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 $\mathrm{WI}_{\mathscr{L}}(\langle\sigma_i\rangle, \mathscr{E}qs)$, which tests whether a composition of variable substitutions σ_i 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_i are unknown. We show that several variants of such interpreters, when specialized using the basic unfold/fold methods, are able to construct the whole solution sets for some classes of the word equations whose left- and right-hand sides share variables. We prove that the specialization process *w.r.t.* the constructed interpreters gives a simple syntactic criterion of the satisfiability of the equations considered, and show that the suggested approach can solve some equations not solvable by `Z3str3` and `CVC4`, the widely-used SMT-solvers.

1 Introduction

In recent decades, program transformation techniques were applied to verification and analysis of several computational models, including cache-coherence and cryptographic protocols, constrained Horn clauses, Petri nets, deductive databases, control-flow analysis, *etc.* [2, 10, 11, 13, 23, 29, 40]. On the other hand, the number of works on analysis of string manipulating programs and string constraint solvers is rapidly growing during last years [1, 4, 6, 8, 19, 20, 21, 24, 32, 36, 41]. As far as we know, there are few interactions between the two research areas, although some of their methods exploit similar concepts.

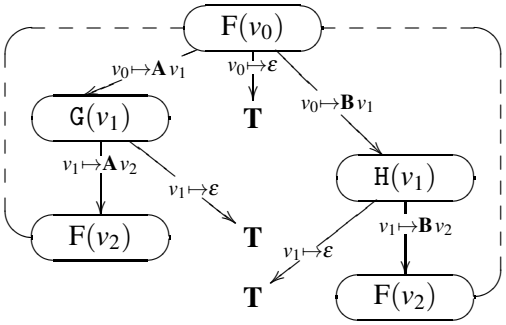
One approach to the verification is to apply an unfold/fold algorithm [5] to model a nondeterministic system behaviour by a deterministic program via introducing an additional path parameter [23, 22]. That is, given a specialization algorithm `Spec` and a non-deterministic program $f(\mathtt{x})$, the 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 [26], Hmelevskij's [14], Makanin's [25] algorithms in 1970s, and including Plandowski's [31] and Jez's algorithms [15] designed in the recent two decades, all use the nondeterministic search. A word equation is an equation $\Phi = \Psi$, where Φ and Ψ are finite words in the joint alphabet $\mathscr{A} \cup \mathscr{V}$ of letters and variables, its solution is a substitution $\sigma : \mathscr{V} \to \mathscr{A}^*$ *s.t.* $\Phi\sigma$ is textually equal to $\Psi\sigma$. All 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

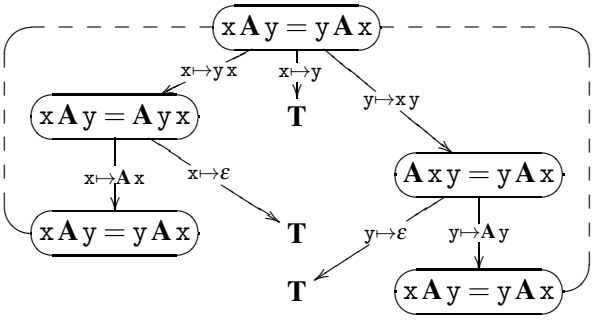
*The reported study was partially supported by Russian Academy of Sciences, research project No. AAAA-A19-119020690043-9.

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

Alexei Lisitsa and Andrei Nemytykh (Eds.): 9th International Workshop
on Verification and Program Transformation (VPT 2021)
EPTCS 341, 2021, pp. 42–72, doi:10.4204/EPTCS.341.4



(a) Program transformation



(b) Solving a word equation

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

paths of the solution tree may be infinite, and are to be either pruned or represented as loops. For example, given 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 labelling the nodes coincide. Figure 1 shows a solution graph for a simple word equation generated 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 [30, 34]. One may observe that the two graphs coincide modulo the node and arc labels, sharing the general structure, although they are constructed for the different purposes. This paper focuses on the similarity of the methods and aims at adjusting the unfold/fold program transformation algorithm to solving the word equations.

Our contributions are the following.

1. We study three interpreters, specialization of which using a general-purpose tool *w.r.t.* a given word equation constructs a residual program presenting the complete solutions set of this word equation.
2. We prove that an analogue of Jones-optimality holds for the interpreters considered [3, 17], which guarantees that if the specialization terminates, then the residual programs represent complete solution sets of the given equation systems. 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].
3. We also reveal several other classes of equations sharing variables in right- and left-hand sides, for which the specialization is proved to always terminate. To the best of our knowledge, these classes of equations were not covered by the published works on the string constraint solvers applied to the unbounded-length case.

4. Finally, we show the application results of the presented approach to the benchmark equation sets developed 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 tests considered. Our algorithm is slow, as compared to the algorithms implemented in the SMT-solvers, when used on satisfiable equations because the algorithm finds all the solutions instead of at least one, however for the equations having no solution, our algorithm shows better success rate.

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[2] 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 properties of the presented algorithms are given in Appendix, as well as the source code of the interpreter models used in the specialization.

We assume that the reader is familiar with the basic notions of the program specialization, *s.t.* partial evaluation, partial deduction, supercompilation, *etc.*

2 Preliminaries

We denote the set of the string variables with $\mathscr{V}$, the finite set of the letters with $\mathscr{A}$. We assume that the bold capitals $\mathbf{A}$, $\mathbf{B}$ denote the letters in $\mathscr{A}$; while the typographic small letters $\mathtt{x}$, $\mathtt{y}$, $\mathtt{z}$ denote the variables in $\mathscr{V}$. 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 $\xi\colon \mathscr{V} \to \{\mathscr{A} \cup \mathscr{V}\}^*$ to a word Φ as $\Phi\xi$. A solution of the equation $\Phi = \Psi$ is a substitution $\xi\colon \mathscr{V} \to \mathscr{A}^*$ *s.t.* $\Phi\xi$ and $\Psi\xi$ coincide textually. Given an equation $E\colon \Phi = \Psi$ 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 Φ with $|\Phi|_t$. The equation length is $|\Phi| + |\Psi|$, ε denotes the empty word.

2.1 Simple Logic Language

Our method is based on an analysis of programs written in a simple logic language $\mathscr{L}$ over the dataset consisting of the word equations. An $\mathscr{L}$ program is a list of narrowings representing the variable substitutions that are to be applied to the word equations. We distinguish between the narrowings in the language $\mathscr{L}$ and the narrowings occurring in the specialization process (Sec. 5). Hence, we call the former $\mathscr{L}$-narrowings, and the latter — parameter narrowings (or par-narrowings).

A word equation $\Phi = \Psi$ is encoded with a pair (Φ, Ψ). An $\mathscr{L}$-narrowing is encoded with a string using the sign $\mapsto$. There are three possible forms of the elementary $\mathscr{L}$-narrowings, corresponding to those used in the Matiyasevich algorithm. We consider these $\mathscr{L}$-narrowing sequences as the programs in a simple acyclic logic programming language $\mathscr{L}$. Thus, every statement in an $\mathscr{L}$-program is an encoded $\mathscr{L}$-narrowing. The syntax of the encoded equations `Eqs` and of the $\mathscr{L}$-programs `Narrs` is given in Figure 2. Here Var, $Var_1 \in \mathscr{V}$, $Var \neq Var_1$, $Char \in \mathscr{A}$. Let $\langle \Phi_i = \Psi_i \rangle_{i=1}^n$ be syntactic sugar for $(\Phi_1, \Psi_1) \ldots (\Phi_n, \Psi_n)$. The $\mathscr{L}$-narrowings sequence $(\sigma_1) \ldots (\sigma_m)$ is also written as $\langle \sigma_i \rangle_{i=1}^m$.

[2] We do not consider the generation of a residual program, since the correctness of the residual programs is provided by the properties of the process graph.

Data	Programs
$Eqs ::= Eq\ Eqs \mid \varepsilon$ $Eq ::= (Side, Side)$ $Side ::= Char\ Side \mid Var\ Side \mid \varepsilon$	$Narrs ::= (Narr)\ Narrs \mid \varepsilon$ $Narr ::= \texttt{'}Var \mapsto Char\ Var\texttt{'} \mid \texttt{'}Var \mapsto Var_1\ Var\texttt{'} \mid \texttt{'}Var \mapsto \varepsilon\texttt{'}$

Figure 2: The syntax of the data and the programs.

2.2 Interpreters' Source Pseudocode Language

The interpreters considered are written in the following pseudocode for functional programs manipulating the strings and based on the pattern matching. The $\mathscr{F}$ programs are lists of term rewriting rules. The rules in the definitions are applied using the top-down matching order. The syntax of the language $\mathscr{F}$ is given in Figure 3. Here ε is the empty word, ++ stands for the associative concatenation constructor (both may be omitted). The set of the constants used as the letters[3] is Σ, elements of which are given in bold, $\mathscr{A} \subset \Sigma$. The variables in the $\mathscr{F}$-program rules range either over expressions or over letters. Henceforth we call these variables $\mathscr{F}$-variables or pattern variables, in order to distinguish them from the variables occurring in the word equations.

$$
\begin{array}{rcl}
Rule & ::= & FName(Pattern, \ldots, Pattern) = Exp \\
Pattern & ::= & \varepsilon \mid Variable \mid Letter \mid (Pattern) \mid Pattern\ ++\ Pattern \\
Exp & ::= & \varepsilon \mid Variable \mid Letter \mid (Exp) \mid FName(Exp, \ldots, Exp) \mid Exp\ ++\ Exp \\
Variable & ::= & \texttt{x}Name \mid \texttt{s}Name
\end{array}
$$

Figure 3: The syntax of the program pseudocode.

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

3 Word Equations' Interpreters

In this section we introduce informally a class of simple interpreters taking $\mathscr{L}$-programs and applying them to the lists of word equations. Given such an interpreter $\mathrm{WI}_{\mathscr{L}}$, a program $\langle\sigma_i\rangle_{i=1}^m$, and a sequence of equations $\langle\Phi_i = \Psi_i\rangle_{i=1}^n$, the call $\mathrm{WI}_{\mathscr{L}}(\langle\sigma_i\rangle_{i=1}^m, \langle\Phi_i = \Psi_i\rangle_{i=1}^n)$ returns **T** iff the following two conditions hold. The notions of the compatibility of an $\mathscr{L}$-narrowing with an equation list and of the operation `Simplify` are given further.

- Every $\mathscr{L}$-narrowing σ_i, where $1 \leq i \leq m$, is compatible with the equation list resulting from the call $\texttt{Simplify}(\langle\Phi_i\sigma_1\ldots\sigma_{i-1} = \Psi_i\sigma_1\ldots\sigma_{i-1}\rangle_{i=1}^n)$.
- And for every i, $1 \leq i \leq n$, $\Phi_i\sigma_1\ldots\sigma_m$ textually coincides with $\Psi_i\sigma_1\ldots\sigma_m$.

The call $\mathrm{WI}_{\mathscr{L}}(\varepsilon, \varepsilon)$ returns **T**. Otherwise, the call $\mathrm{WI}_{\mathscr{L}}(\langle\sigma_i\rangle_{i=1}^m, \langle\Phi_i = \Psi_i\rangle_{i=1}^n)$ returns **F**.

All the $\mathscr{L}$-interpreters share the same structure: they take the first program statement, apply it to the equation list, and then call a simplification function `Simplify` that transforms the current resulting

[3] 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.

equation list to an equation list with the same solution set, and having a simpler form (see Figure 4). The function `Simplify` is the only source part that depends on the concrete interpreter considered.

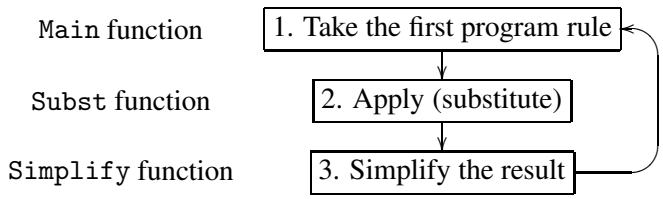
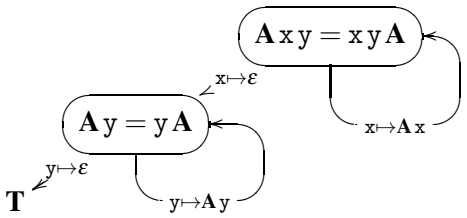


Figure 4: General structure of the interpreters.

We can also say that such an interpreter applies a list of the $\mathscr{L}$-narrowings to an equation, choosing for this purpose a path to one its solution. The solution tree is a tree describing the set of all the solution paths, the tree nodes are labelled with equation lists[4]. Such a tree can be constructed as a (possibly infinite) directed graph representing a non-deterministic unfolding process using the substitutions listed in Figure 5 (a). We see the $\mathscr{L}$-narrowings in the first column. Given an $\mathscr{L}$-narrowing, the second column provides the constraint imposed on the first equation in the list, required for applying this narrowing. Thus, an $\mathscr{L}$-narrowing σ is compatible with the list $\langle \Phi_i = \Psi_i \rangle_{i=1}^{n}$ iff σ is generated by the constraint imposed on the first equation in the list $\Phi_1 = \Psi_1$ given in Figure 5 (a). Following the classical approach [7, 14, 26], no fresh variables and constants are introduced in the $\mathscr{L}$-narrowing rules.

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

(a) The $\mathscr{L}$-narrowings and equations compatible with them.



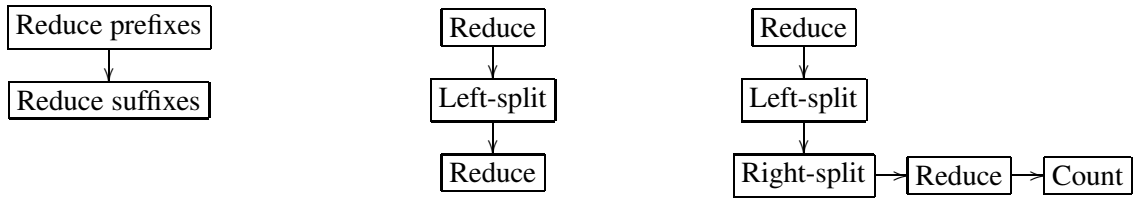
(b) The solution graph of $\mathbf{A}\,\mathtt{x}\,\mathtt{y} = \mathtt{x}\,\mathtt{y}\,\mathbf{A}$.

Figure 5: The narrowings cases and an example of a solution graph.

We use a slightly modified version of Matiyasevich's algorithm. The Nielsen transformation [9], which is the base of the algorithm, states that given $\mathtt{x}\Phi = \mathtt{y}\Psi$, we can replace either $\mathtt{x}$ with $\mathtt{y}\,\mathtt{x}$ if the length of the value of the variable $\mathtt{x}$ is greater than the length of the value of $\mathtt{y}$, or vice versa if the length of $\mathtt{y}$ value is greater, or $\mathtt{x}$ with $\mathtt{y}$ if their value lengths are equal. Figure 5 (a) does not show the last case: it is a composition of the substitutions $\mathtt{x} \mapsto \mathtt{y}\,\mathtt{x}$ and $\mathtt{x} \mapsto \varepsilon$, since we allow the substitution $\mathtt{x} \mapsto \varepsilon$ to be compatible with any equation whose left- or right-hand side starts with $\mathtt{x}$. This modification guarantees that any solution to the equation $\Phi = \Psi$ can be generated[5] as a composition of the elementary $\mathscr{L}$-narrowings given in Figure 5 (a). The following example shows that for the classical version of the algorithm, this statement does not hold.

[4] 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 ...".

[5] 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.



(a) Interpreter WIBase$_{\mathscr{L}}$ Scheme AlgWE$_{\text{Base}}$ (b) Interpreter WISplit$_{\mathscr{L}}$ Scheme AlgWE$_{\text{Split}}$ (c) Interpreter WICount$_{\mathscr{L}}$ Scheme AlgWE$_{\text{Count}}$

Figure 6: Schemes for simplifying the word equations.

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

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 labels 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 with a cycle, thus the tree is represented with the graph. Henceforth we use almost interchangeably these two notions. An example of a graph representing all solutions of the equation $\mathbf{A}\,\mathtt{xy} = \mathtt{xy}\mathbf{A}$ is given in Figure 5 (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.

An interpreter WI$_{\mathscr{L}}(\mathsf{P}, \langle E_1 \ldots, E_n \rangle)$ takes a list of the equations E_i and a logic program P being a list of the $\mathscr{L}$-narrowings that have to be successively applied to the equations. If the composition of the substitutions given in P transforms all the equations in the list to the tautologies, then WI$_{\mathscr{L}}(\mathsf{P}, \langle E_1 \ldots, E_n \rangle)$ returns the value **T**. If a substitution is not compatible with the current equations list or the list of substitutions is empty whereas the equations are not tautologies then WI$_{\mathscr{L}}(\mathsf{P}, \langle E_1 \ldots, E_n \rangle)$ results in the value **F**. Given any input equation list, the interpreter does at most m steps shown in Figure 4, where m is the number of the $\mathscr{L}$-narrowings in the list P, and always returns either value **T** or value **F**.

The general structure of the simplification functions used in the three interpreters that we consider in this paper is given in Figure 6. The transformations shown in this figure are also used for constructing the corresponding solution graphs. According to Figure 6 we say that the interpreter WIBase$_{\mathscr{L}}$ models paths in a solution tree based on the scheme AlgWE$_{\text{Base}}$; and so on. See Figure 7 for examples of the corresponding solution graphs.

3.1 Basic Interpreter

The basic interpreter WIBase$_{\mathscr{L}}(\mathsf{P}, \Phi = \Psi)$ manipulates a single equation and accepts as the first input a list P of the elementary $\mathscr{L}$-narrowings. When an $\mathscr{L}$-narrowing σ from the list P is applied to $\Phi = \Psi$, the simplification function immediately removes equal prefixes and suffixes of $\Phi\sigma$ and $\Psi\sigma$. Thus, the function `Simplify` constructs the reduced form of the equation $(\Phi = \Psi)\sigma$. Actually, this basic interpreter models the classic algorithm for solving the word equations suggested by Matiyasevich. We treat the interpreter WIBase$_{\mathscr{L}}$ as a base for developing more complex ones.

3.2 Splitting Interpreter

The interpreter WISplit$_{\mathscr{L}}(\mathsf{P}, \langle E_1, \ldots, E_n \rangle)$ manipulates the lists of equations (representing equation systems) rather than a single equation. Thus, the substitution and simplification functions are applied to

every equation in the list. If construction of the reduced form of some equation in the list 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. The operation looking for trivial contradictions is also a part of the algorithm deriving the reduced form.

A natural way to manually simplify an equation is to split it using the length argument [7]. *E.g.* given equation $\mathtt{x}\mathbf{A}\,\mathtt{y}\mathbf{B} = \mathbf{A}\,\mathtt{x}\,\mathtt{x}\,\mathtt{x}$ we can split it into the list of $E_1 : \mathtt{x}\mathbf{A} = \mathbf{A}\,\mathtt{x}$ and $E_2 : \mathtt{y} = \mathtt{x}\,\mathtt{x}$, and the system represented by the list of these two equations has the same solution set as the initial equation.

Let us describe more formally the mentioned method. 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* $\mathrm{Pr}_i, \mathrm{S}_i \in \{\mathscr{A} \cup \mathscr{V}\}^*$*. Then the equation is equivalent to the system* $\mathrm{Pr}_1 = \mathrm{Pr}_2$ & $\mathrm{S}_1 = \mathrm{S}_2$ *if at least one of these two statements holds.*

1. *the prefixes* Pr_1 *and* Pr_2 *are non-empty and var-permutated;*
2. *the suffixes* S_1 *and* S_2 *are non-empty and var-permutated.*

If Proposition 1 is applied to the prefixes of an equation we say that the equation is left-split; if it is applied to the suffixes we say the equation is right-split.

The simplification function of the interpreter WISplit$_{\mathscr{L}}$ uses Proposition 1 *w.r.t.* the var-permutated prefixes. Given an equation list $\langle E_1, \ldots, E_n \rangle$, let the substitution σ be applied. Then the function `Simplify` first reduces all the equations in the list $\langle E_1\sigma, \ldots, E_n\sigma \rangle$. For every resulting equation E'_i, the simplification algorithm 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. If that succeeds, the equation E'_i is split into the two equations $\mathrm{Pr}_{1,i,1} = \mathrm{Pr}_{2,i,1}$ and $\mathrm{S}_{1,i,1} = \mathrm{S}_{2,i,1}$. Here the third index is the number of the splitting iterations. Then the simplification algorithm construct the reduced form of 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 (see Figure 6 (b)). The initial equation E'_i is replaced with the generated equations resulting from k successful left-split operations. The equations in the updated list are ordered as follows: $\mathrm{S}_{1,i,k} = \mathrm{S}_{2,i,k}, \mathrm{Pr}_{1,i,1} = \mathrm{Pr}_{2,i,1}, \ldots, \mathrm{Pr}_{1,i,k} = \mathrm{Pr}_{2,i,k}$. The simplification function used in WISplit$_{\mathscr{L}}$ does not change the resulting equations $\mathrm{Pr}_{1,i,k} = \mathrm{Pr}_{2,i,k}$, since they are in the reduced form by the construction.

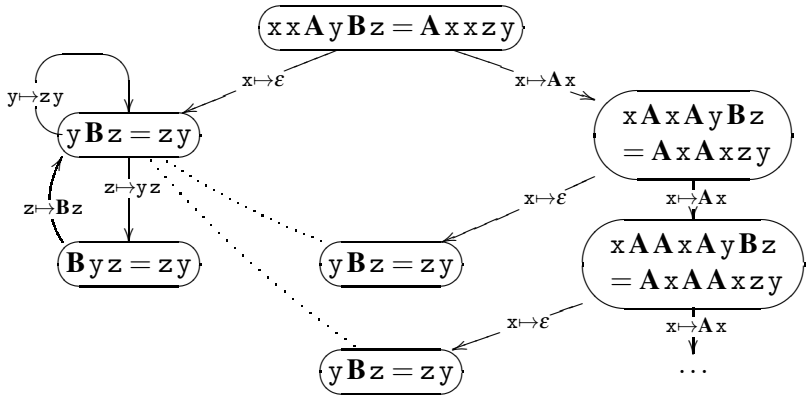
Example 2. *Given* $\sigma : \mathtt{x} \mapsto \varepsilon$ *and the list* $\langle E_1, E_2 \rangle = \langle \mathtt{x}\,\mathtt{z}\,\mathtt{x}\mathbf{B}\,\mathtt{y} = \mathbf{A}\,\mathtt{z}\,\mathtt{z},\ \mathtt{y}\,\mathtt{x}\mathbf{B}\,\mathtt{z}\,\mathtt{y} = \mathbf{A}\,\mathtt{y}\,\mathtt{z}\,\mathtt{z}\,\mathtt{z} \rangle$*, the equation* $E_1\sigma$ *is* $\mathtt{z}\mathbf{B}\,\mathtt{y} = \mathbf{A}\,\mathtt{z}\,\mathtt{z}$*, thus it is split into the equations* $\mathtt{z}\mathbf{B} = \mathbf{A}\,\mathtt{z}$ *and* $\mathtt{y} = \mathtt{z}$*, which replace* $E_1\sigma$ *in the order* $\langle \mathtt{y} = \mathtt{z}, \mathtt{z}\mathbf{B} = \mathbf{A}\,\mathtt{z} \rangle$*, where the non-var-permutated suffixes are written first. The equation* $E_2\sigma : \mathtt{y}\mathbf{B}\,\mathtt{z}\,\mathtt{y} = \mathbf{A}\,\mathtt{y}\,\mathtt{z}\,\mathtt{z}\,\mathtt{z}$ *is first split into the equations:* $\mathtt{y}\mathbf{B} = \mathbf{A}\,\mathtt{y}$ *and* $\mathtt{z}\,\mathtt{y} = \mathtt{z}\,\mathtt{z}\,\mathtt{z}$*. The second equation is then transformed to the reduced form* $\mathtt{y} = \mathtt{z}\,\mathtt{z}$ *and cannot be split any more. These two equations replace the equation* $E_2\sigma$ *in the following order:* $\langle \mathtt{y} = \mathtt{z}\,\mathtt{z}, \mathtt{y}\mathbf{B} = \mathbf{A}\,\mathtt{y} \rangle$*. The resulted list of the simplified equations is* $\langle \mathtt{y} = \mathtt{z}, \mathtt{z}\mathbf{B} = \mathbf{A}\,\mathtt{z}, \mathtt{y} = \mathtt{z}\,\mathtt{z}, \mathtt{y}\mathbf{B} = \mathbf{A}\,\mathtt{y} \rangle$.

3.3 Counting Interpreter

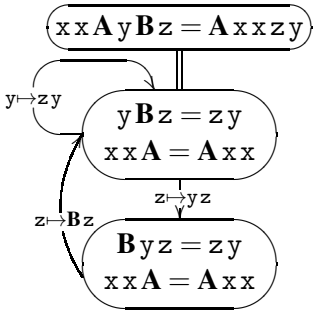
The third variant of our interpreter uses the following well-known simple observation.

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

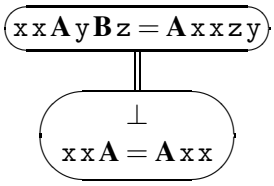
After constructing the reduced form of the equations, the simplification function of the interpreter WICount$_{\mathscr{L}}$ tries to construct their left-splits (as WISplit$_{\mathscr{L}}$ does), and then to construct the right-splits of their suffixes resulted from the left-splits. Finally, WICount$_{\mathscr{L}}$ checks the property stated in Proposition 2



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



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



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

Figure 7: Solution graphs for $\mathtt{x}\,\mathtt{x}\mathbf{A}\,\mathtt{y}\mathbf{B}\,\mathtt{z} = \mathbf{A}\,\mathtt{x}\,\mathtt{x}\,\mathtt{z}\,\mathtt{y}$.

of the resulting equations with non-var-permutated sides. The interpreter $\mathrm{WICount}_{\mathscr{L}}$ has been used in the most benchmark tests (Section 6).

Figure 7 demonstrates the difference between the simplification algorithms used in the presented interpreters. The dotted edges in the graph constructed using the algorithm $\mathrm{AlgWE}_{\mathrm{Base}}$ show the equality of the node labels; but the paths are not folded here, because the equal nodes are not along the same path. The edges given in the double lines show the splits. The sign $\bot$ denotes the contradiction. Every solution of the equation corresponds to a non-empty set[6] of paths rooted in the initial node of its solution tree and ending at a leaf labelled by **T**. Thus, if the solution graph of the equation does not have **T**-leaves, then the equation solution set is empty. The graph constructed using the algorithm $\mathrm{AlgWE}_{\mathrm{Base}}$ is infinite, whereas the other two graphs show that the equation $\mathtt{x}\,\mathtt{x}\mathbf{A}\,\mathtt{y}\mathbf{B}\,\mathtt{z} = \mathbf{A}\,\mathtt{x}\,\mathtt{x}\,\mathtt{z}\,\mathtt{y}$ has no solution.

[6]The set may be infinite, for example the solution $\mathtt{x} = \varepsilon$ & $\mathtt{y} = \varepsilon$ of the equation $\mathtt{x}\,\mathtt{y} = \mathtt{y}\,\mathtt{x}$ may be a result of the composition of the elementary substitutions $\mathtt{x} \mapsto \mathtt{y}\,\mathtt{x}$, $\mathtt{x} \mapsto \varepsilon$, $\mathtt{y} \mapsto \varepsilon$; $\mathtt{x} \mapsto \mathtt{y}\,\mathtt{x}$, $\mathtt{x} \mapsto \mathtt{y}\,\mathtt{x}$, $\mathtt{x} \mapsto \varepsilon$, $\mathtt{y} \mapsto \varepsilon$; *etc.* In such a case, the solutions are always resulting from concatenations not increasing these solutions' lengths.

4 Verification Task

We use the notion of a parameter (*i.e.* 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. The parameter values are used in this paper in order to represent possible paths in the solution tree. Thus, if the $\mathscr{L}$-program P in $\mathrm{WI}_{\mathscr{L}}(\mathsf{P}, \langle E_1 \ldots, E_n \rangle)$ is replaced with a parameter, then the stepwise unfolding of this call generates all the possible programs (*i.e.* the $\mathscr{L}$-narrowings' lists) that are compatible with the equation list. The unfolding stops when either the equation list is empty or no $\mathscr{L}$-narrowing compatible with the current equation list is found. Henceforth the letters u and v, maybe subscripted, stand for the parameters.

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

$$\mathfrak{M}(\mathrm{WI}_{\mathscr{L}}, \mathscr{E}qs) \triangleq \mathtt{Spec}\Big(\underline{\mathrm{WI}_{\mathscr{L}}}, \underline{\mathtt{Go}(v, \llcorner \mathscr{E}qs \lrcorner)}\Big),$$

where `Go` is the name of the entry function of $\mathrm{WI}_{\mathscr{L}}$. Here v ranges over the set of the encoded $\mathscr{L}$-programs (Figure 8) that can be interpreted by $\mathrm{WI}_{\mathscr{L}}$, namely all possible encoded sequences of the $\mathscr{L}$-narrowings. The lengths of the $\mathscr{L}$-programs are unbounded, so there is no bound on the solution lengths.

$$\begin{array}{rcl}
\llcorner \text{'LHS = RHS'} \lrcorner & = & (\llcorner \text{LHS} \lrcorner,\ \llcorner \text{RHS} \lrcorner) \\
\llcorner \text{'x} \mapsto \text{Narr'} \lrcorner & = & \llcorner \text{x} \lrcorner \mapsto \llcorner \text{Narr} \lrcorner \\
\llcorner \text{Term} +\!\!+ \text{Expr} \lrcorner & = & \llcorner \text{Term} \lrcorner +\!\!+ \llcorner \text{Expr} \lrcorner \\
\llcorner \text{Letter} \lrcorner & = & \text{Letter} \\
\llcorner \text{Variable} \lrcorner & = & (\mathbf{V}\ \text{Variable})
\end{array}$$

Figure 8: The encoding used for the data in the interpreters.

The result of this specialization is a program with the input value to be assigned to v. We impose the following minimal requirement on the specialization, which is strengthened in Sec. 6: the specialization succeeds if the resulting process tree generated by `Spec` contains a leaf labelled with the value **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 $\mathrm{WI}_{\mathscr{L}}$. However, we do not require the specialization task to terminate on every equation list. Thus, the power of the suggested verification scheme depends on the underlying interpreter $\mathrm{WI}_{\mathscr{L}}$.

5 Unfold/Fold Program Transformation Method

The specialization tool `Spec` used in the verification scheme above is based on the elementary unfold/fold technique widely used, *e.g.* , in deforestation, supercompilation, partial evaluation, partial deduction, and so on [5, 17, 30, 33]. The algorithm transforms the $\mathscr{F}$-programs (Sec. 2.2). The technique exploits the sub-algorithms presented briefly in this Section and more formally in the paper [23]. The unfold/fold algorithm assumes that every node N in the process tree $\mathscr{T}$ of the $\mathscr{F}$-program is labelled with a configuration, which represents the current parameterized computation state.

Definition 1. *A configuration C is a parameterized expression in the language $\mathscr{F}$. Namely, it is either a parameter, a string in Σ^*, a parameterized expression enclosed in the parentheses, a concatenation of parameterized expressions or a function call with the parameterized expressions as its arguments.*

The active call of the configuration C is the function call (if any) with the leftmost closing right bracket.

We say an $\mathscr{F}$-expression is *ground* if it does not contain function calls, while it may contain parameter occurrences. The set of the parameters is denoted with $\mathscr{P}ar$, and $\mathscr{G}rd$ stands for the set of the ground expressions. We say that a call $\mathrm{F}(\mathrm{Expr}_1,\ldots,\mathrm{Expr}_\mathrm{n})$ matches against $\mathrm{F}(\mathrm{P}_1,\ldots,\mathrm{P}_n)$, where Expr_i are object expressions, and P_i are patterns, if there exists a substitution ξ *s.t.* $\forall i, 1 \leq i \leq n(\mathrm{P}_i\xi = \mathrm{Expr}_\mathrm{i})$.

Definition 2. *Given a program rule* $R : \mathrm{F}(\mathrm{P}_1,\ldots,\mathrm{P}_n) = \mathrm{S}$*, let C be of the form* $\mathrm{F}(\mathrm{Expr}_1,\ldots,\mathrm{Expr}_\mathrm{n})$*, where* Expr_i *are ground*[7]*. We say that the substitution* $\xi : \mathscr{P}ar \to \mathscr{G}rd$ *is* a parameter narrowing *unifying R with C iff* $\exists\sigma : \mathscr{V}ar(\langle \mathrm{P}_1,\ldots,\mathrm{P}_n\rangle) \to \mathscr{G}rd$ *s.t.* $\forall i\,(\mathrm{P}_i\sigma = \mathrm{Expr}_\mathrm{i}\xi)$*. We say that the set of the par-narrowings* $\{\xi_i\}$ *is* exhaustive *w.r.t. R if for every substitution* $\tau : \mathscr{P}ar \to \Sigma^*$ *s.t. the expression* $C\tau$ *matches against the left-hand side of the rule R, there exists a par-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 $\mathscr{T}$ is marked either as open (by default), or as closed with some node N'. The three steps listed below are applied to the tree $\mathscr{T}$ until all the nodes in $\mathscr{T}$ are closed.

- **Unfolding step**. Given an open node N labelled with a parameterized configuration C, consider the active call $\mathrm{F}(\mathrm{Expr}_1,\ldots,\mathrm{Expr}_\mathrm{n})$ in C. For every rule $R_i : \mathrm{F}(\mathrm{P}_{i,1},\ldots,\mathrm{P}_{i,n}) = \mathrm{S}_i$ in the definition of F (where $\mathrm{P}_{i,j}$ are patterns), construct a set of pairs $\langle\sigma_{i,k},\xi_{i,k}\rangle$ *s.t.* $\forall j\,(\mathrm{P}_{i,j}\sigma_{i,j} = \mathrm{Expr}_\mathrm{j}\xi_\mathrm{i,j})$, and the set $\{\xi_{i,k}\}$ of the parameter narrowings is exhaustive *w.r.t.* R_i. For every such a par-narrowing $\xi_{i,k}$ generate an open child node $\mathsf{N}_{i,k}$. Construct $C\xi_{i,k}$, and replace the active call $\mathrm{F}(\mathrm{Expr}_1,\ldots,\mathrm{Expr}_\mathrm{n})\xi_\mathrm{i,k}$ in it with $\mathrm{S}_\mathrm{i}\sigma_{i,k}$. The result is the configuration[8] labelling the node $\mathsf{N}_{i,k}$.
- **Folding step**. Given a node N labelled with a configuration C, if some its ancestor N_0 is labelled with C (up to a parameter renaming), then mark N as closed with N_0 and remove all the 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.

In order to guarantee that the unification algorithm used in the unfolding step can always produce a finite set of the parameter narrowings, we use the following syntactic property of the function `Main` of the interpreters considered. Figure 9 presents the source code of the function `Main` in the interpreter $\mathrm{WIBase}_{\mathscr{L}}$. The other interpreters use this function with some minor changes, such as applying the substitution function to the equation lists and storing information about the number of the equations in the list in the second argument of `Main`. The patterns used for the first argument of `Main` in the left-hand sides of the definitions are the same in all the three interpreters considered.

Property 1. *Given the interpreters* $\mathrm{WIBase}_{\mathscr{L}}$*,* $\mathrm{WISplit}_{\mathscr{L}}$*, and* $\mathrm{WICount}_{\mathscr{L}}$*, the program rewriting rules defining the function* `Main` *in the interpreters are only of the following forms:*

- $\texttt{Main}(\mathrm{P}, S_1) = S_2$*, where* S_2 *is an object expression (rules (1) and (5) in Figure 9);*
- $\texttt{Main}(\mathrm{P} +\!\!+ \texttt{x}_\texttt{rul}, S_1) = \texttt{Main}(\texttt{x}_\texttt{rul}, S_2)$*, where* $(\mathscr{V}ar(S_2) \setminus \mathscr{V}ar(S_1)) \cap \mathscr{V}ar(\mathrm{P}) = \varnothing$*, the part* P *does not contain expression-type variables, and* $\texttt{x}_\texttt{rul} \notin \mathscr{V}ar(\mathrm{S}_2)$ *(rules (2–4 a,b)).*

We recall that the verification task is $\texttt{Spec}\Big(\underline{\mathrm{WI}_{\mathscr{L}}}, \underline{\texttt{Go}(v, \underline{\mathscr{E}qs_\mathrm{j}})}\Big)$, and the rules of the function `Go` of all the three interpreters are $\texttt{Go}(\texttt{x}_\texttt{rul}, \texttt{x}_\texttt{val}) = \texttt{Main}(\texttt{x}_\texttt{rul}, \texttt{Simplify}(\textit{Other args}))$, where $\texttt{x}_\texttt{rul}$ does not occur in the other arguments. This fact together with Property 1 imply the following feature.

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

[8] In the case of the verification task considered, Property 2 implies that the par-narrowing $\xi_{i,k}$ is applied to the only active call, since the arguments of the other calls do not include parameters.

$/*\,(1)$ There are no more $\mathscr{L}-$narrowings, and the equation is trivial. $*/$
$\mathtt{Main}(\varepsilon,(\varepsilon,\varepsilon)) = \mathbf{T};$
$/*\,(2\,\mathrm{a,b})$ The $\mathscr{L}-$narrowing $\mathtt{'s}_x \mapsto \varepsilon\mathtt{'}$ is compatible with an equation whose side starts with a variable named $\mathtt{s}_x$. $*/$
$\mathtt{Main}(((\mathbf{Vs}_x) \mapsto \varepsilon) \mathtt{+\!\!+} \mathtt{x}_{\mathtt{rul}}, (\mathtt{x}_{\mathrm{LHS}}, (\mathbf{Vs}_x) \mathtt{+\!\!+} \mathtt{x}_{\mathrm{RHS}}))$
$= \mathtt{Main}(\mathtt{x}_{\mathtt{rul}}, \mathtt{Simplify}(((\mathbf{Vs}_x) \mapsto \varepsilon), \mathtt{Subst}((\mathbf{Vs}_x) \mapsto \varepsilon, \varepsilon, \mathtt{x}_{\mathrm{LHS}}), \mathtt{Subst}((\mathbf{Vs}_x) \mapsto \varepsilon, \varepsilon, \mathtt{x}_{\mathrm{RHS}})));$
$\mathtt{Main}(((\mathbf{Vs}_x) \mapsto \varepsilon) \mathtt{+\!\!+} \mathtt{x}_{\mathtt{rul}}, ((\mathbf{Vs}_x) \mathtt{+\!\!+} \mathtt{x}_{\mathrm{LHS}}, \mathtt{x}_{\mathrm{RHS}}))$
$= \mathtt{Main}(\mathtt{x}_{\mathtt{rul}}, \mathtt{Simplify}(((\mathbf{Vs}_x) \mapsto \varepsilon), \mathtt{Subst}((\mathbf{Vs}_x) \mapsto \varepsilon, \varepsilon, \mathtt{x}_{\mathrm{LHS}}), \mathtt{Subst}((\mathbf{Vs}_x) \mapsto \varepsilon, \varepsilon, \mathtt{x}_{\mathrm{RHS}})));$
$/*\,(3\,\mathrm{a,b})$ The $\mathscr{L}-$narrowing $\mathtt{'s}_x \mapsto \mathtt{s}_{\mathtt{sym}}\mathtt{s}_x\mathtt{'}$ is compatible with an equation having one side starting with the equation variable named $\mathtt{s}_x$ and the other side starting with a symbol. $*/$
$\mathtt{Main}(((\mathbf{Vs}_x) \mapsto \mathtt{s}_{\mathtt{sym}}(\mathbf{Vs}_x)) \mathtt{+\!\!+} \mathtt{x}_{\mathtt{rul}}, ((\mathbf{Vs}_x) \mathtt{+\!\!+} \mathtt{x}_{\mathrm{LHS}}, \mathtt{s}_{\mathtt{sym}} \mathtt{+\!\!+} \mathtt{x}_{\mathrm{RHS}}))$
$= \mathtt{Main}(\mathtt{x}_{\mathtt{rul}}, \mathtt{Simplify}(((\mathbf{Vs}_x) \mapsto \mathtt{s}_{\mathtt{sym}}(\mathbf{Vs}_x)), \mathtt{Subst}((\mathbf{Vs}_x) \mapsto \mathtt{s}_{\mathtt{sym}}(\mathbf{Vs}_x), (\mathbf{Vs}_x), \mathtt{x}_{\mathrm{LHS}}),$
$\mathtt{Subst}((\mathbf{Vs}_x) \mapsto \mathtt{s}_{\mathtt{sym}}(\mathbf{Vs}_x), \varepsilon, \mathtt{x}_{\mathrm{RHS}})));$
$\mathtt{Main}(((\mathbf{Vs}_x) \mapsto \mathtt{s}_{\mathtt{sym}}(\mathbf{Vs}_x)) \mathtt{+\!\!+} \mathtt{x}_{\mathtt{rul}}, (\mathtt{s}_{\mathtt{sym}} \mathtt{+\!\!+} \mathtt{x}_{\mathrm{LHS}}, (\mathbf{Vs}_x) \mathtt{+\!\!+} \mathtt{x}_{\mathrm{RHS}}))$
$= \mathtt{Main}(\mathtt{x}_{\mathtt{rul}}, \mathtt{Simplify}(((\mathbf{Vs}_x) \mapsto \mathtt{s}_{\mathtt{sym}} \mathtt{+\!\!+} (\mathbf{Vs}_x)), \mathtt{Subst}((\mathbf{Vs}_x) \mapsto \mathtt{s}_{\mathtt{sym}}(\mathbf{Vs}_x), \varepsilon, \mathtt{x}_{\mathrm{LHS}}),$
$\mathtt{Subst}((\mathbf{Vs}_x) \mapsto \mathtt{s}_{\mathtt{sym}}(\mathbf{Vs}_x), (\mathbf{Vs}_x), \mathtt{x}_{\mathrm{RHS}})));$
$/*\,(4\,\mathrm{a,b})$ The $\mathscr{L}-$narrowing $\mathtt{'s}_x \mapsto \mathtt{s}_y\mathtt{s}_x\mathtt{'}$ is compatible with an equation having sides starting with different equation variables named $\mathtt{s}_x$ and $\mathtt{s}_y$. $*/$
$\mathtt{Main}(((\mathbf{Vs}_x) \mapsto (\mathbf{Vs}_y)(\mathbf{Vs}_x)) \mathtt{+\!\!+} \mathtt{x}_{\mathtt{rul}}, ((\mathbf{Vs}_y) \mathtt{+\!\!+} \mathtt{x}_{\mathrm{LHS}}, (\mathbf{Vs}_x) \mathtt{+\!\!+} \mathtt{x}_{\mathrm{RHS}}))$
$= \mathtt{Main}(\mathtt{x}_{\mathtt{rul}}, \mathtt{Simplify}(((\mathbf{Vs}_x) \mapsto (\mathbf{Vs}_y)(\mathbf{Vs}_x)), \mathtt{Subst}((\mathbf{Vs}_x) \mapsto (\mathbf{Vs}_y)(\mathbf{Vs}_x), \varepsilon, \mathtt{x}_{\mathrm{LHS}}),$
$\mathtt{Subst}((\mathbf{Vs}_x) \mapsto (\mathbf{Vs}_y)(\mathbf{Vs}_x), (\mathbf{Vs}_x), \mathtt{x}_{\mathrm{RHS}})));$
$\mathtt{Main}(((\mathbf{Vs}_x) \mapsto (\mathbf{Vs}_y)(\mathbf{Vs}_x)) \mathtt{+\!\!+} \mathtt{x}_{\mathtt{rul}}, ((\mathbf{Vs}_x) \mathtt{+\!\!+} \mathtt{x}_{\mathrm{LHS}}, (\mathbf{Vs}_y) \mathtt{+\!\!+} \mathtt{x}_{\mathrm{RHS}}))$
$= \mathtt{Main}(\mathtt{x}_{\mathtt{rul}}, \mathtt{Simplify}(((\mathbf{Vs}_x) \mapsto (\mathbf{Vs}_y)(\mathbf{Vs}_x)), \mathtt{Subst}((\mathbf{Vs}_x) \mapsto (\mathbf{Vs}_y)(\mathbf{Vs}_x), (\mathbf{Vs}_x), \mathtt{x}_{\mathrm{LHS}}),$
$\mathtt{Subst}((\mathbf{Vs}_x) \mapsto (\mathbf{Vs}_y)(\mathbf{Vs}_x), \varepsilon, \mathtt{x}_{\mathrm{RHS}})));$
$/*\,(5)$ Stop the computation in the default case. $*/$
$\mathtt{Main}(\mathtt{x}_{\mathtt{rul}}, (\mathtt{x}_{\mathrm{LHS}}, \mathtt{x}_{\mathrm{RHS}})) = \mathbf{F};$

Figure 9: The function `Main` accepts two arguments: the first is an encoded substitutions list, and the second is an encoded pair representing an equation. Here $\mathtt{s}_x$ takes a name of an equation variable, if $x \in \mathscr{V}$, while $\mathtt{s}_{\mathtt{sym}}$ takes a character. The second argument of `Subst` function is an accumulator.

Property 2. *Let us consider the process tree $\mathscr{T}$ generated by the specialization task $\mathfrak{M}(\mathrm{WI}_{\mathscr{L}}, \mathscr{E}qs)$, where* $\mathrm{WI}_{\mathscr{L}} \in \{\mathrm{WIBase}_{\mathscr{L}}, \mathrm{WISplit}_{\mathscr{L}}, \mathrm{WICount}_{\mathscr{L}}\}$.

- *Given an arbitrary configuration C labelling a node in $\mathscr{T}$, the only parameterized call in C (if any) is of the form* $\mathtt{Main}(v_i, \text{Other args})$*, where no parameter occurs*[9] *in the* other arguments.
- *The patterns to be unified with the parameterized data never have more than one occurrence of an expression-type variable.*

Henceforth we say that a configuration C is primary if its unfolding results in parameter narrowings, and we call a node primary if configuration labelling it is primary. Property 2 implies that the par-narrowings constructed by the unfolding step are always substituted only to the function call `Main` being the active call. Hence, for the verification task considered, C is primary iff C is of the form $\mathtt{Main}(v_i, \text{Other args})$, where the other arguments do not contain function calls.

Properties 2 and 1 together imply that in the case of the verification task considered the exhaustive narrowing set always consists of the only par-narrowing, thus, a unification with one rewriting rule

[9] A rewriting rule $\mathtt{Main}(\mathrm{P}, S_1) = S_2$ may include letter-type pattern variables shared by P and S_1 that occur in S_2, but the value matched against S_1 is always an object expression, hence these variables are assigned with letters in any pattern matching.

results in a single par-narrowing. Depending on the form of the rule defining the function `Main`, the par-narrowing would be either $v_i \mapsto \varepsilon$, $v_i \mapsto L' +\!\!+ v_i'$, where L' is an object expression, or trivial $v_i \mapsto v_i$, if parameter v_i is unified with the only variable $\mathtt{x_{rul}}$, in rule (5). Hence, every non-trivial par-narrowing corresponds to an $\mathscr{L}$-narrowing of the variables of the equation being transformed. Provided this feature[10], the unification process is always finite [28]. Note that no trivial `otherwise` branch corresponds to a branch in the equation solution tree.

We call a node transient [39], if the one-step unfolding of the configuration labelling it produces no narrowing on the parameters; in particular, if all the $\mathrm{Expr_i}$ in $\mathrm{F}(\mathrm{Expr_1},\ldots,\mathrm{Expr_n})$ are object expressions. A transient node has the only child in the process tree.

6 Results of Specialization

This section discusses some conditions under which the verification succeeds, and presents several sets of word equation systems, which have been solved by means of the specialization task $\mathfrak{M}(\mathrm{WI}_{\mathscr{L}}, \mathscr{E}qs)$, where $\mathrm{WI}_{\mathscr{L}}$ is either $\mathrm{WIBase}_{\mathscr{L}}$, $\mathrm{WISplit}_{\mathscr{L}}$ or $\mathrm{WICount}_{\mathscr{L}}$.

Given an equation list $\mathscr{E}qs$ and an interpreter $\mathrm{WI}_{\mathscr{L}}$, the final result of the stepwise unfolding of $\mathfrak{M}(\mathrm{WI}_{\mathscr{L}}, \mathscr{E}qs)$ can be considered as a possibly infinite process tree modelling the solution tree of $\mathscr{E}qs$. The folding occurs if a node in the process tree has an ancestor labelled with the same configuration modulo the parameter renaming. If the unfolding can produce infinite paths in the tree then the specialization process does not terminate, unless two equal configurations exist along every infinite path. Thus, the specialization terminates iff the relation of the textual equality is a well-quasi order over the configuration sequence along every infinite path in the process tree.

6.1 Optimality of Specialization

In this section, we show that given the structure of the interpreters considered, the residual graphs produced by the specialization may be reduced to the solution graphs of the equations considered. For this purpose we have to consider the unfolding and the folding operations, which generate both the process graph and the solution graph in very similar ways. In Sec. 5 we have shown that every node in the process tree, whose one-step unfolding results in the set $\{\xi_i\}$ of the disjoint par-narrowings marking the outgoing arcs, corresponds to a node in the solution tree, and there is a bijection from the arc set marked by ξ_i to the arc set marked by the corresponding $\mathscr{L}$-narrowings, thus it remains to show that the folding does work exactly on the same nodes where the par-narrowings and $\mathscr{L}$-narrowings are generated.

In general, a partial process tree of the specialization task may require to construct a folding arc connecting transient nodes. That would cause problems with the reasoning on the process graphs in the terms of the solution graphs, because the transient nodes do not correspond to any nodes in the solution graph. Informally, we can say that the specialization result is optimal if no folding arcs connect transient nodes. The structure of the interpreters $\mathrm{WIBase}_{\mathscr{L}}$, $\mathrm{WISplit}_{\mathscr{L}}$, $\mathrm{WICount}_{\mathscr{L}}$ guarantees that all the non-transient nodes are also primary (see Property 2). Thus, we define now a notion of the optimal specialization for the verification task given in Sec. 4.

[10] Here the unfolding step has another important property: for all function rules excluding the last one (rule (5)), the narrowings imposed on the parameter v_i are always disjoint, provided that the equations in the list given to the function `Main` are of the reduced form. The rule (5) accepts an arbitrary input, serving as the `otherwise` branch. The right-hand side of this rule is an object expression, hence there is no need to propagate negative constraints imposed on the parameter value to the successor configurations.

Definition 3. *Given a task* $\mathfrak{M}(\mathrm{WI}_{\mathscr{L}}, \mathscr{Eqs})$*, its specialization result is said to be optimal iff all the arcs folding computation paths in the process graph connect the primary nodes.*

The interpreters considered above satisfy the property that every parameter narrowing occurring in a process tree either generates an $\mathscr{L}$-narrowing or results in an **F**-node never unfolded. If the optimality holds, then all the intermediate steps of the specialization of the interpreters, including specialization of substitution and simplification, correspond to the nodes in which the folding never occurs, hence every transient path segment in the process graph may be represented with a single arc. We can therefore reason on the process graphs using the solution graphs of the equations *w.r.t.* which the interpreters are specialized. Moreover, the optimality guarantees that the residual programs generated by a specialization tool contain no part of the interpreters' source code, except the encoding $\lfloor P \rfloor$ of the $\mathscr{L}$-programs P. Thus, the introduced optimality can be considered as an analogue of the Jones-optimality [3, 16] for the given verification task. The Jones-optimality demands that the interpretation overheads should be completely removed from the residual programs. The notion of the optimality given in Definition 3 implies also that all the interpretation overheads are removed from the specialization result, although some pieces of the encoded $\mathscr{L}$-programs will be present in it, since the parameters are narrowed according to their values.

Since we consider the input sequence of $\mathscr{L}$-narrowings given to an interpreter $\mathrm{WI}_{\mathscr{L}}$ as a straight-line program, we can also use the following reasoning [22]. In the classical first Futamura projection [12], which corresponds to the specialization task $\mathtt{Spec}\Big(\underline{\mathrm{WI}_{\mathscr{L}}}, \underline{\mathtt{Go}(\langle \lfloor \sigma_i \rfloor \rangle, v)}\Big)$, the input data (the equation list) is dynamic, while the program given to the interpreter is static. Here we parameterize the program.

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

The idea of the proof is as follows. If the nodes N_1 and N_2 labelled with the equal non-primary configurations exist along the same path, then the closest primary ancestor of N_1 and the closest primary ancestor of N_2 are also labelled with the equal configurations. The property holds because the structure of the interpreters implies the following two statements. First, given any primary configuration $\mathtt{Main}(v_i, \mathscr{Eqs})$ along the path segment $[\mathsf{N}_1; \mathsf{N}_2]$ and the first primary configuration $\mathtt{Main}(v_j, \mathscr{Eqs}')$ labelling a successor node of N_2, the length of the equation lists $\mathscr{Eqs}$ and $\mathscr{Eqs}'$ are equal. Second, all the equation transformations done along the segment $[\mathsf{N}_1; \mathsf{N}_2]$ are injective. The detailed proof is given in Appendix, Sec. 8.1.

Provided that the optimality holds, given a class of equations K, we say the verification by specialization of the interpreter $\mathrm{WI}_{\mathscr{L}}$ succeeds over K iff for every equation $E \in \mathsf{K}$ the solution graph constructed by the corresponding algorithm solving the equation is finite.

Let us show how the process graph generated by the specialization of an interpreter *w.r.t.* an equation corresponds to a solution graph of the equation, using the following example. Here we consider an equation such that the choice of the simplification algorithm has no impact on its solution.

Example 3. *Let us consider the task* $\mathfrak{M}(\mathrm{WI}_{\mathscr{L}}, \mathtt{x}\,\mathtt{y}\,\mathbf{A} = \mathbf{A}\,\mathtt{x}\,\mathtt{y})$*. Using the operations given above, we construct its process graph. Some structures of this process graph are relevant only to the interpreter, namely to the structures of the parameter narrowings and the function calls. If we delete them, as well as the* otherwise *branches, we will get a solution graph of the equation, as is shown in Figure 10.*

6.2 Specialization of Basic Interpreter

Definition 4. *A word equation* $\Phi = \Psi$ *is said to be* quadratic *iff for all* $\mathtt{x} \in \mathscr{V}$*,* $|\Phi|_{\mathtt{x}} + |\Psi|_{\mathtt{x}} \leq 2$*.*

For every quadratic equation, the solution graph constructed with the use of the algorithm $\mathrm{AlgWE}_{\mathrm{Base}}$ is finite. This fact is well-known due to the works by Matiyasevich [18, 19, 21, 26]. Thus, specialization

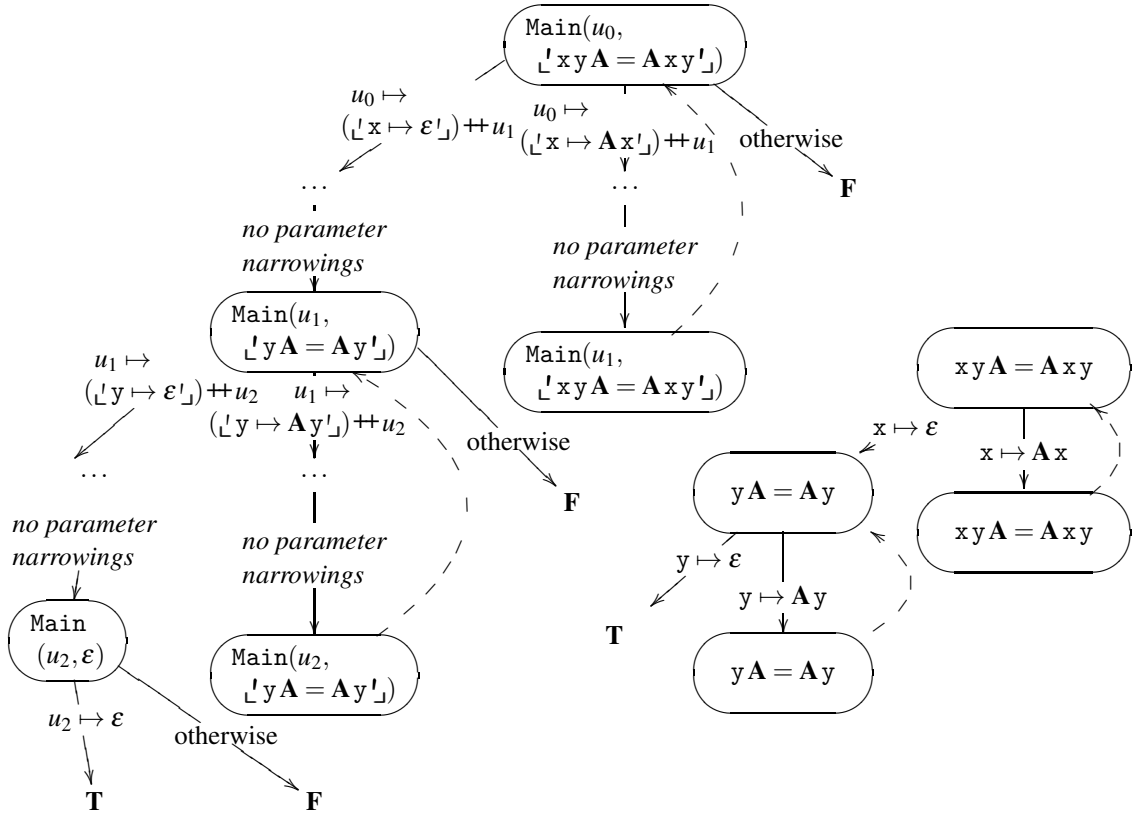


Figure 10: A process graph and the corresponding solution graph. The quoted data (encoded in the process graph) are preserved in the solution graph.

of WIBase$_{\mathscr{L}}$ *w.r.t.* quadratic equations provides a basic test on the optimality of the program model. Namely, the optimality lemma implies the following proposition.

Proposition 3. *For any quadratic equation* $\Phi = \Psi$*, specialization of* $\mathfrak{M}(\text{WIBase}_{\mathscr{L}}, \Phi = \Psi)$ *succeeds.*

6.3 Specialization of Splitting Interpreter

The interpreter WISplit$_{\mathscr{L}}$ was introduced as an optimized version of WIBase$_{\mathscr{L}}$, but the experiments have shown that the specialization of WISplit$_{\mathscr{L}}$ succeeds in significantly more cases. One interesting class of the word equations solvable with the help of WISplit$_{\mathscr{L}}$ consists of a special kind of equations whose solution sets are regular languages.

Definition 5. *Given* $\Phi \in \{\mathscr{A} \cup \mathscr{V}\}^*$*, let* $\xi(\Phi)$ *map any* $\mathbf{A} \in \mathscr{A}$ *explicitly occurring in* Φ *to* ε*, preserving the other part of* Φ*. 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 the strictly regular-ordered

equations with repetitions generalizes the set of the regular ordered equations in which every variable occurs twice [9].

Example 4. *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 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 use of the algorithm* $\mathrm{AlgWE}_{\mathrm{Base}}$ *is infinite.*

The termination of the specialization is provided by the following lemma.

Lemma 2. *Given any strictly regular-ordered equation with repetitions* $\Phi = \Psi$, *every infinite path in its solution tree generated with the use of the algorithm* $\mathrm{AlgWE}_{\mathrm{Split}}$ *includes at least two nodes with equal labels.*

The idea of the proof is as follows. Every such an equation $\Phi = \Psi$ is split into the several quadratic equations after a number of substitutions applied to it. The detailed proof is given in Appendix (Sec. 8.2).

Corollary 1. *Given any strictly regular-ordered equation with repetitions* $\Phi = \Psi$, *specialization of the verification task* $\mathfrak{M}\big(\mathrm{WISplit}_{\mathscr{L}}, \langle \Phi = \Psi \rangle\big)$ *succeeds.*

6.4 Specialization of Counting Interpreter

The interpreter $\mathrm{WICount}_{\mathscr{L}}$ uses more simplifying operations as compared to the interpreter $\mathrm{WISplit}_{\mathscr{L}}$. The specialization of this interpreter succeeds additionally in solving one-variable equations. The success of the verification is guaranteed by the following lemma.

Lemma 3. *Given an equation* $\Phi = \Psi$, *where* $\Phi, \Psi \in \{\mathscr{A} \cup \{\mathtt{x}\}\}^*$, *and* $|\Phi\Psi|_{\mathtt{x}} > 2$, *every infinite path of its solution graph constructed with the use of the algorithm* $\mathrm{AlgWE}_{\mathrm{Count}}$ *contains a split.*

The idea of the proof is similar to the one of Lemma 2: if $\Phi = \Psi$ has a solution, then after an application of a number of the substitutions, the resulted equation will have var-permutated prefixes or suffixes. See the Appendix for the details (Sec. 8.3).

Corollary 2. *For any one-variable equation* $\Phi = \Psi$, *specialization of the task* $\mathfrak{M}(\mathrm{WICount}_{\mathscr{L}}, \langle \Phi = \Psi \rangle)$ *succeeds.*

In order to experimentally test the verification technique presented in this paper, we have generated a benchmark consisting of 50 equation systems[11]: 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 the variable occurrences order may be different on the equation sides where the variables occur; the tests 21–30 contain equations of the form $\mathtt{x}\Phi = \Psi\mathtt{x}$, where $\Phi = \Psi$ is a 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 supercompiler SCP4 [27] was mainly used as $\mathfrak{M}$ in the tests. The experimental supercompiler `MSCP-A` was also used and has shown the same solvability results[12] 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 11, the last row. The results show that the scheme $\mathfrak{M}(\mathrm{WICount}_{\mathscr{L}}, \mathscr{E}qs)$ is quite stable modulo small changes in the equations which are guaranteed to be solved by it.

[11] The archive containing the equations is given on the web-page `https://github.com/TonitaN/TestEquations`.

[12] See the web-page `http://refal.botik.ru/mscp/weq_int_readme.html` for details.

Finally, we have tested the scheme $\mathfrak{M}(\text{WICount}_{\mathscr{L}}, \mathscr{Eqs})$ 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[13]. 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 $\mathfrak{M}(\text{WICount}_{\mathscr{L}}, \mathscr{Eqs})$ never contain functions returning **T** if the system $\mathscr{Eqs}$ has been found unsatisfiable by the other solvers. Moreover, if the system $\mathscr{Eqs}$ has solutions, then $\mathfrak{M}(\text{WICount}_{\mathscr{L}}, \mathscr{Eqs})$ always generates programs containing functions with the output **T**, if terminates. That is a practical evidence that the specialization produces sound and complete solution graphs, which is a corollary of the optimality lemma. Second, the equations are successfully solved in 179 out of 200 cases in Track 1 and in 181 out of 200 cases in Track 5. This result is comparable with the verification results done by `Z3str3` [24]; for 17 equations in Track 1 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 average runtime of $\mathfrak{M}(\text{WICount}_{\mathscr{L}}, \mathscr{Eqs})$ on the tests considered is about 3.5 minutes per task. While the runtime of all the tests solved successfully by `CVC4` or `Z3str3` is less than 2 minutes. Although we have used the 3-hour timeout, the long-running tests resulting in the verification success occurred only for the scheme $\mathfrak{M}(\text{WICount}_{\mathscr{L}}, \mathscr{Eqs})$. There are two main reasons of such a difference. First, the scheme uses the general-purpose specialization tool, employing time-consumable transformation operations, such as the residualization. Second, the scheme considered searches for all the solutions of the equations rather than for at least one. That is the main reason for the long runtimes, and is immanent to the problem solved. Many equations have the solution set exponentially-sized *w.r.t.* their length.

Benchmark	Tests (total)	Timeout / not terminated		
		`CVC4`	`Z3str3`	$\text{WICount}_{\mathscr{L}}$
Track 1 (Woorpje)	200	8	13	21
Track 5 (Woorpje)	200	4	14	19
Our benchmark	50	21	28	10

Figure 11: The verification results.

7 Discussion

The discussed specialization tasks above have been solved by using the following two program specializers developed for the string manipulating functional language Refal [38], namely the model supercompiler[14] `MSCP-A` and the experimental supercompiler `SCP4` [27]. Albeit we used the supercompilers, the properties of the residual programs, which we are interested in, do not depend on specific features of the supercompilation method [37] and can be reproduced with other specializers based on partial evaluation, partial deduction, and so on [17, 30]. We used the language $\mathscr{F}$ with built-in concatenation constructor, however the method can be used also over the lisp-data with some minor changes in the interpreters'

[13]The reader can find the residual programs encoding paths in the solution graphs for the equations generated in the experiments at the web-page `https://github.com/TonitaN/TestEquations`.

[14]The supercompiler is presented on the web-page `http://refal.botik.ru/mscp/mscp-a_eng.html`

source code. First, all the parameter narrowings are constructed only by specializing the head of the list of the $\mathscr{L}$-narrowings. Second, the optimality lemma guarantees that the additional loops in the intermediate steps of the interpretation would not cause any folding operations, since the data given to the intermediate functions is not parameterized. Thus, the structure of the residual programs is preserved.

Our approach is able to solve the regular-ordered equations with repetitions (see 6.3), which are hard to solve for the known existing solvers, especially in the case when the solution set is empty. For example, neither `Z3str3` nor `CVC4` terminates on the equation $\mathbf{A}\,\mathbf{B}\,\mathtt{x}\,\mathtt{x}\,\mathtt{y}\,\mathtt{y} = \mathtt{x}\,\mathtt{x}\,\mathtt{y}\,\mathtt{y}\,\mathbf{B}\,\mathbf{A}$, which is proved to have no solution by the WISplit$_{\mathscr{L}}$ or WICount$_{\mathscr{L}}$ specialization. The feature to solve equations with the empty solution set is especially interesting provided the fact that this case is the hardest for the most theoretical algorithms, and is also the bottleneck for the SMT-solvers used in our tests.

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 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 $\mathfrak{M}(\text{WICount}_{\mathscr{L}}, \mathscr{E}qs)$ 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 efficient string constraint solving tools were designed, which look for the word equation solutions bounded by a given length, *e.g.* [4, 8][15]. Reasonings on the unbounded case can provide efficient methods for 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. The difference is again rooted in the tasks considered: our approach looks for a description of the whole set of equation solutions, while SMT-solvers aim to find at least one solution.

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 [26]. In the paper [21], the algorithm using non-deterministic counter systems for searching solutions of the unbounded-length word equations with regular constraints via Nielsen's transformation is 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 AlgWE$_{\text{Split}}$ algorithm (Sec. 3), can be solved by this method as well. In the paper [19], Nielsen's transformation is used for solving unbounded-length quadratic word equations. As in the original Matiyasevich work, the algorithm does not terminate if the input equation contains more than two occurrences of some variable. Thus, these algorithms are not able to solve non-quadratic regular-ordered equations with repetitions, which are solvable by our specialization scheme. Advanced SMT-solvers such as `CVC4` [20], `Z3str3` [24], or `S3P` presented in [36] demonstrate a very good efficiency in many practical cases, however the paper [19] shows that their algorithms are not complete even *w.r.t.* the set of the quadratic equations, *e.g.* do not solve on 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 very first proof of this fact. The tests of our benchmark have shown that the current version of `CVC4` solves equations of the given form, but fails to solve more complex quadratic equations,

[15] Actually, 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].

e.g. $\mathtt{x}_1\,\mathtt{x}_2\,\mathtt{x}_3\,\mathbf{ABABAB} = \mathbf{AAABBB}\,\mathtt{x}_2\,\mathtt{x}_3\,\mathtt{x}_1$ (which is solved by specialization even of the basic interpreter WIBase$_{\mathscr{L}}$). Based on the results of the verification presented in this paper we may conclude that the most troublesome cases for the SMT-solvers are the ones when the equation system has no solution, and this fact cannot be shown by reasoning on solution lengths. In that cases, the verification scheme $\mathfrak{M}(\text{WICount}_{\mathscr{L}}, \mathscr{E}qs)$ 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$_{\mathscr{L}}$, 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 are able to prove unsatisfiability of word equation systems, 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 [34]. 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.

Acknowledgements

We would like to thank A. P. Nemytykh, who contributed greatly to the improvement of the paper, and the anonymous referees for the thoughtful comments which helped a much to clarify the presentation.

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] S. Barker, M. Leuschel & M. 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* 2(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*. *POPL* 3, pp. 1–30, doi:10.1145/3290362.

[7] Christian Choffrut & Juhani Karhumäki (1997): *Combinatorics of Words*. *Handbook of Formal Languages*, pp. 329–438, 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, pp. 93–106, 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.

[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.

[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 (2002): *Computability and Complexity from a Programming Perspective*. 62, NATO Science Series, Springer, doi:10.1007/978-94-010-0413-8_4.

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

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

[19] Quang Loc Le & Mengda He (2018): *A Decision Procedure for String Logic with Quadratic Equations, Regular Expressions and Length Constraints*, pp. 350–372. 11275, Lecture Notes in Computer Science, doi:10.1007/978-3-030-02768-1_19.

[20] 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.

[21] 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, pp. 352–369, doi:10.1007/978-3-030-01090-4_21.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

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

[36] 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.

[37] 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.

[38] 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/`.

[39] 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, pp. 1–28.

[40] 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.

[41] 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 on Optimality

Given a word Φ, a prefix (suffix) of Φ is said to be proper if it does not coincide with Φ. Henceforth we consider only the non-empty prefixes, *e.g.* the var-permutated prefixes are non-empty by their definition.

Proposition 4. *Let E be a reduced word equation $\Phi = \Psi$ with the var-permutated sides and without var-permutated suffixes and prefixes; σ be an arbitrary substitution given in Figure 5. Let E' be $\Phi\sigma = \Psi\sigma$ after the reduction. 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 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 of the form $\sigma\colon \mathtt{x} \mapsto t\,\mathtt{x}$, where $t \in \mathscr{A} \cup \mathscr{V}$. We may present the equation sides as $\Phi = \Phi_1\mathtt{x}\ldots\Phi_{n-1}\mathtt{x}\Phi_n$ 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 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 proper 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 $\Psi_1\,t\,\mathtt{x}\ldots t\,\mathtt{x}\Psi_i\,t$, or vice versa. We consider only the first case, because they are symmetric.

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[16].

2. Let a substitution σ be of the form $\mathtt{x} \mapsto t\,\mathtt{x}$, where $t \in \mathscr{A} \cup \mathscr{V}$ and $\mathtt{x}$ occurs either in Φ or in Ψ. Otherwise, the substitution has no impact on the equation length. Following the first case proven above, we do not 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 $\mathrm{AlgWE}_{\mathrm{Split}}$ or $\mathrm{AlgWE}_{\mathrm{Count}}$.

Proposition 5. *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.*

[16] 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.

Proof. Let E be a reduced equation $\Phi = \Psi$ *s.t.* Φ and Ψ start with different terms, and let us assume that there exists E' *s.t.* $E = E'\sigma$, where σ is compatible with E'. 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 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 $\sigma^{-1}\colon t\,\mathtt{x} \mapsto \mathtt{x}$, 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 $t\,(\Psi\sigma^{-1})$. □

Proposition 5 states that given an equation E and a substitution σ of a special kind *s.t.* a node N in a solution tree generated with the use of the algorithm $\mathrm{AlgWE}_{\mathrm{Base}}$ is labelled by E and has the ingoing arc labelled with σ, then the label of the parent of N can be restored. But this proposition does not require σ to be generated according to the substitutions given in Figure 5: both Φ and Ψ may start with terms differing from $\mathtt{x}$.

The previous propositions refer to the equation solution trees. The following proof of Lemma 1 refers to the process trees (graphs). We recall that the notion of a configuration is given in Definition 1.

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

First, consider the interpreter $\mathrm{WIBase}_{\mathscr{L}}$. See Sec. 8.4 for its source code. The node configurations in the process tree of $\mathrm{WIBase}_{\mathscr{L}}$ are of the following forms:

1. $\mathtt{Main}(v_i, \llcorner E \lrcorner)$ (primary configurations);
2. $\mathtt{Main}(v_i, \mathtt{Simplify}(\llcorner \sigma \lrcorner, \mathit{Other\ args}))$, where $\llcorner \sigma \lrcorner$ is static data encoding the substitution that is last applied[17] to the equation list, and the other arguments may include a call of the function `Subst`.

The function `Main` applies the first substitution in the list given in its first argument to the equation list given in the second argument; the function `Simplify` simplifies, *i.e.* reduces, the equations. If only the primary configurations are folded, the specialization is already optimal. Let the nodes N_1 and N_2 be labelled with the configurations of the form $\mathtt{Main}(v_i, \mathtt{Simplify}(\llcorner \sigma \lrcorner, \mathit{Other\ args}))$ *s.t.* the configurations coincide up to the parameter renaming. We will prove now that their ancestors labelled with the primary configurations also should have equal labels, thus, the folding operation should be applied before the whole path part $[\mathsf{N}_1;\mathsf{N}_2]$ is unfolded[18].

If N_1 and N_2 are non-primary and are labelled with the equal configurations, then the primary ancestors of the nodes N_1 and N_2 always exist. Note that the first call of `Simplify` initialized by `Go`, which is not preceded with a primary configuration, is of the form $\mathtt{Simplify}(\varepsilon, \ldots)$, while all other calls of `Simplify` have a non-empty first argument. This implies that two configurations of the form $\mathtt{Main}(v_i, \mathtt{Simplify}(\varepsilon, \mathit{Other\ args}))$ cannot be folded — that would imply that no par-narrowing is generated along the path segment $[\mathsf{N}_1;\mathsf{N}_2]$, thus, the function `Simplify` would be non-terminating. Thus, the configurations that label the nodes N_1 and N_2, assumed to be folded, can be only of the form $\mathtt{Main}(v_i, \mathtt{Simplify}(\llcorner \sigma \lrcorner, \mathit{Other\ args}))$ with non-empty σ. Note that there may be several primary nodes along the segment $[\mathsf{N}_1;\mathsf{N}_2]$.

Given the nodes N_1 and N_2, let us take their closest primary ancestors, named N'_1 and N'_2 respectively. Let the configurations labelling them be of the forms $\mathtt{Main}(v, \llcorner E \lrcorner)$ and $\mathtt{Main}(v', \llcorner E' \lrcorner)$. Let the closest primary successors of N'_1 and N'_2, named $\mathsf{N}_{\mathrm{loop\_1}}$ and $\mathsf{N}_{\mathrm{loop\_2}}$ respectively, be labelled with the configurations

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

[18] The scheme of the proof given in Figure 12 refers to the interpreters $\mathrm{WISplit}_{\mathscr{L}}$ and $\mathrm{WICount}_{\mathscr{L}}$, although the only significant difference between the reasonings is that the simplification function and `Main` function in $\mathrm{WIBase}_{\mathscr{L}}$ do not use an additional information about the number of equations in the list, namely, n on the scheme.

$\texttt{Main}(v_i, \llcorner E_i \lrcorner)$. The path segments $[\mathsf{N}_1; \mathsf{N}_{\text{loop\_1}}]$ and $[\mathsf{N}_2; \mathsf{N}_{loop\_2}]$ contain only transient nodes. Thus, the configurations labelling $\mathsf{N}_{\text{loop\_1}}$ and $\mathsf{N}_{\text{loop\_2}}$ coincide up to the name of the parameter v_i. Let $E_i = E_0$. No $\mathscr{L}$-narrowings generating a substitution $\mathtt{x} \mapsto \varepsilon$ can occur along the segment $[\mathsf{N}_{\text{loop\_1}}; \mathsf{N}_{\text{loop\_2}}]$, otherwise the equation E_0 would not be preserved in $\mathsf{N}_{\text{loop\_2}}$. Thus σ is $\mathtt{x} \mapsto t\,\mathtt{x}$, $t \in \mathscr{A} \cup \mathscr{V}$. Here the substitution σ is the same one used in the configurations labelling N_1 and N_2, because it is the first argument of the function call `Simplify`, and, when applied to the equations E and E', σ generates the same equation E_0. Hence by Proposition 5 E and E' coincide. This proves the lemma in the case of the interpreter $\text{WIBase}_{\mathscr{L}}$.

Let us consider now specialization of interpreters $\text{WISplit}_{\mathscr{L}}$ and $\text{WICount}_{\mathscr{L}}$. In these cases the possible configurations are exhausted with the following ones:

1. $\texttt{Main}(v, (n) +\!\!+ \llcorner \langle E_i \rangle_{i=1}^{n'} \lrcorner)$, where n coincides with n' unless $\langle E_i \rangle_{i=1}^{n'}$ consists of a single unsatisfiable equation;
2. $\texttt{Main}(v, \texttt{Simplify}(n, \llcorner \sigma \lrcorner, \mathit{Other\ args}))$, where n is the length of the equation list in the configuration labelling the closest primary ancestor of the node considered, and σ is the last substitution that was applied to the equations; the other arguments may contain function calls;
3. $\texttt{Main}(v, \texttt{CountEq}(n, \llcorner \sigma \lrcorner, \mathit{Other\ args}))$.

The argumentation for the case 3 does not differ from the one that will be given in the case 2, thus we assume the fold operation works only with the nodes N_1 and N_2 labelled with the configurations of the form $\texttt{Main}(v_i, \texttt{Simplify}(n, \llcorner \sigma \lrcorner, \mathit{Other\ args}))$. Once again we consider their closest ancestor nodes N'_1 and N'_2 labelled with the primary configurations $\texttt{Main}(v_0, (n) +\!\!+ \llcorner \langle E_i \rangle_{i=1}^{n} \lrcorner)$ and $\texttt{Main}(v_{k-1}, (n) +\!\!+ \llcorner \langle E'_i \rangle_{i=1}^{n} \lrcorner)$, as that was done for $\text{WIBase}_{\mathscr{L}}$ case.

Here the number n is the same in the both configurations, because it is the same in the configurations of the nodes N_1, N_2. Function `Simplify` takes its first argument from the last `Main` call, hence they are the same in all the four configurations. The substitution $\mathtt{x} \mapsto \varepsilon$ cannot be applied along the path segment $[\mathsf{N}_1; \mathsf{N}_2]$, by the reasoning above. Let σ be $\mathtt{x} \mapsto t\,\mathtt{x}$, $t \in \mathscr{A} \cup \mathscr{V}$. By Proposition 4, the number of the equations in $[\mathsf{N}'_1; \mathsf{N}'_2]$ cannot decrease. Thus, the number of the equations is a constant along the path segment, namely n, 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 algorithms implemented in $\text{WIBase}_{\mathscr{L}}$. Given the first primary successors $\mathsf{N}_{\text{loop\_1}}$ and $\mathsf{N}_{\text{loop\_2}}$ of N_1 and N_2, by their choice, the segments $[\mathsf{N}_1; \mathsf{N}_{\text{loop\_1}}]$ and $[\mathsf{N}_2; \mathsf{N}_{\text{loop\_2}}]$ consist only of the transient nodes, Thus, $\mathsf{N}_{\text{loop\_1}}$ and $\mathsf{N}_{\text{loop\_2}}$ are labelled with the equal lists of equations, thus by Proposition 5 the lists of equations $\langle E'_i \rangle_{i=1}^{n}$ and $\langle E_i \rangle_{i=1}^{n}$ also coincide. □

8.2 Proofs and Auxiliary Propositions on Regularly-Ordered Equations with Repetitions

The next propositions consider the equation solution algorithms presented in Sec. 3 and do not refer to the interpreters' structure. The equation solution graphs are considered here, like ones shown in Figure 7.

Proposition 6. *Let $Q_i, Q'_i \in \mathscr{A}^+$. Given a list of equations $\mathscr{E}qs = \langle Q_1\,\mathtt{x}_1 = \mathtt{x}_1\,Q'_1, \ldots, Q_n\,\mathtt{x}_n = \mathtt{x}_n\,Q'_n \rangle$, where for some i, j, $\mathtt{x}_i = \mathtt{x}_j$ may hold, the label of any node in the solution tree constructed with the use of the algorithm $\text{AlgWE}_{\text{Split}}(\mathscr{E}qs)$ is a list consisting of at most n equations. If it consists exactly of n equations $\langle \Phi'_1 = \Psi'_1, \ldots, \Phi'_n = \Psi'_n \rangle$, then $|\Phi'_i| \leq |Q_i| + 1$, $|\Psi'_i| \leq |Q'_i| + 1$.*

Proof. Figure 5 shows that the possible substitutions that can be compatible with the initial equation list $\langle Q_1\,\mathtt{x}_1 = \mathtt{x}_1\,Q'_1, \ldots, Q_n\,\mathtt{x}_n = \mathtt{x}_n\,Q'_n \rangle$ 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 $Q_i\,\mathtt{x}_i = \mathtt{x}_i\,Q'_i$

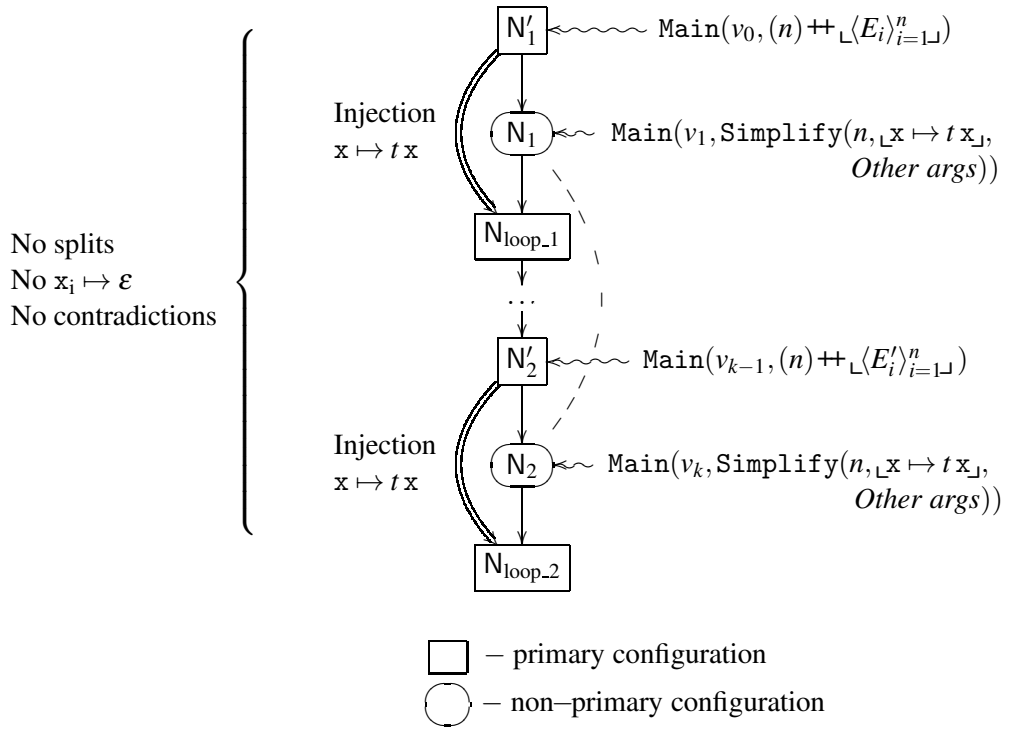


Figure 12: Scheme of the Optimality Lemma proof.

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 3. *If $Q_i, Q'_i \in \mathscr{A}^+$ then every infinite path of a solution graph constructed with the use of the algorithm* $\mathrm{AlgWE}_{\mathrm{Split}}$ *applied to the list $\langle Q_1\,\mathtt{x}_1 = \mathtt{x}_1\,Q'_1, \ldots, Q_n\,\mathtt{x}_n = \mathtt{x}_n\,Q'_n\rangle$ contains two nodes with equal labels.*

Proposition 7. *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 constructed with the use of the algorithm* $\mathrm{AlgWE}_{\mathrm{Split}}(\Phi = \Psi)$ *never includes an application of the $\mathscr{L}$-narrowing $\mathtt{x} \mapsto \mathtt{y}\,\mathtt{x}$ given in Figure 5.*
3. *Every infinite path in the solution tree constructed with the use of* $\mathrm{AlgWE}_{\mathrm{Split}}(\Phi = \Psi)$ *contains a finite number of the split operations.*

Proof. 1. For every variable $\mathtt{x}_i \in \mathscr{V}$, the var-permutated property 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 in the prefixes and suffixes as well.

2. Consider the variable $\mathtt{x}$ leading in Φ (and occurring in Ψ before any other variable). 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 an $\mathscr{L}$-narrowing 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 strictly-regular-ordered property, as well as the splitting operation does.

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

□

Here we repeat Lemma 2 before giving its detailed proof.

Lemma 2. *Given a strictly regular-ordered equation with repetitions $\Phi = \Psi$, every infinite path in its solution tree constructed with the use of the algorithm* $\mathrm{AlgWE}_{\mathrm{Split}}(\Phi = \Psi)$ *includes at least two nodes with equal labels.*

Proof. Every infinite path in the tree generated with the use of $\mathrm{AlgWE}_{\mathrm{Split}}(\Phi = \Psi)$ 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 label 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 only 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 $\sigma\colon \mathtt{x} \mapsto t\,\mathtt{x}$ ($t \in \mathscr{A}$), followed by the reduction, preserves its length. The number of the equations in the lists labelling the nodes along the path, as well as the lengths of the equations, cannot grow, hence the $\mathscr{L}$-narrowings do not generate fresh variables, and thus the set of the labels of the nodes 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 $\mathtt{x} \mapsto c_1\,\mathtt{x}, \ldots, \mathtt{x} \mapsto c_m\,\mathtt{x}$, where $c_i \in \mathscr{A}$ are letters of Θ. Let $\Theta' = c_1 \ldots c_m$. The ending node of the m-length path segment is labelled with the list containing the following equation: $\langle \ldots, \Phi'_{0,j}\,\mathtt{x}\,\Phi_{1,j}\,\Theta'\,\mathtt{x}\,\Phi'_{2,j} = \mathtt{x}\,\Psi_{1,j}\,\Theta'\,\mathtt{x}\,\Psi'_{2,j}, \ldots \rangle$, $|\Phi'_{0,j}| = |\Phi_{0,j}|$. 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, there exists such a $k \leq m$ that[19] a split takes place in k-th node along the path segment.
3. Some equation $E_j\colon \Phi_j = \Psi_j$ 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 13, 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 the nodes with the equal labels exist along the given subpath or a split is constructed. That contradicts the choice of the subpath. □

[19] $k < m$, if $\Phi_{1,j}$ or $\Psi_{1,j}$ do not end with variables, and $k = m$ otherwise.

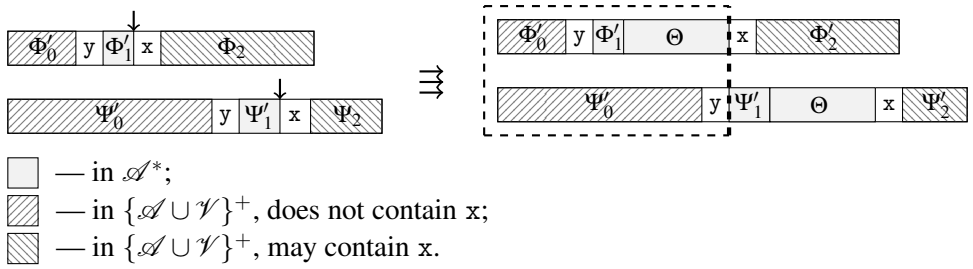


Figure 13: Splitting a strictly regular-ordered equation with repetitions resulting from substitution $\mathtt{x}\sigma = \Theta\mathtt{x}$, where $|\Theta| = |\Psi'_0| - |\Phi'_0| - |\Phi'_1|$.

8.3 Proofs and Auxiliary Propositions on One-Variable Equations

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 8. *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 $n = m + k$, $k > 0$, $\mathtt{x}\sigma_i = t_i \mathtt{x}$. The k-th unfolding step observes that the equation is either already split or is of the following form:

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

This equation has var-permutated prefixes. □

Here we repeat Lemma 3 before giving its whole proof.

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

Proof. If at least one side of the equation does not contain x, the equation always generates a finite solution tree. Thus, we consider only the following three cases.

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 8.
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 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 $m = n * i + j$, $j < n$, $\mathtt{x}\sigma = t_1 \dots t_n \mathtt{x}$, $\mathtt{x}\sigma_j = t_1 \dots t_j \mathtt{x}$, $\xi = \sigma^i \sigma_j$. Thus, σ is the composition of n elementary substitutions, and σ_j is the composition of j elementary substitutions. 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. □

8.4 Source Pseudocode of Interpreters

We recall that the expression-type variables start with x, and may be subscripted (*e.g.* $\mathtt{x_{rul}}$) or followed by other letters (*e.g.* xms), or both. The symbol-type variables are denoted with $\mathtt{s_{sym}}$, maybe subscripted. The constructor ++ is sometimes omitted, mainly in expressions enclosed in the parentheses.

The data encoding is given in Figure 8; thus, the equation variables are encoded with the two symbols enclosed in the parentheses. The parentheses are also used to form a structure of functions' arguments, *e.g.* Main takes two arguments, where the second one is a pair; the function Simplify returns a pair.

We use the following syntactic sugar in the source pseudocode of the interpreters. Variables $\mathtt{x_{num}}$ and $\mathtt{x_{freshnum}}$ range over the natural numbers. The operations +1 and -1 may be applied to these variable values instead of the corresponding arithmetic operations taking numbers given in the unary Peano system. An equation $\Phi_{\text{lhs}} = \Phi_{\text{rhs}}$ is encoded as $(\Phi_{\text{lhs}}, \Phi_{\text{rhs}})$ instead of $((\Phi_{\text{lhs}})(\Phi_{\text{rhs}}))$ (see also Figure 8). For example, the encoding of the equation $\mathbf{A}\,\mathtt{x}\,\mathtt{y} = \mathtt{x}\,\mathtt{y}\,\mathbf{A}$ is as follows: $(\mathbf{A}\,(\mathbf{V}\,\mathbf{X})\,(\mathbf{V}\,\mathbf{Y}), (\mathbf{V}\,\mathbf{X})\,(\mathbf{V}\,\mathbf{Y})\,\mathbf{A})$.

8.4.1 Basic Interpreter

$\mathtt{Go}(\mathtt{x_{rul}}, \mathtt{x_{val}}) = \mathtt{Main}(\mathtt{x_{rul}}, \mathtt{Simplify}(\varepsilon, \mathtt{x_{val}}));$

$\mathtt{Main}(\varepsilon, (\varepsilon, \varepsilon)) = \mathbf{T};$
$\mathtt{Main}(((\mathbf{V}\mathtt{s}_x) \mapsto \varepsilon) {+\!+} \mathtt{x_{rul}}, (\mathtt{x_{LHS}}, (\mathbf{V}\mathtt{s}_x) {+\!+} \mathtt{x_{RHS}}))$
$\quad = \mathtt{Main}(\mathtt{x_{rul}}, \mathtt{Simplify}(((\mathbf{V}\mathtt{s}_x) \mapsto \varepsilon), \mathtt{Subst}((\mathbf{V}\mathtt{s}_x) \mapsto \varepsilon, \varepsilon, \mathtt{x_{LHS}}), \mathtt{Subst}((\mathbf{V}\mathtt{s}_x) \mapsto \varepsilon, \varepsilon, \mathtt{x_{RHS}})));$
$\mathtt{Main}(((\mathbf{V}\mathtt{s}_x) \mapsto \varepsilon) {+\!+} \mathtt{x_{rul}}, ((\mathbf{V}\mathtt{s}_x) {+\!+} \mathtt{x_{LHS}}, \mathtt{x_{RHS}}))$
$\quad = \mathtt{Main}(\mathtt{x_{rul}}, \mathtt{Simplify}(((\mathbf{V}\mathtt{s}_x) \mapsto \varepsilon), \mathtt{Subst}((\mathbf{V}\mathtt{s}_x) \mapsto \varepsilon, \varepsilon, \mathtt{x_{LHS}}), \mathtt{Subst}((\mathbf{V}\mathtt{s}_x) \mapsto \varepsilon, \varepsilon, \mathtt{x_{RHS}})));$
$\mathtt{Main}(((\mathbf{V}\mathtt{s}_x) \mapsto \mathtt{s_{sym}}\,(\mathbf{V}\mathtt{s}_x)) {+\!+} \mathtt{x_{rul}}, ((\mathbf{V}\mathtt{s}_x) {+\!+} \mathtt{x_{LHS}}, \mathtt{s_{sym}} {+\!+} \mathtt{x_{RHS}}))$
$\quad = \mathtt{Main}(\mathtt{x_{rul}}, \mathtt{Simplify}(((\mathbf{V}\mathtt{s}_x) \mapsto \mathtt{s_{sym}}\,(\mathbf{V}\mathtt{s}_x)),$
$\qquad\qquad \mathtt{Subst}((\mathbf{V}\mathtt{s}_x) \mapsto \mathtt{s_{sym}}\,(\mathbf{V}\mathtt{s}_x), (\mathbf{V}\mathtt{x}), \mathtt{x_{LHS}}),$
$\qquad\qquad \mathtt{Subst}((\mathbf{V}\mathtt{s}_x) \mapsto \mathtt{s_{sym}}\,(\mathbf{V}\mathtt{s}_x), \varepsilon, \mathtt{x_{RHS}})));$
$\mathtt{Main}(((\mathbf{V}\mathtt{s}_x) \mapsto \mathtt{s_{sym}}\,(\mathbf{V}\mathtt{s}_x)) {+\!+} \mathtt{x_{rul}}, (\mathtt{s_{sym}} {+\!+} \mathtt{x_{LHS}}, (\mathbf{V}\mathtt{s}_x) {+\!+} \mathtt{x_{RHS}}))$
$\quad = \mathtt{Main}(\mathtt{x_{rul}}, \mathtt{Simplify}(((\mathbf{V}\mathtt{s}_x) \mapsto \mathtt{s_{sym}}\,(\mathbf{V}\mathtt{s}_x)),$
$\qquad\qquad \mathtt{Subst}((\mathbf{V}\mathtt{s}_x) \mapsto \mathtt{s_{sym}}\,(\mathbf{V}\mathtt{s}_x), \varepsilon, \mathtt{x_{LHS}}),$
$\qquad\qquad \mathtt{Subst}((\mathbf{V}\mathtt{s}_x) \mapsto \mathtt{s_{sym}}\,(\mathbf{V}\mathtt{s}_x), (\mathbf{V}\mathtt{s}_x), \mathtt{x_{RHS}})));$
$\mathtt{Main}(((\mathbf{V}\mathtt{s}_x) \mapsto (\mathbf{V}\mathtt{s}_y)\,(\mathbf{V}\mathtt{s}_x)) {+\!+} \mathtt{x_{rul}}, ((\mathbf{V}\mathtt{s}_y) {+\!+} \mathtt{x_{LHS}}, (\mathbf{V}\mathtt{s}_x) {+\!+} \mathtt{x_{RHS}}))$
$\quad = \mathtt{Main}(\mathtt{x_{rul}}, \mathtt{Simplify}(((\mathbf{V}\mathtt{s}_x) \mapsto (\mathbf{V}\mathtt{s}_y)\,(\mathbf{V}\mathtt{s}_x)),$
$\qquad\qquad \mathtt{Subst}((\mathbf{V}\mathtt{s}_x) \mapsto (\mathbf{V}\mathtt{s}_y)\,(\mathbf{V}\mathtt{s}_x), \varepsilon, \mathtt{x_{LHS}}),$
$\qquad\qquad \mathtt{Subst}((\mathbf{V}\mathtt{s}_x) \mapsto (\mathbf{V}\mathtt{s}_y)\,(\mathbf{V}\mathtt{s}_x), (\mathbf{V}\mathtt{s}_x), \mathtt{x_{RHS}})));$
$\mathtt{Main}(((\mathbf{V}\mathtt{s}_x) \mapsto (\mathbf{V}\mathtt{s}_y)\,(\mathbf{V}\mathtt{s}_x)) {+\!+} \mathtt{x_{rul}}, ((\mathbf{V}\mathtt{s}_x) {+\!+} \mathtt{x_{LHS}}, (\mathbf{V}\mathtt{s}_y) {+\!+} \mathtt{x_{RHS}}))$
$\quad = \mathtt{Main}(\mathtt{x_{rul}}, \mathtt{Simplify}(((\mathbf{V}\mathtt{s}_x) \mapsto (\mathbf{V}\mathtt{s}_y)\,(\mathbf{V}\mathtt{s}_x)),$
$\qquad\qquad \mathtt{Subst}((\mathbf{V}\mathtt{s}_x) \mapsto (\mathbf{V}\mathtt{s}_y)\,(\mathbf{V}\mathtt{s}_x), (\mathbf{V}\mathtt{s}_x), \mathtt{x_{LHS}}),$
$\qquad\qquad \mathtt{Subst}((\mathbf{V}\mathtt{s}_x) \mapsto (\mathbf{V}\mathtt{s}_y)\,(\mathbf{V}\mathtt{s}_x), \varepsilon, \mathtt{x_{RHS}})));$
$\mathtt{Main}(\mathtt{x_{rul}}, (\mathtt{x_{LHS}}, \mathtt{x_{RHS}})) = \mathbf{F};$

$/*$ The first argument of Subst is the substitution, the second one serves as an accumulator. $*/$
$\mathtt{Subst}((\mathbf{V}\mathtt{s}_x) \mapsto \mathtt{x_{value}}, \mathtt{x_{result}}, \varepsilon) = \mathtt{x_{result}};$
$\mathtt{Subst}((\mathbf{V}\mathtt{s}_x) \mapsto \mathtt{x_{value}}, \mathtt{x_{result}}, (\mathbf{V}\mathtt{s}_x) {+\!+} \mathtt{x_{expr}}) = \mathtt{Subst}((\mathbf{V}\mathtt{s}_x) \mapsto \mathtt{x_{value}}, \mathtt{x_{result}} {+\!+} \mathtt{x_{value}}, \mathtt{x_{expr}});$
$\mathtt{Subst}((\mathbf{V}\mathtt{s}_x) \mapsto \mathtt{x_{value}}, \mathtt{x_{result}}, \mathtt{s_{sym}} {+\!+} \mathtt{x_{expr}}) = \mathtt{Subst}((\mathbf{V}\mathtt{s}_x) \mapsto \mathtt{x_{value}}, \mathtt{x_{result}} {+\!+} \mathtt{s_{sym}}, \mathtt{x_{expr}});$
$\mathtt{Subst}((\mathbf{V}\mathtt{s}_x) \mapsto \mathtt{x_{value}}, \mathtt{x_{result}}, (\mathbf{V}\mathtt{s}_y) {+\!+} \mathtt{x_{expr}}) = \mathtt{Subst}((\mathbf{V}\mathtt{s}_x) \mapsto \mathtt{x_{value}}, \mathtt{x_{result}} {+\!+} (\mathbf{V}\mathtt{s}_y), \mathtt{x_{expr}});$

$\mathtt{Simplify}(\mathtt{x_{subst}}, (\mathbf{V}\mathtt{s}_x) {+\!+} \mathtt{x_{LHS}}, (\mathbf{V}\mathtt{s}_x) {+\!+} \mathtt{x_{LHS}}) = \mathtt{Simplify}(\mathtt{x_{subst}}, \mathtt{x_{LHS}}, \mathtt{x_{LHS}});$
$\mathtt{Simplify}(\mathtt{x_{subst}}, \mathtt{s_{sym}} {+\!+} \mathtt{x_{LHS}}, \mathtt{s_{sym}} {+\!+} \mathtt{x_{LHS}}) = \mathtt{Simplify}(\mathtt{x_{subst}}, \mathtt{x_{LHS}}, \mathtt{x_{LHS}});$
$\mathtt{Simplify}(\mathtt{x_{subst}}, \mathtt{s_{sym_1}} {+\!+} \mathtt{x_{LHS}}, \mathtt{s_{sym_2}} {+\!+} \mathtt{x_{LHS}}) = (\mathtt{s_{sym_1}}, \mathtt{s_{sym_2}});$
$\mathtt{Simplify}(\mathtt{x_{subst}}, \mathtt{x_{LHS}} {+\!+} (\mathbf{V}\mathtt{s}_x), \mathtt{x_{LHS}} {+\!+} (\mathbf{V}\mathtt{s}_x)) = \mathtt{Simplify}(\mathtt{x_{subst}}, \mathtt{x_{LHS}}, \mathtt{x_{LHS}});$
$\mathtt{Simplify}(\mathtt{x_{subst}}, \mathtt{x_{LHS}} {+\!+} \mathtt{s_{sym}}, \mathtt{x_{LHS}} {+\!+} \mathtt{s_{sym}}) = \mathtt{Simplify}(\mathtt{x_{subst}}, \mathtt{x_{LHS}}, \mathtt{x_{LHS}});$
$\mathtt{Simplify}(\mathtt{x_{subst}}, \mathtt{x_{LHS}} {+\!+} \mathtt{s_{sym_1}}, \mathtt{x_{LHS}} {+\!+} \mathtt{s_{sym_2}}) = (\mathtt{s_{sym_1}}, \mathtt{s_{sym_2}});$
$\mathtt{Simplify}(\mathtt{x_{subst}}, \mathtt{x_{LHS}}, \mathtt{x_{LHS}}) = (\mathtt{x_{LHS}}, \mathtt{x_{RHS}});$

8.4.2 Splitting Interpreter

This interpreter has the following refinements as compared to WIBase$_{\mathscr{L}}$. In manipulates a list of equations rather than a single equations, and uses additional simplifying functions.

- The functions `Simplify` and `CountEq` 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 process trees.
- The second argument of the function `Main` is a list of equations concatenated with the natural number $\mathtt{x}_{\mathrm{num}}$, described above.
- The function `Split` and the auxiliary multiset-handling function are added. In order to guarantee that all the equations resulting from a split are reduced, we introduce the `Reduce` function reducing a given single equation. The new function `CountEq` transforms a list of equations to a single unsatisfiable equation if at least one contradiction is found in the list, otherwise the function `CountEq` counts how many equations are included in the list.
- A number of rewriting rules marked with the corresponding comments are given in a sugared syntax. If a symbol and a variable are treated in the same way, then instead of the two (or four) rules we write the only one, where the term considered is replaced by the letter t, maybe subscripted.

$$\mathtt{Go}(\mathtt{x}_{\mathtt{rul}},\mathtt{x}_{\mathtt{val}}) = \mathtt{Main}(\mathtt{x}_{\mathtt{rul}},\mathtt{Simplify}(0,\varepsilon,\varepsilon,\mathtt{x}_{\mathtt{val}}));$$

$$\mathtt{Main}(\varepsilon,(\mathtt{x}_{\mathrm{num}})\mathtt{+\!\!+}((\varepsilon,\varepsilon))) = \mathbf{T};$$
$$\mathtt{Main}(\mathtt{x}_{\mathtt{rul}},(\mathtt{x}_{\mathrm{num}})\mathtt{+\!\!+}((\varepsilon,\varepsilon)\mathtt{+\!\!+}\mathtt{x}_{\mathtt{eqs}})) = \mathtt{Main}(\mathtt{x}_{\mathtt{rul}},(\mathtt{x}_{\mathrm{num}}-1)\mathtt{+\!\!+}\mathtt{x}_{\mathtt{eqs}});$$
$$\begin{aligned}&\mathtt{Main}(((\mathbf{Vs}_x)\mapsto\varepsilon)\mathtt{+\!\!+}\mathtt{x}_{\mathtt{rul}},(\mathtt{x}_{\mathrm{num}})\mathtt{+\!\!+}((\mathtt{x}_{\mathrm{LHS}},(\mathbf{Vs}_x)\,\mathtt{x}_{\mathrm{RHS}})\mathtt{+\!\!+}\mathtt{x}_{\mathtt{eqs}}))\\&\quad=\mathtt{Main}(\mathtt{x}_{\mathtt{rul}},\mathtt{Simplify}(\mathtt{x}_{\mathrm{num}},((\mathbf{Vs}_x)\mapsto\varepsilon),\varepsilon,\\&\qquad\qquad(\mathtt{Subst}((\mathbf{Vs}_x)\mapsto\varepsilon,\varepsilon,\mathtt{x}_{\mathrm{LHS}}),\mathtt{Subst}((\mathbf{Vs}_x)\mapsto\varepsilon,\varepsilon,\mathtt{x}_{\mathrm{RHS}}))\\&\qquad\qquad\mathtt{+\!\!+}\mathtt{SubstAll}((\mathbf{Vs}_x)\mapsto\varepsilon,\mathtt{x}_{\mathtt{eqs}})));\end{aligned}$$
$$\begin{aligned}&\mathtt{Main}(((\mathbf{Vs}_x)\mapsto\varepsilon)\mathtt{+\!\!+}\mathtt{x}_{\mathtt{rul}},(\mathtt{x}_{\mathrm{num}})\mathtt{+\!\!+}(((\mathbf{Vs}_x)\,\mathtt{x}_{\mathrm{LHS}},\mathtt{x}_{\mathrm{RHS}})\mathtt{+\!\!+}\mathtt{x}_{\mathtt{eqs}}))\\&\quad=\mathtt{Main}(\mathtt{x}_{\mathtt{rul}},\mathtt{Simplify}(\mathtt{x}_{\mathrm{num}},((\mathbf{Vs}_x)\mapsto\varepsilon),\varepsilon,\\&\qquad\qquad(\mathtt{Subst}((\mathbf{Vs}_x)\mapsto\varepsilon,\varepsilon,\mathtt{x}_{\mathrm{LHS}}),\mathtt{Subst}((\mathbf{Vs}_x)\mapsto\varepsilon,\varepsilon,\mathtt{x}_{\mathrm{RHS}}))\\&\qquad\qquad\mathtt{+\!\!+}\mathtt{SubstAll}((\mathbf{Vs}_x)\mapsto\varepsilon,\mathtt{x}_{\mathtt{eqs}})));\end{aligned}$$
$$\begin{aligned}&\mathtt{Main}(((\mathbf{Vs}_x)\mapsto\mathtt{s}_{\mathtt{sym}}(\mathbf{Vs}_x))\mathtt{+\!\!+}\mathtt{x}_{\mathtt{rul}},(\mathtt{x}_{\mathrm{num}})\mathtt{+\!\!+}(((\mathbf{Vs}_x)\,\mathtt{x}_{\mathrm{LHS}},\mathtt{s}_{\mathtt{sym}}\,\mathtt{x}_{\mathrm{RHS}})\mathtt{+\!\!+}\mathtt{x}_{\mathtt{eqs}}))\\&\quad=\mathtt{Main}(\mathtt{x}_{\mathtt{rul}},\mathtt{Simplify}(\mathtt{x}_{\mathrm{num}},((\mathbf{Vs}_x)\mapsto\mathtt{s}_{\mathtt{sym}}(\mathbf{Vs}_x)),\varepsilon,\\&\qquad\qquad(\mathtt{Subst}((\mathbf{Vs}_x)\mapsto\mathtt{s}_{\mathtt{sym}}(\mathbf{Vs}_x),(\mathbf{Vs}_x),\mathtt{x}_{\mathrm{LHS}}),\mathtt{Subst}((\mathbf{Vs}_x)\mapsto\mathtt{s}_{\mathtt{sym}}(\mathbf{Vs}_x),\varepsilon,\mathtt{x}_{\mathrm{RHS}}))\\&\qquad\qquad\mathtt{+\!\!+}\mathtt{SubstAll}((\mathbf{Vs}_x)\mapsto\mathtt{s}_{\mathtt{sym}}(\mathbf{Vs}_x),\mathtt{x}_{\mathtt{eqs}})));\end{aligned}$$
$$\begin{aligned}&\mathtt{Main}(((\mathbf{Vs}_x)\mapsto\mathtt{s}_{\mathtt{sym}}(\mathbf{Vs}_x))\mathtt{+\!\!+}\mathtt{x}_{\mathtt{rul}},(\mathtt{x}_{\mathrm{num}})\mathtt{+\!\!+}((\mathtt{s}_{\mathtt{sym}}\,\mathtt{x}_{\mathrm{LHS}},(\mathbf{Vs}_x)\,\mathtt{x}_{\mathrm{RHS}})\mathtt{+\!\!+}\mathtt{x}_{\mathtt{eqs}}))\\&\quad=\mathtt{Main}(\mathtt{x}_{\mathtt{rul}},\mathtt{Simplify}(\mathtt{x}_{\mathrm{num}},((\mathbf{Vs}_x)\mapsto\mathtt{s}_{\mathtt{sym}}(\mathbf{Vs}_x)),\varepsilon,\\&\qquad\qquad(\mathtt{Subst}((\mathbf{Vs}_x)\mapsto\mathtt{s}_{\mathtt{sym}}(\mathbf{Vs}_x),\varepsilon,\mathtt{x}_{\mathrm{LHS}}),\mathtt{Subst}((\mathbf{Vs}_x)\mapsto\mathtt{s}_{\mathtt{sym}}(\mathbf{Vs}_x),(\mathbf{Vs}_x),\mathtt{x}_{\mathrm{RHS}}))\\&\qquad\qquad\mathtt{+\!\!+}\mathtt{SubstAll}((\mathbf{Vs}_x)\mapsto\mathtt{s}_{\mathtt{sym}}(\mathbf{Vs}_x),\mathtt{x}_{\mathtt{eqs}})));\end{aligned}$$
$$\begin{aligned}&\mathtt{Main}(((\mathbf{Vs}_x)\mapsto(\mathbf{Vs}_y)\,(\mathbf{Vs}_x))\mathtt{+\!\!+}\mathtt{x}_{\mathtt{rul}},(\mathtt{x}_{\mathrm{num}})\mathtt{+\!\!+}(((\mathbf{Vs}_y)\,\mathtt{x}_{\mathrm{LHS}},(\mathbf{Vs}_x)\,\mathtt{x}_{\mathrm{RHS}})\mathtt{+\!\!+}\mathtt{x}_{\mathtt{eqs}}))\\&\quad=\mathtt{Main}(\mathtt{x}_{\mathtt{rul}},\mathtt{Simplify}(\mathtt{x}_{\mathrm{num}},((\mathbf{Vs}_x)\mapsto(\mathbf{Vs}_y)\,(\mathbf{Vs}_x)),\varepsilon,\\&\qquad\qquad(\mathtt{Subst}((\mathbf{Vs}_x)\mapsto(\mathbf{Vs}_y)\,(\mathbf{Vs}_x),\varepsilon,\mathtt{x}_{\mathrm{LHS}}),\mathtt{Subst}((\mathbf{Vs}_x)\mapsto(\mathbf{Vs}_y)\,(\mathbf{Vs}_x),(\mathbf{Vs}_x),\mathtt{x}_{\mathrm{RHS}}))\\&\qquad\qquad\mathtt{+\!\!+}\mathtt{SubstAll}((\mathbf{Vs}_x)\mapsto(\mathbf{Vs}_y)\,(\mathbf{Vs}_x),\mathtt{x}_{\mathtt{eqs}})));\end{aligned}$$
$$\begin{aligned}&\mathtt{Main}(((\mathbf{Vs}_x)\mapsto(\mathbf{Vs}_y)\,(\mathbf{Vs}_x))\mathtt{+\!\!+}\mathtt{x}_{\mathtt{rul}},(\mathtt{x}_{\mathrm{num}})\mathtt{+\!\!+}(((\mathbf{Vs}_x)\,\mathtt{x}_{\mathrm{LHS}},(\mathbf{Vs}_y)\,\mathtt{x}_{\mathrm{RHS}})\mathtt{+\!\!+}\mathtt{x}_{\mathtt{eqs}}))\\&\quad=\mathtt{Main}(\mathtt{x}_{\mathtt{rul}},\mathtt{Simplify}(\mathtt{x}_{\mathrm{num}},((\mathbf{Vs}_x)\mapsto(\mathbf{Vs}_y)\,(\mathbf{Vs}_x)),\varepsilon,\\&\qquad\qquad(\mathtt{Subst}((\mathbf{Vs}_x)\mapsto(\mathbf{Vs}_y)\,(\mathbf{Vs}_x),(\mathbf{Vs}_x),\mathtt{x}_{\mathrm{LHS}}),\mathtt{Subst}((\mathbf{Vs}_x)\mapsto(\mathbf{Vs}_y)\,(\mathbf{Vs}_x),\varepsilon,\mathtt{x}_{\mathrm{RHS}}))\\&\qquad\qquad\mathtt{+\!\!+}\mathtt{SubstAll}((\mathbf{Vs}_x)\mapsto(\mathbf{Vs}_y)\,(\mathbf{Vs}_x),\mathtt{x}_{\mathtt{eqs}})));\end{aligned}$$
$$\mathtt{Main}(\mathtt{x}_{\mathtt{rul}},(\mathtt{x}_{\mathrm{num}})\mathtt{+\!\!+}\mathtt{x}_{\mathtt{eqs}}) = \mathbf{F};$$

$$\mathtt{Subst}((\mathbf{Vs}_x)\mapsto\mathtt{x}_{\mathtt{value}},\mathtt{x}_{\mathtt{result}},\varepsilon) = \mathtt{x}_{\mathtt{result}};$$
$$\mathtt{Subst}((\mathbf{Vs}_x)\mapsto\mathtt{x}_{\mathtt{value}},\mathtt{x}_{\mathtt{result}},(\mathbf{Vs}_x)\mathtt{+\!\!+}\mathtt{x}_{\mathtt{expr}}) = \mathtt{Subst}((\mathbf{Vs}_x)\mapsto\mathtt{x}_{\mathtt{value}},\mathtt{x}_{\mathtt{result}}\mathtt{+\!\!+}\mathtt{x}_{\mathtt{value}},\mathtt{x}_{\mathtt{expr}});$$
$$\mathtt{Subst}((\mathbf{Vs}_x)\mapsto\mathtt{x}_{\mathtt{value}},\mathtt{x}_{\mathtt{result}},\mathtt{s}_{\mathtt{sym}}\mathtt{+\!\!+}\mathtt{x}_{\mathtt{expr}}) = \mathtt{Subst}((\mathbf{Vs}_x)\mapsto\mathtt{x}_{\mathtt{value}},\mathtt{x}_{\mathtt{result}}\mathtt{+\!\!+}\mathtt{s}_{\mathtt{sym}},\mathtt{x}_{\mathtt{expr}});$$
$$\mathtt{Subst}((\mathbf{Vs}_x)\mapsto\mathtt{x}_{\mathtt{value}},\mathtt{x}_{\mathtt{result}},(\mathbf{Vs}_y)\mathtt{+\!\!+}\mathtt{x}_{\mathtt{expr}}) = \mathtt{Subst}((\mathbf{Vs}_x)\mapsto\mathtt{x}_{\mathtt{value}},\mathtt{x}_{\mathtt{result}}\mathtt{+\!\!+}(\mathbf{Vs}_y),\mathtt{x}_{\mathtt{expr}});$$

$$\texttt{SubstAll}((\mathbf{V}\texttt{s}_x) \mapsto \texttt{x}_{\texttt{value}}, (\texttt{x}_{\text{LHS}}, \texttt{x}_{\text{RHS}}) +\!\!+ \texttt{x}_{\texttt{eqs}})$$
$$= (\texttt{Subst}((\mathbf{V}\texttt{s}_x) \mapsto \texttt{x}_{\texttt{value}}, \varepsilon, \texttt{x}_{\text{LHS}}), \texttt{Subst}((\mathbf{V}\texttt{s}_x) \mapsto \texttt{x}_{\texttt{value}}, \varepsilon, \texttt{x}_{\text{RHS}})) +\!\!+ \texttt{SubstAll}((\mathbf{V}\texttt{s}_x) \mapsto \texttt{x}_{\texttt{value}}, \texttt{x}_{\texttt{eqs}});$$
$$\texttt{SubstAll}((\mathbf{V}\texttt{s}_x) \mapsto \texttt{x}_{\texttt{value}}, \varepsilon) = \varepsilon;$$

$$\texttt{Simplify}(\texttt{x}_{\text{num}}, \texttt{x}_{\text{subst}}, \texttt{x}_{\texttt{result}}, ((\mathbf{V}\texttt{s}_x)\,\texttt{x}_{\text{LHS}}, (\mathbf{V}\texttt{s}_x)\,\texttt{x}_{\text{RHS}}) +\!\!+ \texttt{x}_{\texttt{eqs}})$$
$$= \texttt{Simplify}(\texttt{x}_{\text{num}}, \texttt{x}_{\text{subst}}, \texttt{x}_{\texttt{result}}, (\texttt{x}_{\text{LHS}}, \texttt{x}_{\text{RHS}}) +\!\!+ \texttt{x}_{\texttt{eqs}});$$
$$\texttt{Simplify}(\texttt{x}_{\text{num}}, \texttt{x}_{\text{subst}}, \texttt{x}_{\texttt{result}}, (\texttt{s}_{\texttt{sym}}\,\texttt{x}_{\text{LHS}}, \texttt{s}_{\texttt{sym}}\,\texttt{x}_{\text{RHS}}) +\!\!+ \texttt{x}_{\texttt{eqs}})$$
$$= \texttt{Simplify}(\texttt{x}_{\text{num}}, \texttt{x}_{\text{subst}}, \texttt{x}_{\texttt{result}}, (\texttt{x}_{\text{LHS}}, \texttt{x}_{\text{RHS}}) +\!\!+ \texttt{x}_{\texttt{eqs}});$$
$$\texttt{Simplify}(\texttt{x}_{\text{num}}, \texttt{x}_{\text{subst}}, \texttt{x}_{\texttt{result}}, (\texttt{x}_{\text{LHS}}\,(\mathbf{V}\texttt{s}_x), \texttt{x}_{\text{RHS}}\,(\mathbf{V}\texttt{s}_x)) +\!\!+ \texttt{x}_{\texttt{eqs}})$$
$$= \texttt{Simplify}(\texttt{x}_{\text{num}}, \texttt{x}_{\text{subst}}, \texttt{x}_{\texttt{result}}, (\texttt{x}_{\text{LHS}}, \texttt{x}_{\text{RHS}}) +\!\!+ \texttt{x}_{\texttt{eqs}});$$
$$\texttt{Simplify}(\texttt{x}_{\text{num}}, \texttt{x}_{\text{subst}}, \texttt{x}_{\texttt{result}}, (\texttt{x}_{\text{LHS}}\,\texttt{s}_{\texttt{sym}}, \texttt{x}_{\text{RHS}}\,\texttt{s}_{\texttt{sym}}) +\!\!+ \texttt{x}_{\texttt{eqs}})$$
$$= \texttt{Simplify}(\texttt{x}_{\text{num}}, \texttt{x}_{\text{subst}}, \texttt{x}_{\texttt{result}}, (\texttt{x}_{\text{LHS}}, \texttt{x}_{\text{RHS}}) +\!\!+ \texttt{x}_{\texttt{eqs}});$$
$$\texttt{Simplify}(\texttt{x}_{\text{num}}, \texttt{x}_{\text{subst}}, \texttt{x}_{\texttt{result}}, (\texttt{s}_{\texttt{sym}_1}\,\texttt{x}_{\text{LHS}}, \texttt{s}_{\texttt{sym}_2}\,\texttt{x}_{\text{RHS}}) +\!\!+ \texttt{x}_{\texttt{eqs}}) = (0) +\!\!+ (\texttt{s}_{\texttt{sym}_1}, \texttt{s}_{\texttt{sym}_2});$$
$$\texttt{Simplify}(\texttt{x}_{\text{num}}, \texttt{x}_{\text{subst}}, \texttt{x}_{\texttt{result}}, (\texttt{x}_{\text{LHS}}\,\texttt{s}_{\texttt{sym}_1}, \texttt{x}_{\text{RHS}}\,\texttt{s}_{\texttt{sym}_2}) +\!\!+ \texttt{x}_{\texttt{eqs}}) = (0) +\!\!+ (\texttt{s}_{\texttt{sym}_1}, \texttt{s}_{\texttt{sym}_2});$$
$$\texttt{Simplify}(\texttt{x}_{\text{num}}, \texttt{x}_{\text{subst}}, \texttt{x}_{\texttt{result}}, (\varepsilon, \varepsilon) +\!\!+ \texttt{x}_{\texttt{eqs}}) = \texttt{Simplify}(\texttt{x}_{\text{num}}, \texttt{x}_{\text{subst}}, \texttt{x}_{\texttt{result}}, \texttt{x}_{\texttt{eqs}});$$
$$\texttt{Simplify}(\texttt{x}_{\text{num}}, \texttt{x}_{\text{subst}}, \texttt{x}_{\texttt{result}}, (\texttt{x}_{\text{LHS}}, \texttt{x}_{\text{RHS}}) +\!\!+ \texttt{x}_{\texttt{eqs}})$$
$$= \texttt{Simplify}(\texttt{x}_{\text{num}}, \texttt{x}_{\text{subst}}, \texttt{x}_{\texttt{result}} +\!\!+ \texttt{Split}(\varepsilon, \mathbf{N} +\!\!+ ((\mathbf{CONST}\ 0)) +\!\!+ ((\mathbf{CONST}\ 0)), \varepsilon, \varepsilon, \texttt{x}_{\text{LHS}}, \texttt{x}_{\text{RHS}}), \texttt{x}_{\texttt{eqs}});$$
$$\texttt{Simplify}(\texttt{x}_{\text{num}}, \texttt{x}_{\text{subst}}, \texttt{x}_{\texttt{result}}, \varepsilon) = \texttt{CountEq}(\texttt{x}_{\text{num}}, 0, \texttt{x}_{\text{subst}}, \varepsilon, \texttt{x}_{\texttt{result}});$$

$$\texttt{CountEq}(\texttt{x}_{\text{num}}, \texttt{x}_{\text{freshnum}}, \texttt{x}_{\text{subst}}, \texttt{x}_{\texttt{result}}, (\texttt{s}_{\texttt{sym}_1}, \texttt{s}_{\texttt{sym}_2}) +\!\!+ \texttt{x}_{\texttt{eqs}}) = (0) +\!\!+ (\texttt{s}_{\texttt{sym}_1}, \texttt{s}_{\texttt{sym}_2});$$
$$\texttt{CountEq}(\texttt{x}_{\text{num}}, \texttt{x}_{\text{freshnum}}, \texttt{x}_{\text{subst}}, \texttt{x}_{\texttt{result}}, (\texttt{x}_{\text{eq}}) +\!\!+ \texttt{x}_{\texttt{eqs}})$$
$$= \texttt{CountEq}(\texttt{x}_{\text{num}}, \texttt{x}_{\text{freshnum}} + 1, \texttt{x}_{\text{subst}}, \texttt{x}_{\texttt{result}} +\!\!+ (\texttt{x}_{\text{eq}}), \texttt{x}_{\texttt{eqs}});$$
$$\texttt{CountEq}(\texttt{x}_{\text{num}}, \texttt{x}_{\text{freshnum}}, \texttt{x}_{\text{subst}}, \texttt{x}_{\texttt{result}}, \varepsilon) = (\texttt{x}_{\text{freshnum}}) +\!\!+ \texttt{x}_{\texttt{result}};$$

$$\texttt{Reduce}((\mathbf{V}\,\texttt{s}_x) +\!\!+ \texttt{x}_{\text{LHS}}, (\mathbf{V}\,\texttt{s}_x) +\!\!+ \texttt{x}_{\text{RHS}}) = \texttt{Reduce}(\texttt{x}_{\text{LHS}}, \texttt{x}_{\text{RHS}});$$
$$\texttt{Reduce}(\texttt{s}_{\texttt{sym}} +\!\!+ \texttt{x}_{\text{LHS}}, \texttt{s}_{\texttt{sym}} +\!\!+ \texttt{x}_{\text{RHS}}) = \texttt{Reduce}(\texttt{x}_{\text{LHS}}, \texttt{x}_{\text{RHS}});$$
$$\texttt{Reduce}(\texttt{x}_{\text{LHS}} +\!\!+ (\mathbf{V}\,\texttt{s}_x), \texttt{x}_{\text{RHS}} +\!\!+ (\mathbf{V}\,\texttt{s}_x)) = \texttt{Reduce}(\texttt{x}_{\text{LHS}}, \texttt{x}_{\text{RHS}});$$
$$\texttt{Reduce}(\texttt{x}_{\text{LHS}} +\!\!+ \texttt{s}_{\texttt{sym}}, \texttt{x}_{\text{RHS}} +\!\!+ \texttt{s}_{\texttt{sym}}) = \texttt{Reduce}(\texttt{x}_{\text{LHS}}, \texttt{x}_{\text{RHS}});$$
$$\texttt{Reduce}(\texttt{s}_1 +\!\!+ \texttt{x}_{\text{LHS}}, \texttt{s}_2 +\!\!+ \texttt{x}_{\text{RHS}}) = (\texttt{s}_1, \texttt{s}_2);$$
$$\texttt{Reduce}(\texttt{x}_{\text{LHS}} +\!\!+ \texttt{s}_1, \texttt{x}_{\text{RHS}} +\!\!+ \texttt{s}_2) = (\texttt{s}_1, \texttt{s}_2);$$
$$\texttt{Reduce}(\texttt{x}_{\text{LHS}}, \texttt{x}_{\text{RHS}}) = (\texttt{x}_{\text{LHS}}, \texttt{x}_{\text{RHS}});$$

/* Here we use a syntactic sugar : t_i denotes either an encoded variable or a symbol (one rule corresponds to the four desugared rules). The last two arguments recursively decrease. */
$$\texttt{Split}(\texttt{x}_{\texttt{result}}, \mathbf{N} +\!\!+ (\texttt{xms}_1) +\!\!+ (\texttt{xms}_2), \texttt{xpref}_{\text{LHS}}, \texttt{xpref}_{\text{RHS}}, t_1 +\!\!+ \texttt{x}_{\text{LHS}}, t_2 +\!\!+ \texttt{x}_{\text{RHS}})$$
$$= \texttt{Split}(\texttt{x}_{\texttt{result}},$$
$$\texttt{CountMS}(\texttt{Include}(t_1, \varepsilon, (\texttt{xms}_1)), \texttt{Include}(t_2, \varepsilon, (\texttt{xms}_2))), \texttt{xpref}_{\text{LHS}} +\!\!+ t_1, \texttt{xpref}_{\text{RHS}} +\!\!+ t_2, \texttt{x}_{\text{LHS}}, \texttt{x}_{\text{RHS}});$$
$$\texttt{Split}(\texttt{x}_{\texttt{result}}, \mathbf{F} +\!\!+ (\texttt{xms}_1) +\!\!+ (\texttt{xms}_2), \texttt{xpref}_{\text{LHS}}, \texttt{xpref}_{\text{RHS}}, t_1 +\!\!+ \texttt{x}_{\text{LHS}}, t_2 +\!\!+ \texttt{x}_{\text{RHS}})$$
$$= \texttt{Split}(\texttt{x}_{\texttt{result}},$$
$$\texttt{CountMS}(\texttt{Include}(t_1, \varepsilon, (\texttt{xms}_1)), \texttt{Include}(t_2, \varepsilon, (\texttt{xms}_2))), \texttt{xpref}_{\text{LHS}} +\!\!+ t_1, \texttt{xpref}_{\text{RHS}} +\!\!+ t_2, \texttt{x}_{\text{LHS}}, \texttt{x}_{\text{RHS}});$$
$$\texttt{Split}(\texttt{x}_{\texttt{result}}, \mathbf{T} +\!\!+ (\texttt{xms}_1) +\!\!+ (\texttt{xms}_2), \texttt{xpref}_{\text{LHS}}, \texttt{xpref}_{\text{RHS}}, \texttt{x}_{\text{LHS}}, \texttt{x}_{\text{RHS}})$$
$$= \texttt{Split}(\texttt{x}_{\texttt{result}} +\!\!+ (\texttt{xpref}_{\text{LHS}}, \texttt{xpref}_{\text{RHS}}), \mathbf{N} +\!\!+ ((\mathbf{CONST}\ 0)) +\!\!+ ((\mathbf{CONST}\ 0)), \varepsilon, \varepsilon, \texttt{x}_{\text{LHS}}, \texttt{x}_{\text{RHS}});$$
$$\texttt{Split}(\texttt{x}_{\texttt{result}}, \texttt{s}_{\texttt{sym}} +\!\!+ (\texttt{xms}_1) +\!\!+ (\texttt{xms}_2), \varepsilon, \varepsilon, \varepsilon, \varepsilon) = \texttt{x}_{\texttt{result}};$$
$$\texttt{Split}(\texttt{x}_{\texttt{result}}, \texttt{s}_{\texttt{sym}} +\!\!+ (\texttt{xms}_1) +\!\!+ (\texttt{xms}_2), \texttt{xpref}_{\text{LHS}}, \texttt{xpref}_{\text{RHS}}, \texttt{x}_{\text{LHS}}, \texttt{x}_{\text{RHS}})$$
$$= \texttt{Reduce}(\texttt{xpref}_{\text{LHS}} +\!\!+ \texttt{x}_{\text{LHS}}, \texttt{xpref}_{\text{RHS}} +\!\!+ \texttt{x}_{\text{RHS}}) +\!\!+ \texttt{x}_{\texttt{result}};$$

$$\texttt{Include}(\texttt{s}_{\texttt{sym}}, \texttt{x}_{\texttt{prev}}, (\texttt{xms} +\!\!+ (\mathbf{CONST}\ \texttt{x}_{\text{num}}))) = (\texttt{x}_{\texttt{prev}} +\!\!+ \texttt{xms} +\!\!+ (\mathbf{CONST}\ \texttt{x}_{\text{num}} + 1));$$
$$\texttt{Include}((\mathbf{V}\texttt{s}_x), \texttt{x}_{\texttt{prev}}, (((\mathbf{V}\texttt{s}_x)\ \texttt{x}_{\text{num}}) +\!\!+ \texttt{xms})) = (\texttt{x}_{\texttt{prev}} +\!\!+ ((\mathbf{V}\texttt{s}_x)\ \texttt{x}_{\text{num}} + 1) +\!\!+ \texttt{xms});$$
$$\texttt{Include}((\mathbf{V}\texttt{s}_x), \texttt{x}_{\texttt{prev}}, (((\mathbf{V}\texttt{s}_y)\ \texttt{x}_{\text{num}}) +\!\!+ \texttt{xms})) = \texttt{Include}((\mathbf{V}\texttt{s}_x), \texttt{x}_{\texttt{prev}} +\!\!+ ((\mathbf{V}\texttt{s}_y)\ \texttt{x}_{\text{num}}), \texttt{xms});$$
$$\texttt{Include}((\mathbf{V}\texttt{s}_x), \texttt{x}_{\texttt{prev}}, ((\mathbf{CONST}\ \texttt{x}_{\text{num}}))) = (((\mathbf{V}\texttt{s}_x)\ 1) +\!\!+ \texttt{x}_{\texttt{prev}} +\!\!+ (\mathbf{CONST}\ \texttt{x}_{\text{num}}));$$

/* Here we use a syntactic sugar : t denotes an expression enclosed in parentheses having either an encoded variable or **CONST** as the prefix (one rule corresponds to the two desugared rules). */
$$\texttt{CountMS}((t +\!\!+ \texttt{xms}_1), (\texttt{xms}_2)) = \texttt{AreEqual}(\texttt{xms}_1, \texttt{CheckElement}(t, \varepsilon, \texttt{xms}_2)) +\!\!+ (t +\!\!+ \texttt{xms}_1) +\!\!+ (\texttt{xms}_2);$$

$$\texttt{AreEqual}(\texttt{xms}_1, \texttt{xms}_2 +\!\!+ \mathbf{F}) = \mathbf{F};$$
$$\texttt{AreEqual}(\varepsilon, \varepsilon) = \mathbf{T};$$
$$\texttt{AreEqual}(\texttt{xms}_1, \varepsilon) = \mathbf{F};$$
$$\texttt{AreEqual}(\varepsilon, \texttt{xms}_2) = \mathbf{F};$$
$$\texttt{AreEqual}(((\mathbf{Vs}_{\texttt{sym}})\, \texttt{x}_{\text{num}}) +\!\!+ \texttt{xms}_1, \texttt{xms}_2) = \texttt{AreEqual}(\texttt{xms}_1, \texttt{CheckElement}(((\mathbf{Vs}_{\texttt{sym}})\, \texttt{x}_{\text{num}}), \varepsilon, \texttt{xms}_2));$$
$$\texttt{AreEqual}((\mathbf{CONST}\ \texttt{x}_{\text{num}}) +\!\!+ \texttt{xms}_1, \texttt{xms}_2) = \texttt{AreEqual}(\texttt{xms}_1, \texttt{CheckElement}((\mathbf{CONST}\ \texttt{x}_{\text{num}}), \varepsilon, \texttt{xms}_2));$$

$$\texttt{CheckElement}((\mathbf{CONST}\ 0), \varepsilon, \texttt{xms} +\!\!+ (\mathbf{CONST}\ 0)) = \texttt{xms};$$
$$\texttt{CheckElement}((\mathbf{CONST}\ \texttt{x}_{\text{num}1} + 1), \varepsilon, \texttt{xms} +\!\!+ (\mathbf{CONST}\ \texttt{x}_{\text{num}2} + 1))$$
$$= \texttt{CheckElement}((\mathbf{CONST}\ \texttt{x}_{\text{num}1}), \varepsilon, \texttt{xms} +\!\!+ (\mathbf{CONST}\ \texttt{x}_{\text{num}2}));$$
$$\texttt{CheckElement}(((\mathbf{Vs}_x)\ 0), \texttt{x}_{\texttt{prev}}, ((\mathbf{Vs}_x)\ 0) +\!\!+ \texttt{xms}) = \texttt{x}_{\texttt{prev}} +\!\!+ \texttt{xms};$$
$$\texttt{CheckElement}(((\mathbf{Vs}_x)\ \texttt{x}_{\text{num}1} + 1), \texttt{x}_{\texttt{prev}}, ((\mathbf{Vs}_x)\ \texttt{x}_{\text{num}2} + 1) +\!\!+ \texttt{xms})$$
$$= \texttt{CheckElement}(((\mathbf{Vs}_x)\ \texttt{x}_{\text{num}1}), \texttt{x}_{\texttt{prev}}, ((\mathbf{Vs}_x)\ \texttt{x}_{\text{num}2}) +\!\!+ \texttt{xms});$$
$$\texttt{CheckElement}(((\mathbf{Vs}_x)\ \texttt{x}_{\text{num}1}), \texttt{x}_{\texttt{prev}}, ((\mathbf{Vs}_x)\ \texttt{x}_{\text{num}2}) +\!\!+ \texttt{xms}) = \mathbf{F};$$
$$\texttt{CheckElement}(((\mathbf{Vs}_x)\ \texttt{x}_{\text{num}1}), \texttt{x}_{\texttt{prev}}, ((\mathbf{Vs}_y)\ \texttt{x}_{\text{num}2}) +\!\!+ \texttt{xms})$$
$$= \texttt{CheckElement}(((\mathbf{Vs}_x)\ \texttt{xnum}_1), \texttt{x}_{\texttt{prev}} +\!\!+ ((\mathbf{Vs}_y)\ \texttt{x}_{\text{num}2}), \texttt{xms});$$
$$\texttt{CheckElement}(((\mathbf{Vs}_x)\ \texttt{x}_{\text{num}1}), \texttt{x}_{\texttt{prev}}, (\mathbf{CONST}\ \texttt{x}_{\text{num}2})) = \mathbf{F};$$

8.4.3 Counting Interpreter

This interpreter uses the function definitions given for the interpreter WISplit$_{\mathscr{L}}$ plus some additional functions, provided that the function `CheckElement` is modified and the last rule of the function `Split` is replaced with the following rewriting rule.

$$/* \text{This rule replaces the last rule of } \texttt{Split} \text{ definition given in WISplit}_{\mathscr{L}}. */$$
$$\texttt{Split}(\texttt{x}_{\texttt{result}}, \texttt{s}_{\texttt{sym}} +\!\!+ (\texttt{xms}_1) +\!\!+ (\texttt{xms}_2), \texttt{xpref}_{\text{LHS}}, \texttt{xpref}_{\text{RHS}}, \texttt{x}_{\text{LHS}}, \texttt{x}_{\text{RHS}})$$
$$= \texttt{SplitR}(\texttt{x}_{\texttt{result}}, \mathbf{N} +\!\!+ ((\mathbf{CONST}\ 0)) +\!\!+ ((\mathbf{CONST}\ 0)), \varepsilon, \varepsilon, \texttt{Reduce}(\texttt{xpref}_{\text{LHS}} +\!\!+ \texttt{x}_{\text{LHS}}, \texttt{xpref}_{\text{RHS}} +\!\!+ \texttt{x}_{\text{RHS}}));$$

The additional function definitions are given below. The following definition replaces the version of the `CheckElement` source code given in WISplit$_{\mathscr{L}}$ source code.

$$\texttt{CheckElement}((\mathbf{CONST}\ \texttt{x}_{\text{num}1}), \varepsilon, \texttt{xms} +\!\!+ (\mathbf{CONST}\ \texttt{x}_{\text{num}2})) = \texttt{xms} +\!\!+ \texttt{CmpNumbers}(\texttt{x}_{\text{num}1}, \texttt{x}_{\text{num}2});$$
$$\texttt{CheckElement}(((\mathbf{Vs}_{\texttt{sym}})\ \texttt{x}_{\text{num}1}), \texttt{x}_{\texttt{prev}}, ((\mathbf{Vs}_{\texttt{sym}})\ \texttt{x}_{\text{num}2}) +\!\!+ \texttt{xms}) = \texttt{x}_{\texttt{prev}} +\!\!+ \texttt{xms} +\!\!+ \texttt{CmpNumbers}(\texttt{x}_{\text{num}1}, \texttt{x}_{\text{num}2});$$
$$\texttt{CheckElement}(((\mathbf{Vs}_{\texttt{sym}})\ \texttt{x}_{\text{num}1}), \texttt{x}_{\texttt{prev}}, (\mathbf{CONST}\ \texttt{x}_{\text{num}2})) = \mathbf{G} +\!\!+ \mathbf{F};$$
$$\texttt{CheckElement}(((\mathbf{Vs}_{\texttt{sym}_1})\ \texttt{x}_{\text{num}1}), \texttt{x}_{\texttt{prev}}, ((\mathbf{Vs}_{\texttt{sym}_2})\ \texttt{x}_{\text{num}2}) +\!\!+ \texttt{xms})$$
$$= \texttt{CheckElement}(((\mathbf{Vs}_{\texttt{sym}_1})\ \texttt{x}_{\text{num}1}), \texttt{x}_{\texttt{prev}} +\!\!+ ((\mathbf{Vs}_{\texttt{sym}_2})\ \texttt{x}_{\text{num}2}), \texttt{xms});$$

$$\texttt{CmpNumbers}(0, 0) = \varepsilon;$$
$$\texttt{CmpNumbers}(\texttt{x}_{\text{num}1} + 1, \texttt{x}_{\text{num}2} + 1) = \texttt{CmpNumbers}(\texttt{x}_{\text{num}1}, \texttt{x}_{\text{num}2});$$
$$\texttt{CmpNumbers}(\varepsilon, \texttt{x}_{\text{num}}) = \mathbf{L} +\!\!+ \mathbf{F};$$
$$\texttt{CmpNumbers}(\texttt{x}_{\text{num}}, \varepsilon) = \mathbf{G} +\!\!+ \mathbf{F};$$

$$/* \text{ Here we use a syntactic sugar : } t_i \text{ denotes either an encoded variable or a symbol (one rule corresponds to}$$
$$\text{the four desugared rules). The last two arguments recursively decrease. } */$$
$$\texttt{SplitR}(\texttt{x}_{\texttt{result}}, \texttt{s}_{\texttt{sym}} +\!\!+ (\texttt{xms}_1) +\!\!+ (\texttt{xms}_2), \varepsilon, \varepsilon, \varepsilon, \varepsilon) = \texttt{x}_{\texttt{result}};$$
$$\texttt{SplitR}(\texttt{x}_{\texttt{result}}, \mathbf{N} +\!\!+ (\texttt{xms}_1) +\!\!+ (\texttt{xms}_2), \texttt{xsuff}_{\text{LHS}}, \texttt{xsuff}_{\text{RHS}}, \texttt{x}_{\text{LHS}} +\!\!+ t_1, \texttt{x}_{\text{RHS}} +\!\!+ t_2)$$
$$= \texttt{SplitR}(\texttt{x}_{\texttt{result}},$$
$$\texttt{CountMS}(\texttt{Include}(t_1, \varepsilon, (\texttt{xms}_1)), \texttt{Include}(t_2, \varepsilon, (\texttt{xms}_2))), t_1 +\!\!+ \texttt{xsuff}_{\text{LHS}}, t_2 +\!\!+ \texttt{xsuff}_{\text{RHS}}, \texttt{x}_{\text{LHS}}, \texttt{x}_{\text{RHS}});$$

$$\texttt{SplitR}(\texttt{x}_{\texttt{result}}, \mathbf{F} +\!\!+ (\texttt{xms}_1) +\!\!+ (\texttt{xms}_2), \texttt{xsuff}_{\text{LHS}}, \texttt{xsuff}_{\text{RHS}}, \texttt{x}_{\text{LHS}} +\!\!+ t_1, \texttt{x}_{\text{RHS}} +\!\!+ t_2)$$
$$= \texttt{SplitR}(\texttt{x}_{\texttt{result}},$$
$$\texttt{CountMS}(\texttt{Include}(t_1, \varepsilon, (\texttt{xms}_1)), \texttt{Include}(t_2, \varepsilon, (\texttt{xms}_2))), t_1 +\!\!+ \texttt{xsuff}_{\text{LHS}}, t_2 +\!\!+ \texttt{xsuff}_{\text{RHS}}, \texttt{x}_{\text{LHS}}, \texttt{x}_{\text{RHS}});$$

$$\texttt{SplitR}(\texttt{x}_{\texttt{result}}, \mathbf{T} +\!\!+ (\texttt{xms}_1) +\!\!+ (\texttt{xms}_2), \texttt{xsuff}_{\text{LHS}}, \texttt{xsuff}_{\text{RHS}}, \texttt{x}_{\text{LHS}}, \texttt{x}_{\text{RHS}})$$
$$= \texttt{SplitR}(\texttt{x}_{\texttt{result}} +\!\!+ (\texttt{xsuff}_{\text{LHS}}, \texttt{xsuff}_{\text{RHS}}), \mathbf{N} +\!\!+ (\mathbf{CONST}\ 0) +\!\!+ (\mathbf{CONST}\ 0), \varepsilon, \varepsilon, \texttt{x}_{\text{LHS}}, \texttt{x}_{\text{RHS}});$$
$$\texttt{SplitR}(\texttt{x}_{\texttt{result}}, \texttt{s}_{\texttt{sym}} +\!\!+ (\texttt{xms}_1) +\!\!+ (\texttt{xms}_2), \texttt{xsuff}_{\text{LHS}}, \texttt{xsuff}_{\text{RHS}}, \texttt{x}_{\text{LHS}}, \texttt{x}_{\text{RHS}})$$
$$= \texttt{CheckLengths}(\texttt{YieldCheck}(\texttt{AddExprMS}(\texttt{x}_{\text{LHS}}, (\texttt{xms}_1)), \texttt{AddExprMS}(\texttt{x}_{\text{RHS}}, (\texttt{xms}_2))),$$
$$\texttt{x}_{\text{LHS}} +\!\!+ \texttt{xsuff}_{\text{LHS}}, \texttt{x}_{\text{RHS}} +\!\!+ \texttt{xsuff}_{\text{RHS}}) +\!\!+ \texttt{x}_{\texttt{result}};$$

$$\texttt{CheckLengths}(\mathbf{F}, \texttt{x}_{\text{LHS}}, \texttt{x}_{\text{RHS}}) = (\texttt{x}_{\text{LHS}}, \texttt{x}_{\text{RHS}});$$
$$\texttt{CheckLengths}(\mathbf{T}, \texttt{x}_{\text{LHS}}, \texttt{x}_{\text{RHS}}) = (\mathbf{A}, \mathbf{B});$$

$$\texttt{CheckInclusion}(\mathbf{G}, \texttt{xms}_1, \varepsilon) = \mathbf{T};$$
$$\texttt{CheckInclusion}(\mathbf{L}, \varepsilon, \texttt{xms}_2) = \mathbf{T};$$
$$\texttt{CheckInclusion}(\mathbf{G}, ((\mathbf{Vs}_{\texttt{sym}})\ \texttt{x}_{\text{num}}) +\!\!+ \texttt{xms}_1, \texttt{xms}_2) = \texttt{CheckInfo}(\mathbf{G}, \texttt{CheckElement}(((\mathbf{Vs}_{\texttt{sym}})\ \texttt{x}_{\text{num}}), \varepsilon, \texttt{xms}_2), \texttt{xms}_1);$$
$$\texttt{CheckInclusion}(\mathbf{G}, (\mathbf{CONST}\ \texttt{x}_{\text{num}}) +\!\!+ \texttt{xms}_1, \texttt{xms}_2) = \texttt{CheckInfo}(\mathbf{G}, \texttt{CheckElement}((\mathbf{CONST}\ \texttt{x}_{\text{num}}), \varepsilon, \texttt{xms}_2), \texttt{xms}_1);$$
$$\texttt{CheckInclusion}(\mathbf{L}, ((\mathbf{Vs}_{\texttt{sym}})\ \texttt{x}_{\text{num}}) +\!\!+ \texttt{xms}_1, \texttt{xms}_2) = \texttt{CheckInfo}(\mathbf{L}, \texttt{CheckElement}(((\mathbf{Vs}_{\texttt{sym}})\ \texttt{x}_{\text{num}}), \varepsilon, \texttt{xms}_2), \texttt{xms}_1);$$
$$\texttt{CheckInclusion}(\mathbf{L}, (\mathbf{CONST}\ \texttt{x}_{\text{num}}) +\!\!+ \texttt{xms}_1, \texttt{xms}_2) = \texttt{CheckInfo}(\mathbf{L}, \texttt{CheckElement}((\mathbf{CONST}\ \texttt{x}_{\text{num}}), \varepsilon, \texttt{xms}_2), \texttt{xms}_1);$$
$$\texttt{CheckInclusion}(\texttt{s}_{\texttt{sym}}, \texttt{xms}_1, \texttt{xms}_2) = \mathbf{F};$$

$$\texttt{CheckInfo}(\texttt{s}_{\texttt{sym}}, \texttt{xms}_2 +\!\!+ \texttt{s}_{\texttt{sym}} +\!\!+ \mathbf{F}, \texttt{xms}_1) = \texttt{CheckInclusion}(\texttt{s}_{\texttt{sym}}, \texttt{xms}_1, \texttt{xms}_2);$$
$$\texttt{CheckInfo}(\mathbf{G}, \texttt{xms}_2 +\!\!+ \mathbf{L} +\!\!+ \mathbf{F}, \texttt{xms}_1) = \mathbf{F};$$
$$\texttt{CheckInfo}(\mathbf{L}, \texttt{xms}_2 +\!\!+ \mathbf{G} +\!\!+ \mathbf{F}, \texttt{xms}_1) = \mathbf{F};$$
$$\texttt{CheckInfo}(\texttt{s}_{\texttt{sym}}, \texttt{xms}_2, \texttt{xms}_1) = \texttt{CheckInclusion}(\texttt{s}_{\texttt{sym}}, \texttt{xms}_1, \texttt{xms}_2);$$

$$\texttt{YieldCheck}(\texttt{xms}_1 +\!\!+ (\mathbf{CONST}\ \texttt{xnum}_1), \texttt{xms}_2 +\!\!+ (\mathbf{CONST}\ \texttt{xnum}_2))$$
$$= \texttt{YieldCheckAux}(\texttt{CheckElement}((\mathbf{CONST}\ \texttt{xnum}_1), \varepsilon, (\mathbf{CONST}\ \texttt{xnum}_2)), \texttt{xms}_1, \texttt{xms}_2);$$

$$\texttt{YieldCheckAux}(\varepsilon, \texttt{xms}_1, \texttt{xms}_2) = \mathbf{F};$$
$$\texttt{YieldCheckAux}(\texttt{xms} +\!\!+ \texttt{s}_{\texttt{sym}} +\!\!+ \mathbf{F}, \texttt{xms}_1, \texttt{xms}_2) = \texttt{CheckInclusion}(\texttt{s}_{\texttt{sym}}, \texttt{xms}_1, \texttt{xms}_2);$$

$$\texttt{AddExprMS}((\mathbf{Vs}_{\texttt{sym}}) +\!\!+ \texttt{x}_{\texttt{expr}}, \texttt{xms}) = \texttt{AddExprMS}(\texttt{x}_{\texttt{expr}}, \texttt{Include}((\mathbf{Vs}_{\texttt{sym}}), \varepsilon, \texttt{xms}));$$
$$\texttt{AddExprMS}(\texttt{s}_{\texttt{sym}} +\!\!+ \texttt{x}_{\texttt{expr}}, \texttt{xms}) = \texttt{AddExprMS}(\texttt{x}_{\texttt{expr}}, \texttt{Include}(\texttt{s}_{\texttt{sym}}, \varepsilon, \texttt{xms}));$$
$$\texttt{AddExprMS}(\varepsilon, (\texttt{xms})) = \texttt{xms};$$