}\mathbf{A}\mathtt{y}\mathbf{B}\mathtt{z} = \mathbf{A}\mathtt{xxzy}$.

The result of this specialization is a program with the input value to be assigned to v. We say that the specialization succeeds if the resulting process tree generated by `Spec` contains a leaf labelled by the configuration **T** iff the equation system $\mathscr{E}qs$ has a solution. In that case the specialization tool `Spec` verifies the existence of a sequence of substitutions that generates a solution of the system $\mathscr{E}qs$ given to the interpreter WI. Every solution of the equation corresponds to a non-empty (possibly infinite) set of paths rooted in the initial configuration of its solution tree and ending at a leaf labelled by **T**. Thus, the verification task is successfully solved if the solution graph is finite. The finiteness is guaranteed with the following property of the solution tree: there are at least two equal configurations along every infinite path in the tree.

5 Unfold/Fold Program Transformation Method

The specialization tool `Spec` used in the verification scheme is based on the elementary unfold/fold technique widely used, *e.g.* in deforestation, supercompilation, partial evaluation, partial deduction, and so on [5, 16, 32, 35]. The technique exploits the sub-algorithms presented informally in this Section and more formally in the paper [22]. The unfold/fold algorithm assumes that every node N in the process tree of the program is labelled with a configuration, which represents the call-stack.

Definition 1. *Given a program rule* $R : \mathtt{F}(\mathrm{P}_1, \ldots, \mathrm{P}_\mathrm{n}) = \mathrm{T}$ *and an expression* $C = \mathtt{F}(\mathrm{Expr}_1, \ldots, \mathrm{Expr}_\mathrm{n})$, *where* $\mathrm{Expr_i}$ *are ground*[6], *we say that the substitution* $\xi : \mathscr{P}\!ar \mapsto \mathscr{G}rd$ *is* a narrowing *generated by the rule* R *iff* $\exists \sigma : \mathscr{V}ar(\langle \mathrm{P}_1, \ldots, \mathrm{P_n} \rangle) \mapsto \mathscr{G}rd$ *s.t.* $\forall i\, (\mathrm{P_i}\sigma = \mathrm{Expr_i}\xi)$. *We say that the set of the narrowings* $\{\xi_i\}$ *is* exhaustive *w.r.t.* R *if for every substitution* $\tau : \mathscr{P}\!ar \mapsto \Sigma^*$ *s.t. the expression* $C\tau$ *matches against the left-hand side of the rule* R *(i.e.* $\exists \sigma' : \mathscr{V}ar(\langle \mathrm{P}_1, \ldots, \mathrm{P_n} \rangle) \mapsto \Sigma^*$ *s.t.* $\forall i\, (\mathrm{Expr_i}\tau = \mathrm{P_i}\sigma')$*), there exists a narrowing* ξ_i *s.t.* τ *is an instance of* ξ_i.

Now we are ready to describe the unfold/fold algorithm. Every node in the process tree is marked either as open (by default), or as closed with some node N'. The three steps listed below are applied to the process tree until all the nodes in the tree are closed.

- **Unfolding step**. Given an open node N labelled by a parameterized configuration C, consider the call $\mathtt{F}(\mathrm{Expr}_1, \ldots, \mathrm{Expr_n})$ at the call-stack top. For every rule $R_i : \mathtt{F}(\mathrm{P}_{i,1}, \ldots, \mathrm{P}_{i,n}) = \mathrm{T_i}$ in the definition of $\mathtt{F}$ (where $\mathrm{P_{i,j}}$ are patterns), construct a set of pairs $\langle \sigma_{i,k}, \xi_{i,k} \rangle$ *s.t.* $\forall j\, (\mathrm{P_{i,j}}\sigma_{\mathrm{i,j}} = \mathrm{Expr_j}\xi_{\mathrm{i,j}})$, and the set $\{\xi_{i,k}\}$ of the narrowings is exhaustive *w.r.t.* R_i. For every such narrowing $\xi_{i,k}$ generate a child node $\mathsf{N}_{i,k}$ labelled by the configuration $C_{i,k}$, where $C_{i,k}$ is $C\xi_{i,k}$ in which the stack top call $\mathtt{F}(\mathrm{Expr}_1, \ldots, \mathrm{Expr_n})\sigma_{\mathrm{i,k}}$ is replaced[7] by $\mathrm{T_i}\sigma_{i,k}$. Mark all the newly generated nodes as open.
- **Folding step**. Given a node N labelled by a configuration C, if some its ancestor node N' is labelled by C (up to a parameter renaming), mark N as closed with N', and stop unfolding paths originating from N.
- **Close**. Mark an open node N as closed with N if either N is labelled with a ground expression, or all the successors of N are closed.

To guarantee that the matching algorithm in the unfolding step can always produce a finite set of the narrowings (and there are no multiple parameter occurrences in a configuration, which would require a more complex matching algorithm), we use the following syntactic property of the interpreters considered.

Property 1. *Given the interpreters* WIBase*,* WISplit*, and* WICount*, any program rule defining the function* `Main` *in the interpreters can be only as follows:*

- $\mathtt{Main}(\mathrm{P}, \mathrm{T}_1) = \mathrm{T}_2$*, where* T_2 *is an object expression;*
- $\mathtt{Main}(\mathrm{P}, \mathrm{T}_1) = \mathtt{Main}(\mathtt{x_{rul}}, \mathrm{T}_2)$*, where* $(\mathscr{V}ar(\mathrm{T}_2) \setminus \mathscr{V}ar(\mathrm{T}_1)) \cap \mathscr{V}ar(\mathrm{P}) = \varnothing$*, and the only expression-type pattern variable belonging to* $\mathscr{V}ar(\mathrm{P})$ *(namely* $\mathtt{x_{rul}}$*) does not occur in* T_2.

We recall that the verification task is $\mathtt{Spec}(\underline{\mathrm{WI}}, \underline{\mathtt{Go}(}v, \underline{\mathscr{E}qs)})$, and the rules of the function `Go` for all the three interpreters are $\mathtt{Go}(\mathtt{x}, \mathtt{x}') = \mathtt{Main}(\mathtt{x}, \mathtt{Sim}(\text{Other arguments}))$, where the other arguments have no occurrences of `x`. This fact together with Property 1 imply the following feature.

Property 2. *Let us consider the process tree of* $\mathfrak{M}(\mathrm{WI}, \mathscr{E}qs)$*, where* $\mathrm{WI} \in \{\mathrm{WIBase}, \mathrm{WISplit}, \mathrm{WICount}\}$.

- *Given an arbitrary configuration C labelling a node in the process tree, the only parameterized call in the call-stack C (if any) is of the form* $\mathtt{Main}(v_i, \text{Other arguments})$*, where no parameter occurs*[8] *in the other arguments.*
- *The patterns to be matched against the parameterized data never have more than one occurrence of an expression-type pattern variable.*

[6]This property is guaranteed by the call-by-value semantics.

[7]In the case of the verification task considered, the narrowings $\xi_{i,k}$ are not substituted in the other calls in the stack because Corollary 2 holds, so we can simply state that $C_{i,k}$ is the configuration C with the stack top call replaced by $\mathrm{T_i}\sigma_{i,k}$.

[8]A program rule $\mathtt{Main}(\mathrm{P}, \mathrm{T}_1) = \mathrm{T}_2$ can have letter-type pattern variables shared by P and T_1 that occur in T_2, but the value matched against T_1 is always an object expression, hence these variables are substituted by letters in any pattern matching.

Property 2 guarantees that the narrowings imposed on the parameters in a configuration that are constructed by the unfolding step are always applied only to the function call `Main` at the call-stack top. Moreover, Property 2 together with Property 1 show that in the case of the verification task considered we can construct the exhaustive narrowing set consisting exactly of one narrowing. Depending on the form of the rule defining the function `Main`, the narrowing would be either $v_i \mapsto \varepsilon$ of $v_i \mapsto P' {+}{+} v_i'$, where P' is an object expression. Provided this feature[9], the matching process is always finite [27].

We call a node transient [41], if the unfolding step generates no narrowing on the parameters in its configuration (in particular, if all the $\mathrm{Expr_i}$ in $\mathtt{F}(\mathrm{Expr}_1, \dots, \mathrm{Expr_n})$ are object expressions). A transient node has a unique child in the process tree.

6 Results of Specialization

This section presents several sets of word equation systems, which have been solved by the specialization task $\mathfrak{M}(\mathrm{WI}, \mathscr{E}qs)$, where WI is either WIBase, WISplit, or WICount.

Given an equation list $\mathscr{E}qs$ and a word equations interpreter WI, the result of the specialization $\mathfrak{M}(\mathrm{WI}, \mathscr{E}qs)$ is a process tree modelling the solution tree of $\mathscr{E}qs$. If the process tree is infinite, then the specialization process does not terminate, unless on every infinite path in the process tree two equal configurations exist. Thus, the specialization succeeds iff the relation of the textual equality is a well-quasi order on the configuration sequence along every infinite path in the process tree.

6.1 Optimality of Specialization

In general, a process tree may require to construct a folding arc connecting transient configurations. That makes the reasoning on the process trees in the terms of the solution graphs difficult. All the arcs in a solution graph are marked with narrowings, but the process tree after such a folding will contain some additional arcs, which do not impose a narrowing. The structure of the interpreters WIBase, WISplit, WICount guarantees that all the non-transient nodes are labelled by configurations having a call of the function `Main` at the call-stack top (see Corollary 2). Thus, we introduce a notion of the optimal specialization for the verification task given in Sec. 4.

Definition 2. *A result of the specialization* $\mathfrak{M}(\mathrm{WI}, \mathscr{E}qs)$ *is said to be optimal iff all the arcs folding computation paths in the process tree connect the nodes labelled by* $\mathtt{Main}(v_i, \mathscr{E}qs_i)$*, where* $\mathscr{E}qs_i$ *is an object expression.*

If the optimality is guaranteed, then all the intermediate steps of the interpretation, including splitting, reducing, *etc.* correspond to the nodes having exactly one ingoing arc in the folded process graph. We can therefore reason on the process trees using the solution graphs of the equations. Moreover, the optimality guarantees that the residual programs generated by a specialization tool contain no part of the interpreters' source code. Thus, such an optimality can be considered as an analogue of the Jones-optimality [3, 16] for the given verification task. If we consider the input sequence of narrowings given to an interpreter WI as a linear program, we can also use the following reasoning [21]. Unlike the classical Futamura first projection [12], which corresponds to the specialization task $\mathtt{Spec}(\underline{\mathrm{WI}}, \underline{\mathtt{Go}(\langle\sigma_i\rangle}, \underline{v}))$, where

[9]Another feature important for the unfolding step is that for all function rules except the last one and for all i, j S_i is never a prefix of S_j, hence the narrowings imposed on the parameter v_i by the corresponding matchings are always disjoint (provided the equations in the list given to the function `Main` are reduced). The exception is the last rule of `Main`, which accepts an arbitrary input, serving as the `default` or `otherwise` branch. The right-hand side of the last rule is an object expression, hence there is no need to transfer any negative data on the parameter to the successor configurations.

the input data (the equation list) is replaced with a parameter, while the program given to the interpreter is fixed, we parameterize the program. We say that the result is Jones-optimal if the residual program contains no part of the interpreter WI.

Lemma 1. *For every word equation* $\Phi = \Psi$*, the result of the specialization task* $\mathfrak{M}(\mathrm{WI}, \Phi = \Psi)$*, where* WI *is either* WIBase*,* WISplit*, or* WICount*, is optimal.*

The proof is given in Appendix (Proof 8.1).

When the optimality holds, we say the verification by specialization of the interpreter WI *w.r.t.* a given equation set succeeds iff for every equation E in this set the solution graph constructed by the corresponding equation solving algorithm is finite.

6.2 Specialization of Interpreter WIBase

Definition 3. *A word equation* $\Phi = \Psi$ *is said to be* quadratic *iff for all* $\mathtt{x} \in \mathscr{V}$*,* $|\Phi|_{\mathtt{x}} + |\Psi|_{\mathtt{x}} \leq 2$.

The optimality lemma implies the following proposition: every solution graph for a quadratic equation constructed by the algorithm $\mathrm{AlgWE}_{\mathrm{Base}}$ is finite. This fact is well-known due to the work of Matiyasevich [17, 18, 20, 25]. Thus, specialization of WIBase *w.r.t.* quadratic equations provides a basic test on the optimality of the program model.

Proposition 3. *For every quadratic word equation* $\Phi = \Psi$*, the specialization task* $\mathfrak{M}(\mathrm{WI}, \Phi = \Psi)$ *succeeds.*

6.3 Specialization of Interpreter WISplit

The interpreter WISplit was introduced as an optimized version of WIBase, but the experiments have shown that the specialization of WISplit succeeds in significantly more cases. One possibly interesting class of the word equations solvable with the help of WISplit is the equations of a special kind with the solutions belonging to a regular language.

Definition 4. *Let* $\xi(\Phi)$ *be a transformation mapping any* $\mathbf{A} \in \mathscr{A}$ *explicitly occurring in* Φ *to* ε*. We say an equation* $\Phi = \Psi$ *is* strictly regular-ordered with repetitions *iff* $\xi(\Phi)$ *is textually equal to* $\xi(\Psi)$.

Thus, if the equation $\Phi = \Psi$ is strictly regular-ordered with repetitions, then $\forall \mathtt{x} \in \mathscr{V}\ (|\Phi|_{\mathtt{x}} = |\Psi|_{\mathtt{x}})$ and the variable occurrences are ordered in Φ and Ψ in the same way. The set of strictly regular-ordered equations with repetitions generalizes the set of regular ordered equations in which every variable occurs twice [9].

Example 2. *The solution sets of the three equations* $\mathbf{A}\,\mathtt{x} = \mathtt{x}\,\mathbf{A}$*,* $\mathbf{A}\,\mathbf{A}\,\mathtt{x} = \mathtt{x}\,\mathbf{A}\,\mathbf{A}$*,* $\mathbf{A}\,\mathtt{x}\,\mathtt{x} = \mathtt{x}\,\mathtt{x}\,\mathbf{A}$ *are all equal, namely the sets are* $\mathbf{A}^*$*. The first two equations are quadratic; the third is strictly regular-ordered with repetitions, but is not quadratic. Its solution graph constructed with the unfolding procedure* $\mathrm{AlgWE}_{\mathrm{Base}}$ *is infinite.*

The following proposition is a corollary of the optimality lemma and Lemma 2 given in Appendix.

Proposition 4. *For a strictly regular-ordered equation with repetitions* $\Phi = \Psi$*, the verification task* $\mathfrak{M}(\mathrm{WISplit}, \langle \Phi = \Psi \rangle)$ *succeeds.*

6.4 Specialization of Interpreter WICount

The interpreter WICount uses additional simplifying operations as compared to the interpreter WISplit. The specialization of this interpreter succeeds additionally in solving one-variable equations. The optimality lemma and Lemma 3 given in Appendix imply the following proposition (which fails for WISplit).

Proposition 5. *For a one-variable word equation* $\Phi = \Psi$*, the verification task* $\mathfrak{M}(\text{WICount}, \langle \Phi = \Psi \rangle)$ *succeeds.*

In order to test the verification technique presented in this paper experimentally, we have generated a benchmark consisting of 50 equation systems: the tests 1–10 are the regular-ordered equations with repetitions; the tests 11–20 are similar to the regular-ordered equations with repetitions, but may contain additional variable occurrences or have some variable occurrences permutated; the tests 21–30 contain equations of the form $\mathtt{x}\Phi = \Psi\mathtt{x}$, where $\Phi = \Psi$ is the regular-ordered equation with repetitions and neither Φ nor Ψ contain $\mathtt{x}$; the tests 31–40 present systems of the regular-ordered equations with repetitions mixed with equations of the form $\Phi\Psi = \Psi\Phi$; the tests 41–50 are equations of no special form sharing several variables in right- and left-hand sides. The archive containing the equations is given on the web-page [29].

The supercompiler SCP4 [26] was mainly used as $\mathfrak{M}$ in the tests of the scheme $\mathfrak{M}(\text{WICount}, \mathscr{E}qs)$. The experimental supercompiler `MSCP-A` [30] was also used and showed the same solvability results on the tests above, but it spends much more time for producing the results as compared with `SCP4`. The comparative verification results between the approach presented in this paper and the external SMT-solvers `CVC4`, `Z3str3` are presented in Figure 7, the last row. The results show that the scheme $\mathfrak{M}(\text{WICount}, \mathscr{E}qs)$ is quite stable modulo small changes in the equations which are guaranteed to be solved by it.

Finally, we have tested the scheme $\mathfrak{M}(\text{WICount}, \mathscr{E}qs)$ on the equation set provided by the paper [8] as a benchmark for the string constraint solver `Woorpje`, namely Track 1 consisting of 200 equations guaranteed to have a solution; and Track 5 consisting of 200 equation systems. We have removed the length constraints from the Track 5 benchmark before the specialization starts. The results are quite successful, provided that we use the general-purpose specialization tool for the verification. First of all, the residual programs constructed by $\mathfrak{M}(\text{WICount}, \mathscr{E}qs)$ never contain functions returning **T** if the system $\mathscr{E}qs$ has been found unsatisfiable by the other solvers. Moreover, is the system $\mathscr{E}qs$ has solutions, then $\mathfrak{M}(\text{WICount}, \mathscr{E}qs)$ always generates programs containing functions with the output **T**, if terminates. That is a practical evidence that the optimality lemma holds. Second, the equations are successfully solved in 179 out of 200 cases. This result is comparable with the verification results done by `Z3str3` [23]; for 17 equations the specialization process does not terminate. In the remaining cases, the specialization process is theoretically terminating but takes too much time. The equations for which the specialization is the most time-consuming all are linear, *i.e.* every variable occurs in at most once per such an equation. The reader can find the residual programs encoding paths in the solution graphs for the equations generated in the experiments on the web-page [29].

Benchmark	Tests (total)	Unsuccessful tests		
		`CVC4`	`Z3str3`	WICount
Track 1 (Woorpje)	200	8	13	21
Track 5 (Woorpje)	200	4	14	19
New benchmark	50	21	28	10

Figure 7: The verification results obtained by `CVC4` [19], `Z3str3` [23], $\mathfrak{M}(\text{WICount}, \mathscr{E}qs)$.

7 Discussion

The discussed specialization tasks above have been solved by using the following two program specializers designed for the string manipulating functional language Refal [40], namely the model supercompiler `MSCP-A` [28] and the experimental supercompiler `SCP4` [26]. Some results of the experiments can be found on the web-page [30]. Albeit we used the supercompilers, the results do not depend on specific features of the supercompilation method [39] and can be reproduced with other specializers based on partial evaluation, partial deduction, and so on [16, 32].

Ou approach is able to solve regular-ordered equations with repetitions, which showed to be hard for the existing solvers, especially in the case when the equation does not have a solution. For example, neither `Z3str3` nor `CVC4` terminate on the equation $\mathbf{AB}\,\mathtt{x}\,\mathtt{x}\,\mathtt{y}\,\mathtt{y} = \mathtt{x}\,\mathtt{x}\,\mathtt{y}\,\mathtt{y}\,\mathbf{BA}$, which is solved by the WISplit specialization.

The domain of the described verification method is not exhausted by the sets of the equations considered above. One more interesting class of the equations with the variables shared by left- and right-hand sides (and non-regular solution sets) consists of the equations of the form $\Phi\,\mathtt{y} = \mathtt{y}\,\Psi$, where $\Phi = \mathtt{x}\,\Phi_1\,\mathtt{x}\ldots\mathtt{x}\,\Phi_n$, $\Psi = \Psi_1\,\mathtt{x}\ldots\mathtt{x}\,\Psi_n\,\mathtt{x}$, where $\Phi_i, \Psi_i \in \mathscr{A}^*$ and $\exists k \in \mathbb{N}\,\forall i, j\,|\Phi_i| = k\;\&\;|\Psi_i| = k$. Examining their solution graphs, we can prove that the specialization task $\mathfrak{M}(\text{WICount}, \mathscr{E}qs)$ successfully solves such equations. The experiments with the randomly chosen equations mentioned above promise to find other interesting classes of the word equations that can be solved by automated specialization tools.

7.1 Related Works

A number of quite efficient string constraint solving tools were designed, which are looking for the word equation solutions bounded by a given length, *e.g.* [4, 8][10]. A reasoning on the unbounded case can provide efficient methods of solution search if the equations considered satisfy some special properties. For example, a number of efficient solving algorithms have been designed for the set of straight-line word equations, *e.g.* [1, 6], whereas our specialization *w.r.t.* such equations is too time-consuming.

Several recent works exploit the unfold/fold technique with Nielsen's transformation for solving quadratic word equations, in the way originated by Matiyasevich in 1968 [25]. In the paper [20], non-deterministic counter systems for searching solutions of the unbounded-length word equations with regular constraints via Nielsen's transformation are introduced, and the completeness of the given algorithm has been shown for the set of the regular ordered equations. Maybe the regular ordered equations with repetitions, being split in the way shown in $\text{AlgWE}_{\text{Split}}$ algorithm (Sec. 3), can be solved by this method as well. In the paper [18], Nielsen's transformation is used for solving unbounded-length quadratic word equations. As it the original Matiyasevich work, the termination of the suggested algorithm does not terminate if the input equation contains more than two occurrences of some variable. Advanced SMT-solvers such as `CVC4` [19], `Z3str3` [23], or `S3P` given in [38] demonstrate a very good efficiency in many practical cases, however the paper [18] shows that their algorithms are not complete even *w.r.t.* the set of the quadratic equations, *e.g.* the equation $\mathtt{x}\,\mathtt{v}\,\mathtt{y} = \mathtt{y}\,\mathtt{w}\,\mathtt{x}$, whose solution set is quite complex; see Hmelevskij's work [14] for the first proof of this fact. The tests of our benchmark have shown that the recent version of `CVC4` solves equations of the given form, but fails to solve more complex quadratic equations, *e.g.* $\mathtt{x}_1\,\mathtt{x}_2\,\mathtt{x}_3\,\mathbf{ABABAB} = \mathbf{AAABBB}\,\mathtt{x}_2\,\mathtt{x}_3\,\mathtt{x}_1$. Based on the results of the verification we may conclude that the most troublesome cases for the SMT-solvers are the ones when the equation system has

[10]In fact, if the upper bound is assigned dynamically, such a tool can decide solvability of every word equation, because a minimal solution length is at most doubly exponential in the equation length [15].

no solution and that fact cannot be shown by reasoning on solution lengths. In that cases, the verification scheme $\mathfrak{M}(\text{WICount}, \mathscr{E}qs)$ has the best success rate as compared to `CVC4` and `Z3str3`.

8 Conclusion

We have shown that general-purpose specializers can be useful for solving some classes of the word equations. Instead of modifying the specialization tools, we modify the word equation interpreters specialized according with the verification scheme. This technique uses a modification of the classical first Futamura projection [12] and simplifies the work of interest. Starting from the simplest interpreter WIBase, every new refinement extends significantly the set of the equations solvable via the specialization method. The specialization-time overheads are high as compared with the direct work of the existing string constraint solvers, but the specialization method presented in this paper aims at supporting development of the solver prototypes with a minimal effort. Experiments with the prototypes provide a fruitful research material on the sets of the word equations over which the verification algorithm terminates. Moreover, we have shown that theoretical methods for solving the word equations can be useful in the automatic approach, hence these methods can solve the task of proving unsatisfiability of a word equation system, which, as our experiments show, is hard for some well-known state-of-art SMT-solvers.

Another interesting aspect of the presented verification experiments is the optimality. The nondeterministic algorithms for solving the word equations are well-designed in order to be used in the intermediate interpretation. This paper considers the optimality property in the case of the basic folding, however our experiments show that the constructed interpreters provide possibility for the optimal verification, if one uses a more complex path termination criterion based on the homeomorphic embedding relation [36]. Thereby advanced specialization tools can also be used for solving the word equations, and the additional strategies developed for program transformation may support more efficient algorithms as compared with the basic unfold/fold method.

References

[1] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Lukáš Holík, Ahmed Rezine & Philipp Rümmer (2017): *Flatten and Conquer: A Framework for Efficient Analysis of String Constraints*. *SIGPLAN Not.* 52(6), pp. 602–617, doi:10.1145/3140587.3062384.

[2] Steve Barker, Michael Leuschel & Mauricio Varea (2008): *Efficient and flexible access control via Jones-optimal logic program specialisation*. *High. Order Symb. Comput.* 21(1-2), pp. 5–35, doi:10.1007/s10990-008-9030-8.

[3] A. Ben-Amram & N. Jones (2000): *Computational complexity via programming languages: constant factors do matter*. *Acta Informatica* (37), pp. 83–120, doi:10.1007/s002360000038.

[4] Nikolaj Bjørner, Nikolai Tillmann & Andrei Voronkov (2009): *Path Feasibility Analysis for String-Manipulating Programs*. In Stefan Kowalewski & Anna Philippou, editors: *Tools and Algorithms for the Construction and Analysis of Systems*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 307–321, doi:10.1007/978-3-642-00768-2_27.

[5] R. M. Burstall & John Darlington (1977): *A Transformation System for Developing Recursive Programs*. *J. ACM* 24(1), pp. 44–67, doi:10.1145/321992.321996.

[6] Taolue Chen, Matthew Hague, Anthony W. Lin, Philipp Rümmer & Zhilin Wu (2019): *Decision Procedures for Path Feasibility of String-Manipulating Programs with Complex Operations* 3(POPL), p. 30. doi:10.1145/3290362.

[7] Christian Choffrut & Juhani Karhumäki (1997): *Combinatorics of Words*. doi:10.1007/978-3-642-59136-5_6.

[8] J. D. Day, Thorsten Ehlers, Mitja Kulczynski, Florin Manea, Dirk Nowotka & Danny Bogsted Poulsen (2019): *On Solving Word Equations Using SAT*. In: *Reachability Problems. RP 2019*, 11674, Lecture Notes in Computer Science, doi:10.1007/978-3-030-30806-3_8.

[9] Joel D. Day, Florin Manea & Dirk Nowotka (2017): *The Hardness of Solving Simple Word Equations*. In: *42nd International Symposium on Mathematical Foundations of Computer Science (MFCS 2017)*, *Leibniz International Proceedings in Informatics (LIPIcs)* 83, pp. 18:1–18:14, doi:10.4230/LIPIcs.MFCS.2017.18.

[10] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi & Maurizio Proietti (2018): *Solving Horn Clauses on Inductive Data Types Without Induction*. *Theory Pract. Log. Program.* 18(3-4), pp. 452–469, doi:10.1017/S1471068418000157. Available at `https://doi.org/10.1017/S1471068418000157`.

[11] Jesús J. Doménech, John P. Gallagher & Samir Genaim (2019): *Control-Flow Refinement by Partial Evaluation, and its Application to Termination and Cost Analysis*. *Theory Pract. Log. Program.* 19(5-6), pp. 990–1005, doi:10.1017/S1471068419000310.

[12] Yoshihiko Futamura (1999): *Partial Evaluation of Computation Process - An Approach to a Compiler-Compiler*. *Higher-Order and Symbolic Computation* 12, pp. 381–391, doi:10.1023/A:1010095604496.

[13] Geoff W. Hamilton (2015): *Verifying Temporal Properties of Reactive Systems by Transformation*. In: *Proceedings of the Third International Workshop on Verification and Program Transformation, VPT@ETAPS 2015, London, United Kingdom, 11th April 2015.*, pp. 33–49, doi:10.4204/EPTCS.199.3. Available at `https://doi.org/10.4204/EPTCS.199.3`.

[14] Ju. I. Hmelevskij (1971): *Equations in a free semigroup. (in Russian)*. *Trudy Mat. Inst. Steklov* 107, p. 286.

[15] Artur Jez (2016): *Recompression: A Simple and Powerful Technique for Word Equations*. *J. ACM* 63(1), doi:10.1145/2743014.

[16] Neil Jones, Carsten Gomard & Peter Sestoft (1993): *Partial Evaluation and Automatic Program Generation*. Prentice Hall International.

[17] Juhani Karhumäki, Hermann Maurer, Gheorghe Paun & Grzegorz Rozenberg (1999): *Jewels are Forever, Contributions on Theoretical Computer Science in Honor of Arto Salomaa*. doi:10.1007/978-3-642-60207-8_28.

[18] Quang Loc Le & Mengda He (2018): *A Decision Procedure for String Logic with Quadratic Equations, Regular Expressions and Length Constraints*, pp. 350–372. doi:10.1007/978-3-030-02768-1_19.

[19] Tianyi Liang, Andrew Reynolds, Nestan Tsiskaridze, Cesare Tinelli, Clark Barrett & Morgan Deters (2016): *An Efficient SMT Solver for String Constraints*. *Form. Methods Syst. Des.* 48(3), pp. 206–234, doi:10.1007/s10703-016-0247-6.

[20] Anthony Widjaja Lin & Rupak Majumdar (2018): *Quadratic Word Equations with Length Constraints, Counter Systems, and Presburger Arithmetic with Divisibility*. In: *Automated Technology for Verification and Analysis. ATVA 2018*, 11138, Lecture Notes in Computer Science, doi:10.1007/978-3-030-01090-4_21.

[21] Alexei Lisitsa & Andrei P. Nemytykh (2007): *A Note on Specialization of Interpreters*. In Volker Diekert, Mikhail V. Volkov & Andrei Voronkov, editors: *Computer Science – Theory and Applications*, Springer Berlin Heidelberg, pp. 237–248, doi:10.1007/978-3-540-74510-5_25.

[22] Alexei Lisitsa & Andrei P. Nemytykh (2008): *Reachability Analysis in Verification via Supercompilation*. *Int. J. Foundations of Computer Science* 19(4), pp. 953–970, doi:10.1142/S0129054108006066.

[23] V. Ganesh M. Berzish & Y. Zheng (2017): *Z3str3: A String Solver with Theory-aware Heuristics*. In: *Formal Methods in Computer Aided Design (FMCAD)*, pp. 55–59, doi:10.23919/FMCAD.2017.8102241.

[24] Gennadiy S. Makanin (1977): *The problem of solvability of equations in a free semigroup*. *Math. USSR-Sb.* 32(2), pp. 129–198, doi:10.1070/SM1977v032n02ABEH002376.

[25] Yuri Matiyasevich (1968): *A connection between systems of word and length equations and Hilbert's Tenth Problem (in Russian)*. *Sem. Mat. V. A. Steklov Math. Inst. Leningrad* 8, pp. 132–144.

[26] A. P. Nemytykh (2007): *The Supercompiler SCP4: General Structure (in Russian)*. URSS, Moscow.

[27] Andrei P. Nemytykh (2014): *On Unfolding for Programs Using Strings as a Data Type*. In: *VPT 2014. Second International Workshop on Verification and Program Transformation, July 17-18, 2014, Vienna, Austria, co-located with the 26th International Conference on Computer Aided Verification CAV 2014*, pp. 66–83, doi:10.29007/m8rr.

[28] Antonina Nepeivoda: *The page of supercompiler MSCP-A*. Available at `http://refal.botik.ru/mscp/mscp-a_eng.html`.

[29] Antonina Nepeivoda: *Specialization Results for Benchmark Equations*. Available at `https://github.com/TonitaN/TestEquations`.

[30] Antonina Nepeivoda: *Word Equations Interpreters and Experiments*. Available at `http://refal.botik.ru/mscp/weq_int_readme.html`.

[31] Antonina Nepeivoda (2016): *Ping-pong protocols as prefix grammars: Modelling and verification via program transformation*. *Journal of Logical and Algebraic Methods in Programming* 85(5), pp. 782–804, doi:10.1016/j.jlamp.2016.06.001. Special Issue on Automated Verification of Programs and Web Systems.

[32] Alberto Pettorossi & Maurizio Proietti (1996): *A comparative revisitation of some program transformation techniques*. In: *Partial Evaluation*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 355–385, doi:10.1007/3-540-61580-6_18.

[33] Wojciech Plandowski (2006): *An Efficient Algorithm for Solving Word Equations*. In: *Proceedings of 38th Annual ACM Symposium on Theory of Computing*, Association for Computing Machinery, New York, NY, USA, pp. 467–476, doi:10.1145/1132516.1132584.

[34] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant & D. Song (2010): *A symbolic execution framework for Javascript*. In: *SP*, pp. 513–528, doi:10.1109/SP.2010.38.

[35] Jens P. Secher & Morten Heine Sørensen (1999): *On Perfect Supercompilation*. In: *Perspectives of System Informatics, Third International Andrei Ershov Memorial Conference, PSI'99, Akademgorodok, Novosibirsk, Russia, July 6-9, 1999, Proceedings*, pp. 113–127, doi:10.1007/3-540-46562-6_10.

[36] Morten H. Sørensen & Robert Glück (1995): *An Algorithm of Generalization in Positive Supercompilation*. In: *Proceedings of ILPS'95, the International Logic Programming Symposium*, MIT Press, pp. 465–479, doi:10.7551/mitpress/4301.003.0048.

[37] Morten H. Sørensen, Robert Glück & Neil D. Jones (1993): *A Positive Supercompiler*. *Journal of Functional Programming* 6, pp. 465–479, doi:10.1017/s0956796800002008.

[38] M.-T. Trinh, D.-H. Chu & D.-H. Jaffar (2016): *Progressive Reasoning over Recursively-Defined Strings*. In: *Proc. CAV 2016 (LNCS)*, 9779, pp. 218–240, doi:10.1007/978-3-319-41528-4_12.

[39] Valentin F. Turchin (1986): *The Concept of a Supercompiler*. *ACM Transactions on Programming Languages and Systems* 8(3), pp. 292–325, doi:10.1145/5956.5957.

[40] Valentin F. Turchin (1989): *Refal-5, Programming Guide and Reference Manual*. New England Publishing Co., Holyoke, Massachusetts. Electronic version:`http://www.botik.ru/pub/local/scp/refal5/`.

[41] Valentin F. Turchin (1996): *On Generalization of Lists and Strings in Supercompilation*. In: *Technical Report CSc. TR 96-002*, City College of the City University of New York.

[42] Germán Vidal (2012): *Annotation of logic programs for independent AND-parallelism by partial evaluation*. *Theory Pract. Log. Program.* 12(4-5), pp. 583–600, doi:10.1017/S1471068412000191.

[43] Fang Yu, Tevfik Bultan & Oscar H. Ibarra (2011): *Relational String Verification Using Multi-track Automata*. In Michael Domaratzki & Kai Salomaa, editors: *Implementation and Application of Automata*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 290–299, doi:10.1007/978-3-642-18098-9_31.

Appendix

8.1 Proofs and Auxiliary Propositions

Proposition 6. *Let E be a reduced word equation $\Phi = \Psi$ with the var-permutated sides (and with no non-empty var-permutated suffixes and prefixes); σ be an arbitrary substitution given in Figure 3. Let $E' : \Phi\sigma = \Psi\sigma$ be of the reduced form. Then the two following properties hold.*

1. *If E' is split into $\mathrm{Pr}_1\,\mathrm{S}_1 = \mathrm{Pr}_2\,\mathrm{S}_2$ where Pr_1 and Pr_2 are non-empty and var-permutated, then either σ is $\mathtt{x} \mapsto \varepsilon$ or the equations $\mathrm{Pr}_1 = \mathrm{Pr}_2$ and $\mathrm{S}_1 = \mathrm{S}_2$ cannot be reduced.*
2. *If the length of E' is lesser than the length of E, then σ is $\mathtt{x} \mapsto \varepsilon$.*

Proof.
1. Let a substitution σ be $\sigma\colon \mathtt{x} \mapsto t\,\mathtt{x}$, where $t \in \mathscr{A} \cup \mathscr{V}$, and let $\Phi = \Phi_1\mathtt{x}\ldots\Phi_{n-1}\mathtt{x}\Phi_n$ and $\Psi = \Psi_1\mathtt{x}\ldots\Psi_{n-1}\mathtt{x}\Psi_n$, where $\forall i,j\,(|\Phi_i|_{\mathtt{x}} = 0\ \&\ |\Psi_j|_{\mathtt{x}} = 0)$. The substitution σ results in equation $\Phi_1 t\,\mathtt{x}\ldots\Phi_{n-1}\,t\,\mathtt{x}\Phi_n = \Psi_1 t\,\mathtt{x}\ldots\Psi_{n-1}\,t\,\mathtt{x}\Psi_n$. We consider the left-split operation finding non-empty var-permutated prefixes with the minimal length. The case of the right-split operation uses the analogous reasoning. If the var-permutated prefixes found by the operation are of the form $\Phi_1\,t\,\mathtt{x}\ldots t\,\mathtt{x}\Phi'_i$ and $\Psi_1\,t\,\mathtt{x}\ldots t\,\mathtt{x}\Psi'_i$, where Φ'_i and Ψ'_i are prefixes of Φ_i and Ψ_i respectively, then the words $\Phi_1\,\mathtt{x}\ldots\mathtt{x}\Phi'_i$ and $\Psi_1\,\mathtt{x}\ldots\mathtt{x}\Psi'_i$ are also var-permutated and the equation should be split until the substitution σ is applied. Thereby the var-permutated prefixes Pr_1 and Pr_2 can be only of the forms $\Phi_1\,t\,\mathtt{x}\ldots t\,\mathtt{x}\Phi'_i$ (where Φ'_i is a prefix of Φ_i) and $\Psi_1\,t\,\mathtt{x}\ldots t\,\mathtt{x}\Psi_i t$. Thus, S_1 starts with a term other than $\mathtt{x}$, while S_2 starts with $\mathtt{x}$. Moreover, if the last term of Φ'_i can be reduced with t, then the word $\Phi_1\,\mathtt{x}\ldots\mathtt{x}\Phi'_i$ and $\Psi_1\,\mathtt{x}\ldots\mathtt{x}\Psi_i$ are var-permutated[11].
2. Let σ be $\mathtt{x} \mapsto t\,\mathtt{x}$, where $\mathtt{x}$ occurs in Φ. The first case proven above allows us not to consider reduction operations after splitting the equation. Consider the possible reductions of the equation $E' : \Phi\sigma = \Psi\sigma$ before it is split. A reduction may occur only if Φ starts with $\mathtt{x}$, and Ψ starts with t (or vice versa). Let $k = |\Phi| + |\Psi|$, then $|\Phi\sigma| + |\Psi\sigma| \geq k+2$, and the considered reduction decreases the length of E' by 2. Thus, after an application of such a substitution σ the overall length of the equation cannot decrease.

□

The following proposition does not hold when the algorithm of interest is $\mathrm{AlgWE}_{\mathrm{Split}}$ ot $\mathrm{AlgWE}_{\mathrm{Count}}$.

Proposition 7. *Given a substitution $\sigma\colon \mathtt{x} \mapsto t\,\mathtt{x}$, where $t \in \mathscr{V} \cup \mathscr{A}$ ($t \neq \mathtt{x}$), σ is an injection on the set of reduced equations when they are simplified by the algorithm $\mathrm{AlgWE}_{\mathrm{Base}}$ unless the equations are trivial contradictions.*

Proof. Given a reduced equation $\Phi = \Psi$ *s.t.* Φ and Ψ start with different terms, let us assume that there exists E' *s.t.* $E = E'\sigma$. Let E' be $\Phi' = \Psi'$. If E' is of the reduced form then at most one elementary reduction can be done in $E'\sigma$, namely the we can reduce the first terms in the left- and right-hand sides of the equation. Moreover, the reduction occurs iff Φ' starts with $\mathtt{x}$ and Ψ' with t (or vice versa). If none of Φ and Ψ starts with $\mathtt{x}$, then no reduction is possible in $E'\sigma$ and E' can be computed as a result of the formal inverse substitution $\sigma^{-1} : t\,\mathtt{x} \mapsto \mathtt{x}$, namely $\Phi' = \Phi\sigma^{-1}$, $\Psi' = \Psi\sigma^{-1}$. Let $\Phi = \mathtt{x}\,\Phi_1$. Then Φ' is $\mathtt{x}\,(\Phi_1\sigma^{-1})$, Ψ' is $t\,(\Psi\sigma^{-1})$. □

Informally, Proposition 7 states that given an equation E and a substitution σ of a special kind *s.t.* the node N in a tree generated by algorithm $\mathrm{AlgWE}_{\mathrm{Base}}$ is labelled by E and has the ingoing arc labelled with σ, then the configuration of the parent of N can be restored. But this proposition does not require σ be generated according to the rules given in Figure 3: both Φ and Ψ may start with terms differing from $\mathtt{x}$.

In the following proof the notion of a configuration is used *w.r.t.* the program process trees, *i.e.* a configuration is a state of the call-stack which labels a node in the process tree.

Lemma 1 — Optimality Lemma. Let $[\mathsf{N}_1;\mathsf{N}_2]$ denote a path segment starting at N_1 and ending at N_2 .

First, consider the interpreter WIBase. (see subsection 8.2 for its source code). The node configurations in the process tree of WIBase are as follows:

1. $\mathtt{Main}(v, \underline{E})$;
2. $\mathtt{Main}(v, \underline{\mathtt{Sim}(\sigma, \textit{Other arguments})})$, where $\underline{\sigma}$ is a constant data representing the substitution which was applied last[12] to the equation list, and the other arguments may contain a call of the function `Subst`.

[11] This reasoning still holds if $i = 1$. In the case of $\Psi_1 = \varepsilon$ or $\Phi_1 = \varepsilon$, one reduction is to be done until the splitting, but the overall reasoning is the same.

[12] This substitution is stored as an argument of the function `Sim` as an additional annotation of the calls.

The function `Main` consequently applies substitutions given in its first argument to the equation list given in the second argument; the function `Sim` simplifies, *i.e.* reduces the equations. We call the configurations of the first form above *basic*. If only the basic configurations are folded, the specialization is already optimal. Let the fold operation occur on the nodes N_1 and N_2 labelled with the (equal up to the parameter renaming) configurations of the form $\mathtt{Main}(v_i, \mathtt{Sim}(\underline{\sigma, \text{Args}}))$. Let us consider their closest ancestors N_1' and N_2', such that their configurations are $\mathtt{Main}(v, \underline{E})$ and $\mathtt{Main}(v', \underline{E'})$ [13], and their closest successors such that they are labelled by the configurations $\mathtt{Main}(v_i', \underline{E_0})$. No narrowings generating a substitution $\mathtt{x} \mapsto \varepsilon$ can occur along the segment $[\mathsf{N}_1; \mathsf{N}_2]$, otherwise their configurations cannot be folded. Thus σ is $\mathtt{x} \mapsto t\,\mathtt{x}$. The substitution σ is the same in the configurations labelling N_1 and N_2, and they generate the same equation E_0 after this substitution, hence by Proposition 7 E and E' coincide.

Let us consider now specialization of interpreters WISplit and WICount. In these cases the possible configurations are exhausted with the following ones:

1. $\mathtt{Main}(v, \underline{(n){+}{+}\{E_i\}^+})$, where n coincides with the number of equations in the list unless the list consists of a single unsatisfiable equation;
2. $\mathtt{Main}(v, \mathtt{Sim}(\underline{n, \sigma, \textit{Other arguments}}))$, where n is the number of equations in the list at the last basic configuration, and σ is the last substitution that was applied to the equations; the other arguments may contain function calls;
3. $\mathtt{Main}(v, \mathtt{Sort}(\underline{n, \sigma, \textit{Other arguments}}))$.

The argumentation for the case 3 does not differ from the one for the case 2, thus we assume the folding works only with the nodes N_1 and N_2 with the configurations $\mathtt{Main}(v_i, \mathtt{Sim}(\underline{n, \sigma, \text{Args}}))$. Once again we consider their closest ancestors N_1' and N_2', such that they are labelled with the basic configurations $\mathtt{Main}(v, \underline{(n){+}{+}\{E_i\}^+})$ and $\mathtt{Main}(v', \underline{(n){+}{+}\{E_i'\}^+})$, as that was done for WIBase case. Function `Sim` takes its natural argument from the last `Main` call, hence it is the same in all the four configurations. The substitution $\mathtt{x} \mapsto \varepsilon$ cannot be applied along the path segment $[\mathsf{N}_1; \mathsf{N}_2]$. Let σ be $\mathtt{x} \mapsto t\,\mathtt{x}$. By Proposition 6, the number of equations in $[\mathsf{N}_1'; \mathsf{N}_2']$ cannot decrease. Thus, the number of equations is a constant (namely, n) along the path, and no equation in a configuration in $[\mathsf{N}_1'; \mathsf{N}_2']$ can be split. That means every equation in a configuration along the path starting at N_1' is transformed as it would be transformed by the algorithm $\text{AlgWE}_{\text{Base}}$. Given first successors N_{loop} and $\mathsf{N}_{\text{loop}}'$ of N_1 and N_2 such that they are labelled with the basic configurations (in a looped graph, these successors coincide), they are labelled by the equal lists of equations, thus by Proposition 7 the lists of equations $E_i'^{+}$ and E_i^{+} also coincide (see Figure 8). □

Proposition 8. *Let $Q_i, Q_i' \in \mathscr{A}^+$. Given a list of equations $\mathscr{E}qs = \langle Q_1\,\mathtt{x}_1 = \mathtt{x}_1\,Q_1', \dots, Q_n\,\mathtt{x}_n = \mathtt{x}_n\,Q_n'\rangle$, where for some i, j, $\mathtt{x}_i = \mathtt{x}_j$ may hold, every configuration in the solution tree generated by $\text{AlgWE}_{\text{Split}}(\mathscr{E}qs)$ contains at most n equations. If a configuration contains exactly n equations $\langle \Phi_1' = \Psi_1', \dots, \Phi_n' = \Psi_n'\rangle$, then $|\Phi_i'| \leq |Q_i| + 1$, $|\Psi_i'| \leq |Q_i'| + 1$.*

Proof. Figure 3 shows that the possible substitutions applicable to the initial equation list $\langle Q_1\,\mathtt{x}_1 = \mathtt{x}_1\,Q_1', \dots, Q_n\,\mathtt{x}_n = \mathtt{x}_n\,Q_n'\rangle$ are either $\mathtt{x}_i \mapsto \varepsilon$ or $\mathtt{x}_i \mapsto \mathbf{A}_i\,\mathtt{x}_i$. The first one transforms equations including $\mathtt{x}_i$ to either tautologies or contradictions. The second one transforms an equation $Q_i\,\mathtt{x}_i = \mathtt{x}_i\,Q_i'$ either to a contradiction or to an equation of the form $R_i\,\mathtt{x}_i = \mathtt{x}_i\,Q_i'$, where R_i is a cyclic permutation of the word Q_i, $|R_i| = |Q_i|$. □

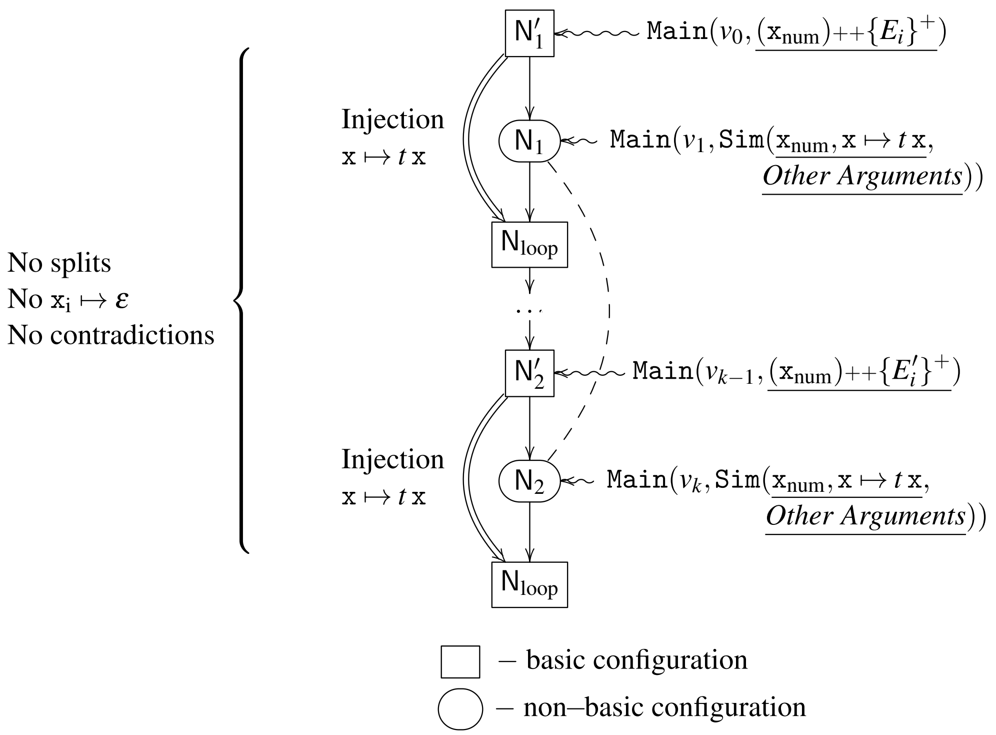
Corollary 1. *If $Q_i, Q_i' \in \mathscr{A}^+$ then every infinite path of a solution graph generated by the algorithm $\text{AlgWE}_{\text{Split}}$ applied to the list $\langle Q_1\,\mathtt{x}_1 = \mathtt{x}_1\,Q_1', \dots, Q_n\,\mathtt{x}_n = \mathtt{x}_n\,Q_n'\rangle$ contains two nodes with equal configurations.*

Proposition 9. *Let $\Phi = \Psi$ be a strictly regular-ordered equation with repetitions. Then the following statements hold.*

1. *Given the shortest non-empty var-permutated prefixes Φ_1 and Ψ_1 s.t. $\Phi = \Phi_1\,\Phi_2$, $\Psi = \Psi_1\,\Psi_2$, the equations $\Phi_1 = \Psi_1$ and $\Phi_2 = \Psi_2$ are strictly regular-ordered with repetitions.*
2. *The solution tree generated by the algorithm $\text{AlgWE}_{\text{Split}}(\Phi = \Psi)$ never includes an application of the rule $\mathtt{x} \mapsto \mathtt{y}\,\mathtt{x}$ given in Figure 6.*
3. *Every infinite path in the solution tree generated by $\text{AlgWE}_{\text{Split}}(\Phi = \Psi)$ contains a finite number of the split operations.*

Proof.
1. For every $\mathtt{x}_i \in \mathscr{V}$, the property of being var-permutated gives $|\Phi_1|_{\mathtt{x}_i} = |\Psi_1|_{\mathtt{x}_i}$, and hence $|\Phi_2|_{\mathtt{x}_i} = |\Psi_2|_{\mathtt{x}_i}$. The order of the variable occurrences in $\Phi = \Psi$ is preserved also in the prefixes and suffixes.
2. Consider the variable $\mathtt{x}$ leading in Φ (and Ψ). Then $\Phi = \mathtt{x}\,\Phi'$, $\Psi = \Psi_0\,\mathtt{x}\,\Psi'$ (or vice versa), where $\Psi_0 \in \mathscr{A}^+$. Let Ψ_0 be $\mathbf{A}\,\Psi_0'$ then a rule unfolding the equation $\mathtt{x}\,\Phi' = \Psi_0\,\mathtt{x}\,\Psi'$ is either $\mathtt{x} \mapsto \varepsilon$ or $\mathtt{x} \mapsto \mathbf{A}\,\mathtt{x}$. Both of the substitutions preserve the property of being strictly regular-ordered, as well as the splitting operation does.

[13] Such ancestors always exist, because the first call of `Sim` initialized by `Go` (and not preceded with a basic configuration) is of the form $\mathtt{Sim}(\varepsilon, \dots)$, while all other calls of `Sim` have a non-empty first argument taken from a basic configuration.



N'_1
$\mathtt{Main}(v_0, \underline{(\mathtt{x}_{\mathrm{num}})\mathbin{+\!+}\{E_i\}^+})$
Injection
$\mathtt{x} \mapsto t\,\mathtt{x}$
N_1
$\mathtt{Main}(v_1, \mathtt{Sim}(\underline{\mathtt{x}_{\mathrm{num}}, \mathtt{x} \mapsto t\,\mathtt{x}}, \underline{\textit{Other Arguments}}))$
$\mathsf{N}_{\mathrm{loop}}$
No splits
No $\mathtt{x}_i \mapsto \varepsilon$
No contradictions
N'_2
$\mathtt{Main}(v_{k-1}, \underline{(\mathtt{x}_{\mathrm{num}})\mathbin{+\!+}\{E'_i\}^+})$
Injection
$\mathtt{x} \mapsto t\,\mathtt{x}$
N_2
$\mathtt{Main}(v_k, \mathtt{Sim}(\underline{\mathtt{x}_{\mathrm{num}}, \mathtt{x} \mapsto t\,\mathtt{x}}, \underline{\textit{Other Arguments}}))$
$\mathsf{N}_{\mathrm{loop}}$
– basic configuration
– non–basic configuration

Figure 8: Scheme of the Optimality Lemma proof.

3. The property (2) follows that given a list $\langle \Phi_1 = \Psi_1, \ldots, \Phi_n = \Psi_n \rangle$ of equations labelling a configuration of the solution tree to $\Phi = \Psi$, for every $\mathtt{x}_i \in \mathscr{V}$, $\sum_{j=1}^{n} |\Phi_j|_{\mathtt{x}_i} + |\Psi_j|_{\mathtt{x}_i} \le |\Phi|_{\mathtt{x}_i} + |\Psi|_{\mathtt{x}_i}$. And every split operation generates two equations containing at least two variables.

□

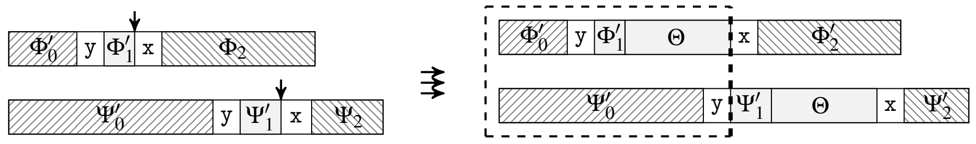
Lemma 2. *Given a strictly regular-ordered equation with repetitions $\Phi = \Psi$, every infinite path in its solution tree generated by the algorithm* $\mathrm{AlgWE_{Split}}(\Phi = \Psi)$ *contains nodes with equal configurations.*

Proof. Every infinite path in the tree generated by $\mathrm{AlgWE_{Split}}(\Phi = \Psi)$ has an infinite subpath satisfying the following two conditions:

1. equations are never split along the subpath;
2. variables are never mapped to ε along the subpath.

Let the first node in such a subpath be N, and the configuration of N be a list $\langle \Theta\,\mathtt{x}\,\Phi_1 = \mathtt{x}\,\Psi_1, \ldots, \Phi_n = \Psi_n \rangle$, $\Theta \in \mathscr{A}^+$. There may be the following three options.

1. For every j, if $|\Phi_j|_{\mathtt{x}} > 0$ then $\Phi_j = \mathtt{x}\,\Phi'_j$, $\Psi_j = \Theta_j\,\mathtt{x}\,\Psi'_j$ (or vice versa), $\Theta_j \in \mathscr{A}^+$, $|\Phi'_j|_{\mathtt{x}} = |\Psi'_j|_{\mathtt{x}} = 0$. Given such an equation, a substitution $\sigma\colon \mathtt{x} \mapsto t\,\mathtt{x}$ ($t \in \mathscr{A}$) preserves its length. The number of the equations in the configurations along the path, as well as their lengths, cannot grow, hence the set of the configurations along the path is finite.
2. Some equation $E_j\colon \Phi_j = \Psi_j$ is of the form $\Phi_{0,j}\,\mathtt{x}\,\Phi_{1,j}\,\mathtt{x}\,\Phi_{2,j} = \mathtt{x}\,\Psi_{1,j}\,\mathtt{x}\,\Psi_{2,j}$, where $\Phi_{0,j} \in \mathscr{A}^+$, $|\Phi_{1,j}|_{\mathtt{x}} = 0$, $|\Psi_{1,j}|_{\mathtt{x}} = 0$. Equation E_j is strictly regular-ordered, hence $\forall k (|\Phi_{1,j}|_{\mathtt{x}_k} = |\Psi_{1,j}|_{\mathtt{x}_k})$. Let $m = \big||\Phi_{0,j}| + |\Phi_{1,j}| - |\Psi_{1,j}|\big|$; $m \neq 0$, otherwise E_j would be split. Consider the path segment starting at N and having the length m. The arcs in this segment are labelled by the substitutions $\mathtt{x} \mapsto c_1\,\mathtt{x}, \ldots, \mathtt{x} \mapsto c_m\,\mathtt{x}$. Let $\Theta' = c_1 \ldots c_m$. The ending node of the m-length path is labelled by the following configuration: $\langle \ldots, \Phi'_{0,j}\,\mathtt{x}\,\Phi_{1,j}\,\Theta'\,\mathtt{x}\,\Phi'_{2,j} = \mathtt{x}\,\Psi_{1,j}\,\Theta'\,\mathtt{x}\,\Psi'_{2,j}, \ldots \rangle$, $|\Phi'_{0,j}| = |\Phi_{0,j}|$. If $|\Phi_{0,j}| + |\Phi_{1,j}| - |\Psi_{1,j}| > 0$, then the prefixes $\Phi'_{0,j}\,\mathtt{x}\,\Phi_{1,j}$ and $\mathtt{x}\,\Psi_{1,j}\,\Theta$ are var-permutated, otherwise the prefixes $\Phi'_{0,j}\,\mathtt{x}\,\Phi_{1,j}\,\Theta$ and $\mathtt{x}\,\Psi_{1,j}$ are var-permutated. In any case, at most in the m-th configuration (and maybe earlier, if $\Phi_{1,j}$ or $\Psi_{1,j}$ do not end with a variable) a split takes place.



▢ — in $\mathscr{A}^*$;
▨ — in $\{\mathscr{A} \cup \mathscr{V}\}^+$, does not contain x;
▧ — in $\{\mathscr{A} \cup \mathscr{V}\}^+$, may contain x.

Figure 9: Splitting a strictly regular-ordered equation with repetitions resulted by substitution $\mathtt{x}\sigma = \Theta\,\mathtt{x}$, where $|\Theta| = |\Psi'_0| - |\Phi'_0| - |\Phi'_1|$.

3. Some equation $E_j : \Phi_j = \Psi_j$ is of the form $\Phi_{1,j}\,\mathtt{x}\,\Phi_{2,j} = \Psi_{1,j}\,\mathtt{x}\,\Psi_{2,j}$, $|\Phi_{1,j}|_{\mathtt{x}} = 0$, $|\Psi_{1,j}|_{\mathtt{x}} = 0$, $\Phi_{1,j}, \Psi_{1,j} \notin \mathscr{A}^+$. Let $m = \Big||\Phi_{1,j}| - |\Psi_{1,j}|\Big|$. The same arguments show that given such an E_j a split would occur along the path at most after m substitutions. If $\Psi_{1,j}$ or $\Phi_{1,j}$ do not end with a variable, then the split can be applied earlier. That case is given in Figure 9, where $\Phi_{1,j} = \Phi'_0\,\mathtt{y}\,\Phi'_1$, $\Psi_{1,j} = \Psi'_0\,\mathtt{y}\,\Psi'_1$, $\Phi'_1, \Psi'_1 \in \mathscr{A}^+$.

Thus, either equal configurations exist along the given subpath or a split is constructed. That contradicts the choice of the subpath. □

The next two propositions use the following notations. The letters T and T' stand for words in $\mathscr{A} \cup \{\mathtt{x}\}$; t_i, u_i are letters in $\mathscr{A}$.

Proposition 10. *Given an equation $t_1 \dots t_n\,\mathtt{x}\,T = \mathtt{x}\,u_1 \dots u_m\,\mathtt{x}\,T'$ s.t. $n > 0$, $m \geq 0$, every infinite path of its solution graph contains a split.*

Proof. If $m \geq n$, the initial equation generates a split. Let $n = m + k$, $k > 0$, $\mathtt{x}\sigma_n = t_1 \dots t_n\,\mathtt{x}$. The k-th unfolding step observes that the equation is either already split or is of the following form:

$$t_{k+1} \dots t_n\, t_1 \dots t_k\,\mathtt{x}\,T\sigma_k = \mathtt{x}\,u_1 \dots u_m\, t_1 \dots t_k\,\mathtt{x}\,T'\sigma_k.$$

The equation has var-permutated prefixes. □

Lemma 3. *Given an equation $\Phi = \Psi$, $\Phi, \Psi \in \{\mathscr{A} \cup \{\mathtt{x}\}\}^*$, $|\Phi\Psi|_{\mathtt{x}} > 2$, every infinite path of its solution graph contains a split.*

Proof. Only the following three cases may appear.

1. $t_1 \dots t_n\,\mathtt{x}\,T = \mathtt{x}\,u_1 \dots u_m\,\mathtt{x}\,T'$, where $t_i \in \mathscr{A}$, $u_j \in \mathscr{A}$, $n > 0$. This case is considered in Proposition 10.
2. $\mathtt{x}\,u_1 \dots u_m = t_1 \dots t_n\,\mathtt{x}\,\Phi\,\mathtt{x}$, where $t_i \in \mathscr{A}$, $u_i \in \mathscr{A}$. Thus, the left-hand side of the equation contains the only occurrence of $\mathtt{x}$.
3. $\mathtt{x} = t_1 \dots t_n\,\mathtt{x}\,\Phi\,\mathtt{x}\,u_1 \dots u_k$, $k \geq 1$. The equation is contradictory according to Proposition 2.

Only the case (2) is of our interest. We assume $m > n$, otherwise the equation is split. Let $m = n * i + j$, $\mathtt{x}\sigma = t_1 \dots t_n\mathtt{x}$, $\mathtt{x}\sigma_j = t_1 \dots t_j\,\mathtt{x}$, $\xi = \sigma^i\sigma_j$. On the m-th unfolding step either the equation is split already or is of the following form: $\mathtt{x}\,u_1 \dots u_m = t_{j+1} \dots t_n\, t_1 \dots t_j\,\mathtt{x}\Phi\xi\,(t_1 \dots t_n)^i\, t_1 \dots t_j\,\mathtt{x}$. This equation has the var-permutated suffixes and is split. □

8.2 Source Code of Interpreters

The encoding of the input equation is as follows. Any constant letter from $\mathscr{A}$ is encoded with itself, as a single symbol. A variable from $\mathscr{V}$ is encoded with the term $(\mathbf{V}\,t)$ (*i.e.* the pair enclosed in the parentheses constructor), where $t \in \Sigma$ is a symbol encoding the unique identifier of the equation variable. The parentheses are also used to form a structure of functions' arguments, *e.g.* `Main` takes the two arguments, but the second one is a pair; the function `Sim` returns a pair.

For the sake of readability, we use the following syntactic sugar in the source code of the interpreters. Variables $\mathtt{x}_{\text{num}}$ and $\mathtt{x}_{\text{freshnum}}$ of the natural number type with the operations $+1$ and -1 are used instead of the unary Peano number representation, namely $(\mathbf{I}^n)$. An equation $\Phi_{\text{lhs}} = \Phi_{\text{rhs}}$ is encoded as $(\Phi_{\text{lhs}}, \Phi_{\text{rhs}})$ instead of $((\Phi_{\text{lhs}})(\Phi_{\text{rhs}}))$.
For example, the encoding of the equation $\mathbf{A}\,\mathtt{x}\,\mathtt{y} = \mathtt{x}\,\mathtt{y}\,\mathbf{A}$ in the source code of the interpreters is $(\mathbf{A}\,(\mathbf{V}\,\mathbf{X})\,(\mathbf{V}\,\mathbf{Y}), (\mathbf{V}\,\mathbf{X})\,(\mathbf{V}\,\mathbf{Y})\,\mathbf{A})$.

8.2.1 **Interpreter** WIBase

$$
\begin{array}{l}
\mathtt{Go}(\mathtt{x}_{\mathtt{rul}},\mathtt{x}_{\mathtt{val}}) = \mathtt{Main}(\mathtt{x}_{\mathtt{rul}},\mathtt{Sim}(\varepsilon,\mathtt{x}_{\mathtt{val}}));\\
\\
\mathtt{Main}(\varepsilon,(\varepsilon,\varepsilon)) = \mathbf{T};\\
\mathtt{Main}(((\mathbf{V}\mathtt{s}_x)\mapsto\varepsilon){+}{+}\mathtt{x}_{\mathtt{rul}},(\mathtt{x}_{\mathrm{LHS}},(\mathbf{V}\mathtt{s}_x)\,\mathtt{x}_{\mathrm{RHS}}))\\
\quad = \mathtt{Main}(\mathtt{x}_{\mathtt{rul}},\mathtt{Sim}(((\mathbf{V}\mathtt{s}_x)\mapsto\varepsilon),\mathtt{Subst}((\mathbf{V}\mathtt{s}_x)\mapsto\varepsilon,\varepsilon,\mathtt{x}_{\mathrm{LHS}}),\mathtt{Subst}(\mathbf{V}\mathtt{s}_x\mapsto\varepsilon,\varepsilon,\mathtt{x}_{\mathrm{RHS}})));\\
\mathtt{Main}(((\mathbf{V}\mathtt{s}_x)\mapsto\varepsilon){+}{+}\mathtt{x}_{\mathtt{rul}},((\mathbf{V}\mathtt{s}_x)\,\mathtt{x}_{\mathrm{LHS}},\mathtt{x}_{\mathrm{RHS}}))\\
\quad = \mathtt{Main}(\mathtt{x}_{\mathtt{rul}},\mathtt{Sim}(((\mathbf{V}\mathtt{s}_x)\mapsto\varepsilon),\mathtt{Subst}((\mathbf{V}\mathtt{s}_x)\mapsto\varepsilon,\varepsilon,\mathtt{x}_{\mathrm{LHS}}),\mathtt{Subst}((\mathbf{V}\mathtt{s}_x)\mapsto\varepsilon,\varepsilon,\mathtt{x}_{\mathrm{RHS}})));\\
\mathtt{Main}(((\mathbf{V}\mathtt{s}_x)\mapsto\mathtt{s}_{\mathtt{sym}}\,(\mathbf{V}\mathtt{s}_x)){+}{+}\mathtt{x}_{\mathtt{rul}},((\mathbf{V}\mathtt{s}_x)\,\mathtt{x}_{\mathrm{LHS}},\mathtt{s}_{\mathtt{sym}}\,\mathtt{x}_{\mathrm{RHS}}))\\
\quad = \mathtt{Main}(\mathtt{x}_{\mathtt{rul}},\mathtt{Sim}(((\mathbf{V}\mathtt{s}_x)\mapsto\mathtt{s}_{\mathtt{sym}}\,(\mathbf{V}\mathtt{s}_x)),\\
\qquad\qquad\qquad\qquad\mathtt{Subst}((\mathbf{V}\mathtt{s}_x)\mapsto\mathtt{s}_{\mathtt{sym}}\,(\mathbf{V}\mathtt{s}_x),(\mathbf{V}\mathtt{x}),\mathtt{x}_{\mathrm{LHS}}),\\
\qquad\qquad\qquad\qquad\mathtt{Subst}((\mathbf{V}\mathtt{s}_x)\mapsto\mathtt{s}_{\mathtt{sym}}\,(\mathbf{V}\mathtt{s}_x),\varepsilon,\mathtt{x}_{\mathrm{RHS}})));\\
\mathtt{Main}(((\mathbf{V}\mathtt{s}_x)\mapsto\mathtt{s}_{\mathtt{sym}}\,(\mathbf{V}\mathtt{s}_x)){+}{+}\mathtt{x}_{\mathtt{rul}},(\mathtt{s}_{\mathtt{sym}}\,\mathtt{x}_{\mathrm{LHS}},(\mathbf{V}\mathtt{s}_x)\,\mathtt{x}_{\mathrm{RHS}}))\\
\quad = \mathtt{Main}(\mathtt{x}_{\mathtt{rul}},\mathtt{Sim}(((\mathbf{V}\mathtt{s}_x)\mapsto\mathtt{s}_{\mathtt{sym}}\,(\mathbf{V}\mathtt{s}_x)),\\
\qquad\qquad\qquad\qquad\mathtt{Subst}((\mathbf{V}\mathtt{s}_x)\mapsto\mathtt{s}_{\mathtt{sym}}\,(\mathbf{V}\mathtt{s}_x),\varepsilon,\mathtt{x}_{\mathrm{LHS}}),\\
\qquad\qquad\qquad\qquad\mathtt{Subst}((\mathbf{V}\mathtt{s}_x)\mapsto\mathtt{s}_{\mathtt{sym}}\,(\mathbf{V}\mathtt{s}_x),(\mathbf{V}\mathtt{s}_x),\mathtt{x}_{\mathrm{RHS}})));\\
\mathtt{Main}(((\mathbf{V}\mathtt{s}_x)\mapsto(\mathbf{V}\mathtt{s}_y)\,(\mathbf{V}\mathtt{s}_x)){+}{+}\mathtt{x}_{\mathtt{rul}},((\mathbf{V}\mathtt{s}_y)\,\mathtt{x}_{\mathrm{LHS}},(\mathbf{V}\mathtt{s}_x)\,\mathtt{x}_{\mathrm{RHS}}))\\
\quad = \mathtt{Main}(\mathtt{x}_{\mathtt{rul}},\mathtt{Sim}(((\mathbf{V}\mathtt{s}_x)\mapsto(\mathbf{V}\mathtt{s}_y)\,(\mathbf{V}\mathtt{s}_x)),\\
\qquad\qquad\qquad\qquad\mathtt{Subst}((\mathbf{V}\mathtt{s}_x)\mapsto(\mathbf{V}\mathtt{s}_y)\,(\mathbf{V}\mathtt{s}_x),\varepsilon,\mathtt{x}_{\mathrm{LHS}}),\\
\qquad\qquad\qquad\qquad\mathtt{Subst}((\mathbf{V}\mathtt{s}_x)\mapsto(\mathbf{V}\mathtt{s}_y)\,(\mathbf{V}\mathtt{s}_x),(\mathbf{V}\mathtt{s}_x),\mathtt{x}_{\mathrm{RHS}})));\\
\mathtt{Main}(((\mathbf{V}\mathtt{s}_x)\mapsto(\mathbf{V}\mathtt{s}_y)\,(\mathbf{V}\mathtt{s}_x)){+}{+}\mathtt{x}_{\mathtt{rul}},((\mathbf{V}\mathtt{s}_x)\,\mathtt{x}_{\mathrm{LHS}},(\mathbf{V}\mathtt{s}_y)\,\mathtt{x}_{\mathrm{RHS}}))\\
\quad = \mathtt{Main}(\mathtt{x}_{\mathtt{rul}},\mathtt{Sim}(((\mathbf{V}\mathtt{s}_x)\mapsto(\mathbf{V}\mathtt{s}_y)\,(\mathbf{V}\mathtt{s}_x)),\\
\qquad\qquad\qquad\qquad\mathtt{Subst}((\mathbf{V}\mathtt{s}_x)\mapsto(\mathbf{V}\mathtt{s}_y)\,(\mathbf{V}\mathtt{s}_x),(\mathbf{V}\mathtt{s}_x),\mathtt{x}_{\mathrm{LHS}}),\\
\qquad\qquad\qquad\qquad\mathtt{Subst}((\mathbf{V}\mathtt{s}_x)\mapsto(\mathbf{V}\mathtt{s}_y)\,(\mathbf{V}\mathtt{s}_x),\varepsilon,\mathtt{x}_{\mathrm{RHS}})));\\
\mathtt{Main}(\mathtt{x}_{\mathtt{rul}},(\mathtt{x}_{\mathrm{LHS}},\mathtt{x}_{\mathrm{RHS}})) = \mathbf{F};
\end{array}
$$

$$
\begin{array}{l}
\mathtt{Subst}((\mathbf{V}\mathtt{s}_x)\mapsto\mathtt{xeqlist},\mathtt{x}_{\mathtt{res}},\varepsilon) = \mathtt{x}_{\mathtt{res}};\\
\mathtt{Subst}((\mathbf{V}\mathtt{s}_x)\mapsto\mathtt{xeqlist},\mathtt{x}_{\mathtt{res}},(\mathbf{V}\mathtt{s}_x)\,\mathtt{x}_{\mathtt{expr}}) = \mathtt{Subst}((\mathbf{V}\mathtt{s}_x)\mapsto\mathtt{xeqlist},\mathtt{x}_{\mathtt{res}}\,\mathtt{xeqlist},\mathtt{x}_{\mathtt{expr}});\\
\mathtt{Subst}((\mathbf{V}\mathtt{s}_x)\mapsto\mathtt{xeqlist},\mathtt{x}_{\mathtt{res}},\mathtt{s}_{\mathtt{sym}}\,\mathtt{x}_{\mathtt{expr}}) = \mathtt{Subst}((\mathbf{V}\mathtt{s}_x)\mapsto\mathtt{xeqlist},\mathtt{x}_{\mathtt{res}}\,\mathtt{s}_{\mathtt{sym}},\mathtt{x}_{\mathtt{expr}});\\
\mathtt{Subst}((\mathbf{V}\mathtt{s}_x)\mapsto\mathtt{xeqlist},\mathtt{x}_{\mathtt{res}},(\mathbf{V}\mathtt{s}_y)\,\mathtt{x}_{\mathtt{expr}}) = \mathtt{Subst}((\mathbf{V}\mathtt{s}_x)\mapsto\mathtt{xeqlist},\mathtt{x}_{\mathtt{res}}\,(\mathbf{V}\mathtt{s}_y),\mathtt{x}_{\mathtt{expr}});
\end{array}
$$

$$
\begin{array}{l}
\mathtt{Sim}(\mathtt{x}_{\mathrm{subst}},(\mathbf{V}\mathtt{s}_x)\,\mathtt{x}_{\mathrm{LHS}},(\mathbf{V}\mathtt{s}_x)\,\mathtt{x}_{\mathrm{LHS}}) = \mathtt{Sim}(\mathtt{x}_{\mathrm{subst}},\mathtt{x}_{\mathrm{LHS}},\mathtt{x}_{\mathrm{LHS}});\\
\mathtt{Sim}(\mathtt{x}_{\mathrm{subst}},\mathtt{s}_{\mathtt{sym}}\,\mathtt{x}_{\mathrm{LHS}},\mathtt{s}_{\mathtt{sym}}\,\mathtt{x}_{\mathrm{LHS}}) = \mathtt{Sim}(\mathtt{x}_{\mathrm{subst}},\mathtt{x}_{\mathrm{LHS}},\mathtt{x}_{\mathrm{LHS}});\\
\mathtt{Sim}(\mathtt{x}_{\mathrm{subst}},\mathtt{s}_{\mathtt{sym}_1}\,\mathtt{x}_{\mathrm{LHS}},\mathtt{s}_{\mathtt{sym}_2}\,\mathtt{x}_{\mathrm{LHS}}) = (\mathtt{s}_{\mathtt{sym}_1},\mathtt{s}_{\mathtt{sym}_2});\\
\mathtt{Sim}(\mathtt{x}_{\mathrm{subst}},\mathtt{x}_{\mathrm{LHS}}\,(\mathbf{V}\mathtt{s}_x),\mathtt{x}_{\mathrm{LHS}}\,(\mathbf{V}\mathtt{s}_x)) = \mathtt{Sim}(\mathtt{x}_{\mathrm{subst}},\mathtt{x}_{\mathrm{LHS}},\mathtt{x}_{\mathrm{LHS}});\\
\mathtt{Sim}(\mathtt{x}_{\mathrm{subst}},\mathtt{x}_{\mathrm{LHS}}\,\mathtt{s}_{\mathtt{sym}},\mathtt{x}_{\mathrm{LHS}}\,\mathtt{s}_{\mathtt{sym}}) = \mathtt{Sim}(\mathtt{x}_{\mathrm{subst}},\mathtt{x}_{\mathrm{LHS}},\mathtt{x}_{\mathrm{LHS}});\\
\mathtt{Sim}(\mathtt{x}_{\mathrm{subst}},\mathtt{x}_{\mathrm{LHS}}\,\mathtt{s}_{\mathtt{sym}_1},\mathtt{x}_{\mathrm{LHS}}\,\mathtt{s}_{\mathtt{sym}_2}) = (\mathtt{s}_{\mathtt{sym}_1},\mathtt{s}_{\mathtt{sym}_2});\\
\mathtt{Sim}(\mathtt{x}_{\mathrm{subst}},\mathtt{x}_{\mathrm{LHS}},\mathtt{x}_{\mathrm{LHS}}) = (\mathtt{x}_{\mathrm{LHS}},\mathtt{x}_{\mathrm{RHS}});
\end{array}
$$

8.2.2 **Interpreter** WISplit**.**

This interpreter has the following refinements as compared to WIBase.

- The functions `Main` and `Sim` take a list of equations rather than a single equation, as an input value of their last arguments.

- The function `Split` and the auxiliary multiset-handling function are added. In order to guarantee that all the equations resulted by a split are reduced, we introduce the `Reduce` function reducing a given single equation. The new function `Sort` transforms a list of equations to a single unsatisfiable equation if at least one contradiction is found in the list and otherwise it counts how many equations are included in the list.

- The functions `Main`, `Sim`, and `Sort` use an additional argument $\mathtt{x}_{\mathrm{num}}$ — a natural number which is 0 if the equations in the list are unchecked or contradictory and is the length of the list otherwise. This argument is used as the annotation that prevents unwanted fold operations in a process tree.

$$
\begin{array}{l}
\mathtt{Go}(x_{\mathtt{rul}}, x_{\mathtt{val}}) = \mathtt{Main}(x_{\mathtt{rul}}, \mathtt{Sim}(0, \varepsilon, \varepsilon, x_{\mathtt{val}}));\\
\\
\mathtt{Main}(\varepsilon, (x_{\mathrm{num}}){+\!+}((\varepsilon, \varepsilon))) = \mathbf{T};\\
\mathtt{Main}(x_{\mathtt{rul}}, (x_{\mathrm{num}}){+\!+}((\varepsilon, \varepsilon){+\!+}\mathtt{xeqs})) = \mathtt{Main}(x_{\mathtt{rul}}, (x_{\mathrm{num}} - 1){+\!+}\mathtt{xeqs});\\
\mathtt{Main}(((\mathbf{Vs}_x) \mapsto \varepsilon){+\!+}x_{\mathtt{rul}}, (x_{\mathrm{num}}){+\!+}((x_{\mathrm{LHS}}, (\mathbf{Vs}_x)\, x_{\mathrm{RHS}}){+\!+}\mathtt{xeqs}))\\
\quad = \mathtt{Main}(x_{\mathtt{rul}}, \mathtt{Sim}(x_{\mathrm{num}}, ((\mathbf{Vs}_x) \mapsto \varepsilon), \varepsilon, (\mathtt{Subst}((\mathbf{Vs}_x) \mapsto \varepsilon, \varepsilon, x_{\mathrm{LHS}}), \mathtt{Subst}((\mathbf{Vs}_x) \mapsto \varepsilon, \varepsilon, x_{\mathrm{RHS}})))\\
\qquad\qquad\qquad {+\!+}\mathtt{SubstAll}((\mathbf{Vs}_x) \mapsto \varepsilon, \mathtt{xeqs})));\\
\mathtt{Main}(((\mathbf{Vs}_x) \mapsto \varepsilon){+\!+}x_{\mathtt{rul}}, (x_{\mathrm{num}}){+\!+}(((\mathbf{Vs}_x)\, x_{\mathrm{LHS}}, x_{\mathrm{RHS}}){+\!+}\mathtt{xeqs}))\\
\quad = \mathtt{Main}(x_{\mathtt{rul}}, \mathtt{Sim}(x_{\mathrm{num}}, ((\mathbf{Vs}_x) \mapsto \varepsilon), \varepsilon, (\mathtt{Subst}((\mathbf{Vs}_x) \mapsto \varepsilon, \varepsilon, x_{\mathrm{LHS}}), \mathtt{Subst}((\mathbf{Vs}_x) \mapsto \varepsilon, \varepsilon, x_{\mathrm{RHS}}))\\
\qquad\qquad\qquad {+\!+}\mathtt{SubstAll}((\mathbf{Vs}_x) \mapsto \varepsilon, \mathtt{xeqs})));\\
\mathtt{Main}(((\mathbf{Vs}_x) \mapsto s_{\mathtt{sym}}\, (\mathbf{Vs}_x)){+\!+}x_{\mathtt{rul}}, (x_{\mathrm{num}}){+\!+}(((\mathbf{Vs}_x)\, x_{\mathrm{LHS}}, s_{\mathtt{sym}}\, x_{\mathrm{RHS}}){+\!+}\mathtt{xeqs}))\\
\quad = \mathtt{Main}(x_{\mathtt{rul}}, \mathtt{Sim}(x_{\mathrm{num}}, ((\mathbf{Vs}_x) \mapsto s_{\mathtt{sym}}\, (\mathbf{Vs}_x)), \varepsilon,\\
\qquad\qquad\qquad (\mathtt{Subst}((\mathbf{Vs}_x) \mapsto s_{\mathtt{sym}}\, (\mathbf{Vs}_x), (\mathbf{Vs}_x), x_{\mathrm{LHS}}), \mathtt{Subst}((\mathbf{Vs}_x) \mapsto s_{\mathtt{sym}}\, (\mathbf{Vs}_x), \varepsilon, x_{\mathrm{RHS}}))\\
\qquad\qquad\qquad {+\!+}\mathtt{SubstAll}((\mathbf{Vs}_x) \mapsto s_{\mathtt{sym}}\, (\mathbf{Vs}_x), \mathtt{xeqs})));\\
\mathtt{Main}(((\mathbf{Vs}_x) \mapsto s_{\mathtt{sym}}\, (\mathbf{Vs}_x)){+\!+}x_{\mathtt{rul}}, (x_{\mathrm{num}}){+\!+}((s_{\mathtt{sym}}\, x_{\mathrm{LHS}}, (\mathbf{Vs}_x)\, x_{\mathrm{RHS}}){+\!+}\mathtt{xeqs}))\\
\quad = \mathtt{Main}(x_{\mathtt{rul}}, \mathtt{Sim}(x_{\mathrm{num}}, ((\mathbf{Vs}_x) \mapsto s_{\mathtt{sym}}\, (\mathbf{Vs}_x)), \varepsilon,\\
\qquad\qquad\qquad (\mathtt{Subst}((\mathbf{Vs}_x) \mapsto s_{\mathtt{sym}}\, (\mathbf{Vs}_x), \varepsilon, x_{\mathrm{LHS}}), \mathtt{Subst}((\mathbf{Vs}_x) \mapsto s_{\mathtt{sym}}\, (\mathbf{Vs}_x), (\mathbf{Vs}_x), x_{\mathrm{RHS}}))\\
\qquad\qquad\qquad {+\!+}\mathtt{SubstAll}((\mathbf{Vs}_x) \mapsto s_{\mathtt{sym}}\, (\mathbf{Vs}_x), \mathtt{xeqs})));\\
\mathtt{Main}(((\mathbf{Vs}_x) \mapsto (\mathbf{Vs}_y)\, (\mathbf{Vs}_x)){+\!+}x_{\mathtt{rul}}, (x_{\mathrm{num}}){+\!+}(((\mathbf{Vs}_y)\, x_{\mathrm{LHS}}, (\mathbf{Vs}_x)\, x_{\mathrm{RHS}}){+\!+}\mathtt{xeqs}))\\
\quad = \mathtt{Main}(x_{\mathtt{rul}}, \mathtt{Sim}(x_{\mathrm{num}}, ((\mathbf{Vs}_x) \mapsto (\mathbf{Vs}_y)\, (\mathbf{Vs}_x)), \varepsilon,\\
\qquad\qquad\qquad (\mathtt{Subst}((\mathbf{Vs}_x) \mapsto (\mathbf{Vs}_y)\, (\mathbf{Vs}_x), \varepsilon, x_{\mathrm{LHS}}),\\
\qquad\qquad\qquad\ \mathtt{Subst}((\mathbf{Vs}_x) \mapsto (\mathbf{Vs}_y)\, (\mathbf{Vs}_x), (\mathbf{Vs}_x), x_{\mathrm{RHS}}))\\
\qquad\qquad\qquad {+\!+}\mathtt{SubstAll}((\mathbf{Vs}_x) \mapsto (\mathbf{Vs}_y)\, (\mathbf{Vs}_x), \mathtt{xeqs})));\\
\mathtt{Main}(((\mathbf{Vs}_x) \mapsto (\mathbf{Vs}_y)\, (\mathbf{Vs}_x)){+\!+}x_{\mathtt{rul}}, (x_{\mathrm{num}}){+\!+}(((\mathbf{Vs}_x)\, x_{\mathrm{LHS}}, (\mathbf{Vs}_y)\, x_{\mathrm{RHS}}){+\!+}\mathtt{xeqs}))\\
\quad = \mathtt{Main}(x_{\mathtt{rul}}, \mathtt{Sim}(x_{\mathrm{num}}, ((\mathbf{Vs}_x) \mapsto (\mathbf{Vs}_y)\, (\mathbf{Vs}_x)), \varepsilon,\\
\qquad\qquad\qquad (\mathtt{Subst}((\mathbf{Vs}_x) \mapsto (\mathbf{Vs}_y)\, (\mathbf{Vs}_x), (\mathbf{Vs}_x), x_{\mathrm{LHS}}),\\
\qquad\qquad\qquad\ \mathtt{Subst}((\mathbf{Vs}_x) \mapsto (\mathbf{Vs}_y)\, (\mathbf{Vs}_x), \varepsilon, x_{\mathrm{RHS}}))\\
\qquad\qquad\qquad {+\!+}\mathtt{SubstAll}((\mathbf{Vs}_x) \mapsto (\mathbf{Vs}_y)\, (\mathbf{Vs}_x), \mathtt{xeqs})));\\
\mathtt{Main}(x_{\mathtt{rul}}, (x_{\mathrm{num}}){+\!+}\mathtt{xeqs}) = \mathbf{F};
\end{array}
$$

$$
\begin{array}{l}
\mathtt{Subst}((\mathbf{Vs}_x) \mapsto \mathtt{xeqlist}, x_{\mathrm{res}}, \varepsilon) = x_{\mathrm{res}};\\
\mathtt{Subst}((\mathbf{Vs}_x) \mapsto \mathtt{xeqlist}, x_{\mathrm{res}}, (\mathbf{Vs}_x)\, x_{\mathrm{expr}}) = \mathtt{Subst}((\mathbf{Vs}_x) \mapsto \mathtt{xeqlist}, x_{\mathrm{res}}\, \mathtt{xeqlist}, x_{\mathrm{expr}});\\
\mathtt{Subst}((\mathbf{Vs}_x) \mapsto \mathtt{xeqlist}, x_{\mathrm{res}}, s_{\mathrm{sym}}\, x_{\mathrm{expr}}) = \mathtt{Subst}((\mathbf{Vs}_x) \mapsto \mathtt{xeqlist}, x_{\mathrm{res}}\, s_{\mathrm{sym}}, x_{\mathrm{expr}});\\
\mathtt{Subst}((\mathbf{Vs}_x) \mapsto \mathtt{xeqlist}, x_{\mathrm{res}}, (\mathbf{Vs}_y)\, x_{\mathrm{expr}}) = \mathtt{Subst}((\mathbf{Vs}_x) \mapsto \mathtt{xeqlist}, x_{\mathrm{res}}\, (\mathbf{Vs}_y), x_{\mathrm{expr}});\\
\\
\mathtt{SubstAll}((\mathbf{Vs}_x) \mapsto \mathtt{xeqlist}, (x_{\mathrm{LHS}}, x_{\mathrm{RHS}}){+\!+}\mathtt{xeqs})\\
\quad = (\mathtt{Subst}((\mathbf{Vs}_x) \mapsto \mathtt{xeqlist}, \varepsilon, x_{\mathrm{LHS}}), \mathtt{Subst}((\mathbf{Vs}_x) \mapsto \mathtt{xeqlist}, \varepsilon, x_{\mathrm{RHS}}))\\
\qquad {+\!+}\mathtt{SubstAll}(((\mathbf{Vs}_x) \mapsto \mathtt{xeqlist}, \mathtt{xeqs});\\
\mathtt{SubstAll}((\mathbf{Vs}_x) \mapsto \mathtt{xeqlist}, \varepsilon) = \varepsilon;
\end{array}
$$

$$
\begin{array}{l}
\mathtt{Sim}(x_{\mathrm{num}}, x_{\mathrm{subst}}, x_{\mathrm{res}}, ((\mathbf{Vs}_x)\, x_{\mathrm{LHS}}, (\mathbf{Vs}_x)\, x_{\mathrm{RHS}}){+\!+}\mathtt{xeqs})\\
\quad = \mathtt{Sim}(x_{\mathrm{num}}, x_{\mathrm{subst}}, x_{\mathrm{res}}, (x_{\mathrm{LHS}}, x_{\mathrm{RHS}}){+\!+}\mathtt{xeqs});\\
\mathtt{Sim}(x_{\mathrm{num}}, x_{\mathrm{subst}}, x_{\mathrm{res}}, (s_{\mathrm{sym}}\, x_{\mathrm{LHS}}, s_{\mathrm{sym}}\, x_{\mathrm{RHS}}){+\!+}\mathtt{xeqs})\\
\quad = \mathtt{Sim}(x_{\mathrm{num}}, x_{\mathrm{subst}}, x_{\mathrm{res}}, (x_{\mathrm{LHS}}, x_{\mathrm{RHS}}){+\!+}\mathtt{xeqs});\\
\mathtt{Sim}(x_{\mathrm{num}}, x_{\mathrm{subst}}, x_{\mathrm{res}}, (x_{\mathrm{LHS}}\, (\mathbf{Vs}), x_{\mathrm{RHS}}\, (\mathbf{Vs})){+\!+}\mathtt{xeqs})\\
\quad = \mathtt{Sim}(x_{\mathrm{num}}, x_{\mathrm{subst}}, x_{\mathrm{res}}, (x_{\mathrm{LHS}}, x_{\mathrm{RHS}}){+\!+}\mathtt{xeqs});\\
\mathtt{Sim}(x_{\mathrm{num}}, x_{\mathrm{subst}}, x_{\mathrm{res}}, (x_{\mathrm{LHS}}\, s_{\mathrm{sym}}, x_{\mathrm{RHS}}\, s_{\mathrm{sym}}){+\!+}\mathtt{xeqs})\\
\quad = \mathtt{Sim}(x_{\mathrm{num}}, x_{\mathrm{subst}}, x_{\mathrm{res}}, (x_{\mathrm{LHS}}, x_{\mathrm{RHS}}){+\!+}\mathtt{xeqs});\\
\mathtt{Sim}(x_{\mathrm{num}}, x_{\mathrm{subst}}, x_{\mathrm{res}}, (s_{\mathrm{sym}_1}\, x_{\mathrm{LHS}}, s_{\mathrm{sym}_2}\, x_{\mathrm{RHS}}){+\!+}\mathtt{xeqs}) = (0){+\!+}(s_{\mathrm{sym}_1}, s_{\mathrm{sym}_2});\\
\mathtt{Sim}(x_{\mathrm{num}}, x_{\mathrm{subst}}, x_{\mathrm{res}}, (x_{\mathrm{LHS}}\, s_{\mathrm{sym}_1}, x_{\mathrm{RHS}}\, s_{\mathrm{sym}_2}){+\!+}\mathtt{xeqs}) = (0){+\!+}(s_{\mathrm{sym}_1}, s_{\mathrm{sym}_2});\\
\mathtt{Sim}(x_{\mathrm{num}}, x_{\mathrm{subst}}, x_{\mathrm{res}}, (\varepsilon, \varepsilon){+\!+}\mathtt{xeqs}) = \mathtt{Sim}(x_{\mathrm{num}}, x_{\mathrm{subst}}, x_{\mathrm{res}}\, (\varepsilon, \varepsilon){+\!+}\mathtt{xeqs});\\
\mathtt{Sim}(x_{\mathrm{num}}, x_{\mathrm{subst}}, x_{\mathrm{res}}, (x_{\mathrm{LHS}}, x_{\mathrm{RHS}}){+\!+}\mathtt{xeqs})\\
\quad = \mathtt{Sim}(x_{\mathrm{num}}, x_{\mathrm{subst}}, x_{\mathrm{res}}{+\!+}\mathtt{Split}(\varepsilon, \mathbf{N}, ((\mathbf{CONST}\ 0)), ((\mathbf{CONST}\ 0)), \varepsilon, \varepsilon, x_{\mathrm{LHS}}, x_{\mathrm{RHS}}, \mathtt{xeqs});\\
\mathtt{Sim}(x_{\mathrm{num}}, x_{\mathrm{subst}}, x_{\mathrm{res}}) = \mathtt{Sort}(x_{\mathrm{num}}, 0, x_{\mathrm{subst}}, \varepsilon, x_{\mathrm{res}});\\
\\
\mathtt{Sort}(x_{\mathrm{num}}, x_{\mathrm{freshnum}}, x_{\mathrm{subst}}, x_{\mathrm{res}}, (s_{\mathrm{sym}_1}, s_{\mathrm{sym}_2}){+\!+}\mathtt{xeqs}) = (0){+\!+}(s_{\mathrm{sym}_1}, s_{\mathrm{sym}_2});\\
\mathtt{Sort}(x_{\mathrm{num}}, x_{\mathrm{freshnum}}, x_{\mathrm{subst}}, x_{\mathrm{res}}, (x_{\mathrm{eq}}){+\!+}\mathtt{xeqs})\\
\quad = \mathtt{Sort}(x_{\mathrm{num}}, x_{\mathrm{freshnum}} + 1, x_{\mathrm{subst}}, x_{\mathrm{res}}{+\!+}(x_{\mathrm{eq}}), \mathtt{xeqs});\\
\mathtt{Sort}(x_{\mathrm{num}}, x_{\mathrm{freshnum}}, x_{\mathrm{subst}}, x_{\mathrm{res}}, \varepsilon) = (x_{\mathrm{freshnum}}){+\!+}x_{\mathrm{res}};
\end{array}
$$

$\mathtt{Reduce}((\mathbf{V}\,\mathtt{s}_x){+}{+}\mathtt{x}_{\mathrm{LHS}},(\mathbf{V}\,\mathtt{s}_x){+}{+}\mathtt{x}_{\mathrm{RHS}}) = \mathtt{Reduce}(\mathtt{x}_{\mathrm{LHS}},\mathtt{x}_{\mathrm{RHS}});$
$\mathtt{Reduce}(\mathtt{s}{+}{+}\mathtt{x}_{\mathrm{LHS}},\mathtt{s}{+}{+}\mathtt{x}_{\mathrm{RHS}}) = \mathtt{Reduce}(\mathtt{x}_{\mathrm{LHS}},\mathtt{x}_{\mathrm{RHS}});$
$\mathtt{Reduce}(\mathtt{x}_{\mathrm{LHS}}{+}{+}(\mathbf{V}\,\mathtt{s}_x),\mathtt{x}_{\mathrm{RHS}}{+}{+}(\mathbf{V}\,\mathtt{s}_x)) = \mathtt{Reduce}(\mathtt{x}_{\mathrm{LHS}},\mathtt{x}_{\mathrm{RHS}});$
$\mathtt{Reduce}(\mathtt{x}_{\mathrm{LHS}}{+}{+}\mathtt{s},\mathtt{x}_{\mathrm{RHS}}{+}{+}\mathtt{s}) = \mathtt{Reduce}(\mathtt{x}_{\mathrm{LHS}},\mathtt{x}_{\mathrm{RHS}});$
$\mathtt{Reduce}(\mathtt{s}_1{+}{+}\mathtt{x}_{\mathrm{LHS}},\mathtt{s}_2{+}{+}\mathtt{x}_{\mathrm{RHS}}) = (\mathtt{s}_1,\mathtt{s}_2);$
$\mathtt{Reduce}(\mathtt{x}_{\mathrm{LHS}}{+}{+}\mathtt{s}_1,\mathtt{x}_{\mathrm{RHS}}{+}{+}\mathtt{s}_2) = (\mathtt{s}_1,\mathtt{s}_2);$
$\mathtt{Reduce}(\mathtt{x}_{\mathrm{LHS}},\mathtt{x}_{\mathrm{RHS}}) = (\mathtt{x}_{\mathrm{LHS}},\mathtt{x}_{\mathrm{RHS}});$

$\mathtt{Split}(\mathtt{x}_{\mathtt{res}},\mathbf{N},(\mathtt{xms}_1),(\mathtt{xms}_2),\mathtt{xpref}_{\mathrm{LHS}},\mathtt{xpref}_{\mathrm{RHS}},t_1\,\mathtt{x}_{\mathrm{LHS}},t_2\,\mathtt{x}_{\mathrm{RHS}})$
$\quad = \mathtt{Split}(\mathtt{x}_{\mathtt{res}},$
$\qquad\quad \mathtt{CountMS}(\mathtt{Include}(t_1,\varepsilon,\mathtt{xms}_1),\mathtt{Include}(t_2,\varepsilon,\mathtt{xms}_2)),\mathtt{xpref}_{\mathrm{LHS}}\,t_1,\mathtt{xpref}_{\mathrm{RHS}}\,t_2,\mathtt{x}_{\mathrm{LHS}},\mathtt{x}_{\mathrm{RHS}});$
$\mathtt{Split}(\mathtt{x}_{\mathtt{res}},\mathbf{F},(\mathtt{xms}_1),(\mathtt{xms}_2),\mathtt{xpref}_{\mathrm{LHS}},\mathtt{xpref}_{\mathrm{RHS}},t_1\,\mathtt{x}_{\mathrm{LHS}},t_2\,\mathtt{x}_{\mathrm{RHS}})$
$\quad = \mathtt{Split}(\mathtt{x}_{\mathtt{res}},$
$\qquad\quad \mathtt{CountMS}(\mathtt{Include}(t_1,\varepsilon,\mathtt{xms}_1),\mathtt{Include}(t_2,\varepsilon,\mathtt{xms}_2)),\mathtt{xpref}_{\mathrm{LHS}}\,t_1,\mathtt{xpref}_{\mathrm{RHS}}\,t_2,\mathtt{x}_{\mathrm{LHS}},\mathtt{x}_{\mathrm{RHS}});$
$\mathtt{Split}(\mathtt{x}_{\mathtt{res}},\mathbf{T},(\mathtt{xms}_1),(\mathtt{xms}_2),\mathtt{xpref}_{\mathrm{LHS}},\mathtt{xpref}_{\mathrm{RHS}},\mathtt{x}_{\mathrm{LHS}},\mathtt{x}_{\mathrm{RHS}})$
$\quad = \mathtt{Split}(\mathtt{x}_{\mathtt{res}}{+}{+}(\mathtt{xpref}_{\mathrm{LHS}},\mathtt{xpref}_{\mathrm{RHS}}),\mathbf{N},((\mathbf{CONST}\,0)),((\mathbf{CONST}\,0)),\varepsilon,\varepsilon,\mathtt{x}_{\mathrm{LHS}},\mathtt{x}_{\mathrm{RHS}});$
$\mathtt{Split}(\mathtt{x}_{\mathtt{res}},\mathtt{s}_{\mathtt{sym}},(\mathtt{xms}_1),(\mathtt{xms}_2),\varepsilon,\varepsilon,\varepsilon,\varepsilon) = \mathtt{x}_{\mathtt{res}};$
$\mathtt{Split}(\mathtt{x}_{\mathtt{res}},\mathtt{s}_{\mathtt{sym}},(\mathtt{xms}_1),(\mathtt{xms}_2),\mathtt{xpref}_{\mathrm{LHS}},\mathtt{xpref}_{\mathrm{RHS}},\mathtt{x}_{\mathrm{LHS}},\mathtt{x}_{\mathrm{RHS}})$
$\quad = \mathtt{Reduce}(\mathtt{xpref}_{\mathrm{LHS}}\,\mathtt{x}_{\mathrm{LHS}},\mathtt{xpref}_{\mathrm{RHS}}\,\mathtt{x}_{\mathrm{RHS}}){+}{+}\mathtt{x}_{\mathtt{res}};$

$\mathtt{Include}(\mathtt{s}_{\mathtt{sym}},\mathtt{x}_{\mathtt{prev}},(\mathtt{xms},(\mathbf{CONST}\,\mathtt{xnum}))) = (\mathtt{x}_{\mathtt{prev}},(\mathbf{CONST}\,\mathtt{xnum}+1));$
$\mathtt{Include}((\mathbf{V}\mathtt{s}_x),\mathtt{x}_{\mathtt{prev}},(((\mathbf{V}\mathtt{s}_x)\ \mathtt{xnum}),\mathtt{xms})) = (\mathtt{x}_{\mathtt{prev}},((\mathbf{V}\mathtt{s}_x)\ \mathtt{xnum}+1),\mathtt{xms});$
$\mathtt{Include}((\mathbf{V}\mathtt{s}_x),\mathtt{x}_{\mathtt{prev}},(((\mathbf{V}\mathtt{s}_y)\ \mathtt{xnum})\,\mathtt{xms})) = \mathtt{Include}((\mathbf{V}\mathtt{s}_x),\mathtt{x}_{\mathtt{prev}}\,((\mathbf{V}\mathtt{s}_y)\ \mathtt{xnum}),\mathtt{xms});$
$\mathtt{Include}((\mathbf{V}\mathtt{s}_x),\mathtt{x}_{\mathtt{prev}},((\mathbf{CONST}\,\mathtt{xnum}))) = (((\mathbf{V}\mathtt{s}_x)\ 1)\,\mathtt{x}_{\mathtt{prev}}\,(\mathbf{CONST}\,\mathtt{xnum}));$

$\mathtt{CountMS}(t\,\mathtt{xms}_1,\mathtt{xms}_2)$
$\quad = \mathtt{AreEqual}(\mathtt{xms}_1,\mathtt{ElMinus}(t,\varepsilon,\mathtt{xms}_2))\,(t\,\mathtt{xms}_1)(\mathtt{xms}_2);$

$\mathtt{AreEqual}(\mathtt{xms}_1,\mathtt{xms}_2\,\mathbf{F}) = \mathbf{F};$
$\mathtt{AreEqual}(\varepsilon,\varepsilon) = \mathbf{T};$
$\mathtt{AreEqual}(\mathtt{xms}_1,\varepsilon) = \mathbf{F};$
$\mathtt{AreEqual}(\varepsilon,\mathtt{xms}_2) = \mathbf{F};$
$\mathtt{AreEqual}(((\mathbf{V}\mathtt{s}_{\mathtt{sym}})\,\mathtt{xnum})\,\mathtt{xms}_1,\mathtt{xms}_2) = \mathtt{AreEqual}(\mathtt{xms}_1,\mathtt{ElMinus}(((\mathbf{V}\mathtt{s}_{\mathtt{sym}})\,\mathtt{xnum}),\varepsilon,\mathtt{xms}_2));$
$\mathtt{AreEqual}((\mathbf{CONST}\ \mathtt{xnum})\,\mathtt{xms}_1,\mathtt{xms}_2) = \mathtt{AreEqual}(\mathtt{xms}_1,\mathtt{ElMinus}((\mathbf{CONST}\ \mathtt{xnum}),\varepsilon,\mathtt{xms}_2));$

$\mathtt{ElMinus}((\mathbf{CONST}\,\mathtt{xnum}),\varepsilon,(\mathtt{xms}\,(\mathbf{CONST}\,\mathtt{xnum}))) = (\mathtt{xms});$
$\mathtt{ElMinus}(((\mathbf{V}\mathtt{s}_x)\ \mathtt{xnum}),\mathtt{x}_{\mathtt{prev}},(((\mathbf{V}\mathtt{s}_x)\ \mathtt{xnum})\,\mathtt{xms})) = (\mathtt{x}_{\mathtt{prev}}\,\mathtt{xms});$
$\mathtt{ElMinus}(((\mathbf{V}\mathtt{s}_x)\ \mathtt{xnum}_1),\mathtt{x}_{\mathtt{prev}},(((\mathbf{V}\mathtt{s}_x)\ \mathtt{xnum}_2)\,\mathtt{xms})) = \mathbf{F};$
$\mathtt{ElMinus}(((\mathbf{V}\mathtt{s}_x)\ \mathtt{xnum}_1),\mathtt{x}_{\mathtt{prev}},(((\mathbf{V}\mathtt{s}_y)\ \mathtt{xnum}_2)\,\mathtt{xms})) = \mathtt{ElMinus}(((\mathbf{V}\mathtt{s}_x)\ \mathtt{xnum}_1),\mathtt{x}_{\mathtt{prev}}\,((\mathbf{V}\mathtt{s}_y)\ \mathtt{xnum}_2),\mathtt{xms});$
$\mathtt{ElMinus}(((\mathbf{V}\mathtt{s}_x)\ \mathtt{xnum}_1),\mathtt{x}_{\mathtt{prev}},((\mathbf{CONST}\,\mathtt{xnum}_2))) = \mathbf{F};$

8.2.3 **Interpreter** WICount.

This interpreter uses the function definitions given for the interpreter WISplit plus some additional functions, provided that the function `ElMinus` is modified and the last rule of the function `Split` is replaced with the following rewriting rule.

$/*\,\mathtt{This\ rule\ replaces\ the\ last\ rule\ of\ Split\ definition\ given\ in}\ \mathrm{WISplit}.\,*/$
$\mathtt{Split}(\mathtt{x}_{\mathtt{res}},\mathtt{s}_{\mathtt{sym}},(\mathtt{xms}_1),(\mathtt{xms}_2),\mathtt{xpref}_{\mathrm{LHS}},\mathtt{xpref}_{\mathrm{RHS}},\mathtt{x}_{\mathrm{LHS}},\mathtt{x}_{\mathrm{RHS}})$
$\quad = \mathtt{SplitR}(\mathtt{x}_{\mathtt{res}},\mathbf{N},((\mathbf{CONST}\,0)),((\mathbf{CONST}\,0)),\varepsilon,\varepsilon,\mathtt{Reduce}(\mathtt{xpref}_{\mathrm{LHS}}\,\mathtt{x}_{\mathrm{LHS}},\mathtt{xpref}_{\mathrm{RHS}}\,\mathtt{x}_{\mathrm{RHS}}));$

The additional functions definitions are given below. The function `ElMinus` replaces the version given in the WISplit source code.

$$\begin{array}{l}
\mathtt{ElMinus}((\mathbf{CONST}\ \mathtt{xnum}_1),\varepsilon,\mathtt{xms}\,(\mathbf{CONST}\ \mathtt{xnum}_2)) = \mathtt{xms}\,\mathtt{CountMinus}(\mathtt{xnum}_1,\mathtt{xnum}_2);\\
\mathtt{ElMinus}(((\mathbf{V}\mathtt{s}_{\mathtt{sym}})\ \mathtt{xnum}_1),\mathtt{x}_{\mathtt{prev}},((\mathbf{V}\mathtt{s}_{\mathtt{sym}})\ \mathtt{xnum}_2)\,\mathtt{xms}) = \mathtt{x}_{\mathtt{prev}}\,\mathtt{xms}\,\mathtt{CountMinus}(\mathtt{xnum}_1,\mathtt{xnum}_2);\\
\mathtt{ElMinus}(((\mathbf{V}\mathtt{s}_{\mathtt{sym}})\ \mathtt{xnum}_1),\mathtt{x}_{\mathtt{prev}},(\mathbf{CONST}\ \mathtt{xnum}_2)) = \mathtt{x}_{\mathtt{prev}}\,\mathbf{G};\\
\mathtt{ElMinus}(((\mathbf{V}\mathtt{s}_{\mathtt{sym}_1})\ \mathtt{xnum}_1),\mathtt{x}_{\mathtt{prev}},((\mathbf{V}\mathtt{s}_{\mathtt{sym}_2})\ \mathtt{xnum}_2)\,\mathtt{xms}) = \mathtt{ElMinus}(((\mathbf{V}\mathtt{s}_{\mathtt{sym}_1})\ \mathtt{xnum}_1),\mathtt{x}_{\mathtt{prev}}\,((\mathbf{V}\mathtt{s}_{\mathtt{sym}_2})\ \mathtt{xnum}_2),\mathtt{xms});
\end{array}$$

$$\begin{array}{l}
\mathtt{CountMinus}(\mathtt{xnum},\mathtt{xnum}) = \varepsilon;\\
\mathtt{CountMinus}(\mathtt{xnum}_1+1,\mathtt{xnum}_2+1) = \mathtt{CountMinus}(\mathtt{xnum}_1,\mathtt{xnum}_2);\\
\mathtt{CountMinus}(\varepsilon,\mathtt{xnum}) = \mathbf{L};\\
\mathtt{CountMinus}(\mathtt{xnum},\varepsilon) = \mathbf{G};
\end{array}$$

$$\begin{array}{l}
\mathtt{SplitR}(\mathtt{x}_{\mathtt{res}},\mathtt{s}_{\mathtt{sym}},\mathtt{xms}_1,\mathtt{xms}_2,\varepsilon,\varepsilon,\varepsilon,\varepsilon) = \mathtt{x}_{\mathtt{res}};\\
\mathtt{SplitR}(\mathtt{x}_{\mathtt{res}},\mathbf{N},\mathtt{xms}_1,\mathtt{xms}_2,\mathtt{xsuff}_{\mathrm{LHS}},\mathtt{xsuff}_{\mathrm{RHS}},\mathtt{x}_{\mathrm{LHS}}\,t_1,\mathtt{x}_{\mathrm{RHS}}\,t_2)\\
\quad = \mathtt{SplitR}(\mathtt{x}_{\mathtt{res}},\\
\qquad\qquad \mathtt{CountMS}(\mathtt{Include}(t_1,\varepsilon,\mathtt{xms}_1),\mathtt{Include}(t_2,\varepsilon,\mathtt{xms}_2),t_1\,\mathtt{xsuff}_{\mathrm{LHS}},t_2\,\mathtt{xsuff}_{\mathrm{RHS}},\mathtt{x}_{\mathrm{LHS}},\mathtt{x}_{\mathrm{RHS}}));\\
\mathtt{SplitR}(\mathtt{x}_{\mathtt{res}},\mathbf{F},\mathtt{xms}_1,\mathtt{xms}_2,\mathtt{xsuff}_{\mathrm{LHS}},\mathtt{xsuff}_{\mathrm{RHS}},\mathtt{x}_{\mathrm{LHS}}\,t_1,\mathtt{x}_{\mathrm{RHS}}\,t_2)\\
\quad = \mathtt{SplitR}(\mathtt{x}_{\mathtt{res}},\\
\qquad\qquad \mathtt{CountMS}(\mathtt{Include}(t_1,\varepsilon,\mathtt{xms}_1),\mathtt{Include}(t_2,\varepsilon,\mathtt{xms}_2),t_1\,\mathtt{xsuff}_{\mathrm{LHS}},t_2\,\mathtt{xsuff}_{\mathrm{RHS}},\mathtt{x}_{\mathrm{LHS}},\mathtt{x}_{\mathrm{RHS}}));\\
\mathtt{SplitR}(\mathtt{x}_{\mathtt{res}},\mathbf{T},\mathtt{xms}_1,\mathtt{xms}_2,\mathtt{xsuff}_{\mathrm{LHS}},\mathtt{xsuff}_{\mathrm{RHS}},\mathtt{x}_{\mathrm{LHS}},\mathtt{x}_{\mathrm{RHS}})\\
\quad = \mathtt{SplitR}(\mathtt{x}_{\mathtt{res}}\,((\mathtt{xsuff}_{\mathrm{LHS}})(\mathtt{xsuff}_{\mathrm{RHS}})),\mathbf{N},(\mathbf{CONST}\ 0),(\mathbf{CONST}\ 0),\varepsilon,\varepsilon,\mathtt{x}_{\mathrm{LHS}},\mathtt{x}_{\mathrm{RHS}});\\
\mathtt{SplitR}(\mathtt{x}_{\mathtt{res}},\mathtt{s}_{\mathtt{sym}},\mathtt{xms}_1,\mathtt{xms}_2,\mathtt{xsuff}_{\mathrm{LHS}},\mathtt{xsuff}_{\mathrm{RHS}},\mathtt{x}_{\mathrm{LHS}},\mathtt{x}_{\mathrm{RHS}})\\
\quad = \mathtt{CheckLengths}(\mathtt{YieldCheck}(\mathtt{AddExprMS}(\mathtt{x}_{\mathrm{LHS}},\mathtt{xms}_1),\mathtt{AddExprMS}(\mathtt{x}_{\mathrm{RHS}},\mathtt{xms}_2)),\\
\qquad\qquad\qquad \mathtt{x}_{\mathrm{LHS}}\,\mathtt{xsuff}_{\mathrm{LHS}},\mathtt{x}_{\mathrm{RHS}}\,\mathtt{xsuff}_{\mathrm{RHS}}) {+}{+} \mathtt{x}_{\mathtt{res}};
\end{array}$$

$$\begin{array}{l}
\mathtt{CheckLengths}(\mathbf{F},\mathtt{x}_{\mathrm{LHS}},\mathtt{x}_{\mathrm{RHS}}) = (\mathtt{x}_{\mathrm{LHS}},\mathtt{x}_{\mathrm{RHS}});\\
\mathtt{CheckLengths}(\mathbf{T},\mathtt{x}_{\mathrm{LHS}},\mathtt{x}_{\mathrm{RHS}}) = (\mathbf{A},\mathbf{B});
\end{array}$$

$$\begin{array}{l}
\mathtt{SubtractMS}(\mathbf{G},\mathtt{xms}_1,\varepsilon) = \mathbf{T};\\
\mathtt{SubtractMS}(\mathbf{L},\varepsilon,\mathtt{xms}_2) = \mathbf{T};\\
\mathtt{SubtractMS}(\mathbf{G},((\mathbf{V}\mathtt{s}_{\mathtt{sym}})\ \mathtt{xnum})\,\mathtt{xms}_1,\mathtt{xms}_2) = \mathtt{CheckInfo}(\mathbf{G},\mathtt{ElMinus}(((\mathbf{V}\mathtt{s}_{\mathtt{sym}})\ \mathtt{xnum}),\varepsilon,\mathtt{xms}_2),\mathtt{xms}_1);\\
\mathtt{SubtractMS}(\mathbf{G},(\mathbf{CONST}\ \mathtt{xnum})\,\mathtt{xms}_1,\mathtt{xms}_2) = \mathtt{CheckInfo}(\mathbf{G},\mathtt{ElMinus}((\mathbf{CONST}\ \mathtt{xnum}),\varepsilon,\mathtt{xms}_2),\mathtt{xms}_1);\\
\mathtt{SubtractMS}(\mathbf{L},((\mathbf{V}\mathtt{s}_{\mathtt{sym}})\ \mathtt{xnum})\,\mathtt{xms}_1,\mathtt{xms}_2) = \mathtt{CheckInfo}(\mathbf{L},\mathtt{ElMinus}(((\mathbf{V}\mathtt{s}_{\mathtt{sym}})\ \mathtt{xnum}),\varepsilon,\mathtt{xms}_2),\mathtt{xms}_1);\\
\mathtt{SubtractMS}(\mathbf{L},(\mathbf{CONST}\ \mathtt{xnum})\,\mathtt{xms}_1,\mathtt{xms}_2) = \mathtt{CheckInfo}(\mathbf{L},\mathtt{ElMinus}((\mathbf{CONST}\ \mathtt{xnum}),\varepsilon,\mathtt{xms}_2),\mathtt{xms}_1);\\
\mathtt{SubtractMS}(\mathtt{s}_{\mathtt{sym}},\mathtt{xms}_1,\mathtt{xms}_2) = \mathbf{F};
\end{array}$$

$$\begin{array}{l}
\mathtt{CheckInfo}(\mathtt{s}_{\mathtt{sym}},\mathtt{xms}_2\,\mathtt{s}_{\mathtt{sym}},\mathtt{xms}_1) = \mathtt{SubtractMS}(\mathtt{s}_{\mathtt{sym}},\mathtt{xms}_1,\mathtt{xms}_2);\\
\mathtt{CheckInfo}(\mathbf{G},\mathtt{xms}_2\,\mathbf{L},\mathtt{xms}_1) = \mathbf{F};\\
\mathtt{CheckInfo}(\mathbf{L},\mathtt{xms}_2\,\mathbf{G},\mathtt{xms}_1) = \mathbf{F};\\
\mathtt{CheckInfo}(\mathtt{s}_{\mathtt{sym}},\mathtt{xms}_1,\mathtt{xms}_2) = \mathtt{SubtractMS}(\mathtt{s}_{\mathtt{sym}},\mathtt{xms}_1,\mathtt{xms}_2);
\end{array}$$

$$\begin{array}{l}
\mathtt{YieldCheckAux}(\varepsilon,\mathtt{xms}_1,\mathtt{xms}_2) = \mathbf{F};\\
\mathtt{YieldCheckAux}(\mathtt{s}_{\mathtt{sym}},\mathtt{xms}_1,\mathtt{xms}_2) = \mathtt{SubtractMS}(\mathtt{s}_{\mathtt{sym}},\mathtt{xms}_1,\mathtt{xms}_2);
\end{array}$$

$$\begin{array}{l}
\mathtt{YieldCheck}(\mathtt{xms}_1\,(\mathbf{CONST}\ \mathtt{xnum}_1),\mathtt{xms}_2\,(\mathbf{CONST}\ \mathtt{xnum}_2))\\
\quad = \mathtt{YieldCheckAux}(\mathtt{ElMinus}((\mathbf{CONST}\ \mathtt{xnum}_1),\varepsilon,(\mathbf{CONST}\ \mathtt{xnum}_1)),\mathtt{xms}_1,\mathtt{xms}_2);
\end{array}$$

$$\begin{array}{l}
\mathtt{AddExprMS}((\mathbf{V}\mathtt{s}_{\mathtt{sym}})\,\mathtt{x},\mathtt{xms}) = \mathtt{AddExprMS}(\mathtt{x},\mathtt{Include}((\mathbf{V}\mathtt{s}_{\mathtt{sym}}),\varepsilon,\mathtt{xms}));\\
\mathtt{AddExprMS}(\mathtt{s}\,\mathtt{x},\mathtt{xms}) = \mathtt{AddExprMS}(\mathtt{x},\mathtt{Include}(\mathtt{s},\varepsilon,\mathtt{xms}));\\
\mathtt{AddExprMS}(\varepsilon,\mathtt{xms}) = \mathtt{xms};
\end{array}$$