Program Specialization as a Tool for Solving Word Equations

Antonina Nepeivoda
Program Systems Institute of RAS

Verification and Program Transformation 2021
March 28th

A satisfiability problem

Given a word equation system $\mathcal{E}qs$, is there a sequence σ of variable narrowings leading to a solution of $\mathcal{E}qs$?

The main contribution

A method for solving word equation satisfiability problem by means of a program specialization, reducing **the satisfiability problem to a (un)reachability problem.**

We show our method works for some the word equations sets for which equations `Z3Str3` and `CVC4` are not able to solve the problem:

- by demonstrating the benchmark experiments using Turchin's supercompilation;
- by proving that the corresponding specialization tasks terminate.

Word equations

Definition

Given a constant alphabet Σ and a variable set $\mathcal{V}$, *a word equation* is an equation $\Phi = \Psi$, where $\Phi, \Psi \in \{\Sigma \cup \mathcal{V}\}^*$. A solution to the word equation is a substitution $\sigma : \mathcal{V} \to \Sigma^*$ s.t. $\Phi\sigma$ textually coincides with $\Psi\sigma$.

Let E be $\mathtt{x}\mathsf{AB} = \mathsf{BA}\mathtt{x}$, where $\mathsf{A}, \mathsf{B} \in \Sigma$, $\mathtt{x} \in \mathcal{V}$. Consider the sequence $\sigma_1 : \mathtt{x} \to \mathsf{B}\mathtt{x}$, $\sigma_2 : \mathtt{x} \to \varepsilon$. Then $\sigma_2 \circ \sigma_1 : \mathtt{x} \to \mathsf{B}$ is a solution to E: $(\mathtt{x}\mathsf{AB})\sigma_1\sigma_2 = \mathsf{BAB} = (\mathsf{BA}\mathtt{x})\sigma_1\sigma_2$.

The history of the word equations

In theory:

- Algorithms for solving the quadratic (e.g. $\mathrm{x}\,\mathrm{A}\,\mathrm{y} = \mathrm{y}\,\mathrm{A}\,\mathrm{x}$) and one-variable word equations (Matiyasevich, 1965)
- An algorithm for solving the three-variable word equations (Hmelevskij, 1971)
- An algorithm for solving the word equations in the general case (Makanin, 1977)
- More efficient (but still worst-case doubly-exponential) algorithms (Plandowski, 2006, Jez, 2016)

The history of the word equations

In practice:

- efficient algorithms for solving the straight-line (e.g. $\mathrm{xxx} = \mathrm{yAz}$) word equations (Rümmer et al., 2014–...)
- algorithms for solving the quadratic word equations (Le et al., Lin et al., 2018)
- algorithms for solving the word equations in the case when the solution lengths are bounded (Bjørner, 2009–..., Day, 2019)

Our contribution

Our method can solve equations in some classes, in which variables may occur on the both sides and more than twice.

- One-variable word equations
- Regular-ordered word equations with repetitions:

The solvers `CVC4` and `Z3Str3` do not terminate on the equation $\mathrm{AB}\mathtt{xxyy} = \mathtt{xxyy}\mathrm{BA}$ which belongs to the second class and is solvable by our method.

Encoded word equations

Definition

The set of encoded word equations Eqs is as follows.

Eqs ::= Eq Eqs | ε

Eq ::= (Side, Side)

Side ::= Char Side | Var Side | ε

There Var $\in \mathcal{V}$, Char $\in \Sigma$, ε is the empty word.

As a sugar, we write the encoded equation (LHS, RHS) as

$$\mathrm{LHS} = \mathrm{RHS};$$

and the sequence $(\mathrm{LHS}_1, \mathrm{RHS}_1) \ldots (\mathrm{LHS}_n, \mathrm{RHS}_n)$ as

$$\langle\ \mathrm{LHS}_i = \mathrm{RHS}_i\ \rangle_{i=1}^{n}.$$

A simple logic programming language $\mathscr{L}$

Definition

A (finite) narrowings sequence Narrs is defined as follows.

$$\text{Narrs} ::= (\text{Narr})\ \text{Narrs} \mid \varepsilon$$

$$\text{Narr} ::= {}'\text{Var} \to \text{Char}\ \text{Var}' \mid {}'\text{Var} \to \text{Var}_1\,\text{Var}' \mid {}'\text{Var} \to \varepsilon'$$

There Var, $\text{Var}_1 \in \mathcal{V}$, $\text{Char} \in \Sigma$, $\text{Var} \neq \text{Var}_1$.

Every narrowings sequence belonging to Narrs defines a substitution $\sigma : \mathcal{V} \to (\mathcal{V} \cup \Sigma)^*$. Given $\mathrm{x} \in \mathcal{V}$, σ is either $\mathrm{x} \to \Phi$ or $\mathrm{x} \to \Phi\mathrm{x}$ where Φ does not contain x.

We consider a set of Narrs sequences as a simple acyclic logic programming language $\mathscr{L}$ over the data Eqs.

A simple logic programming language $\mathscr{L}$

Definition

A (finite) narrowings sequence Narrs is defined as follows.

$$\text{Narrs} ::= (\text{Narr})\ \text{Narrs} \mid \varepsilon$$

$$\text{Narr} ::= \text{'Var} \to \text{Char Var'} \mid \text{'Var} \to \text{Var}_1\,\text{Var'} \mid \text{'Var} \to \varepsilon\text{'}$$

Compatibility of the narrowings with $\langle \Phi_1 = \Psi_1, \ldots, \Phi_n = \Psi_n \rangle$:

$'\mathtt{x} \to \varepsilon'$	$\mathtt{x}\,\Phi_1 = \Psi_1$ or $\Phi_1 = \mathtt{x}\Psi_1$	$'\mathtt{x} \to \mathtt{t}\mathtt{x}'$	$\mathtt{x}\,\Phi_1 = \mathtt{t}\Psi_1$ or $\mathtt{t}\,\Phi_1 = \mathtt{x}\Psi_1$
$'\mathtt{x} \to \mathtt{x}_1\mathtt{x}'$	$\mathtt{x}\,\Phi_1 = \mathtt{x}_1\Psi_1$ or $\mathtt{x}_1\Phi_1 = \mathtt{x}\Psi_1$		

We consider a set of Narrs sequences as a simple acyclic logic programming language $\mathscr{L}$ over the data Eqs.

Operational semantics of $\mathscr{L}$

An $\mathscr{L}$ interpreter $\mathsf{WI}_{\mathscr{L}}$ takes a finite sequence $(\sigma_1)(\sigma_2)...(\sigma_n)$ and a datum $\langle \Phi_i = \Psi_i \rangle_{i=1}^{m}$.

The call $\mathsf{WI}_{\mathscr{L}}((\sigma_1)(\sigma_2)...(\sigma_n), \langle \Phi_i = \Psi_i \rangle_{i=1}^{m})$ returns T iff $\forall i,\ 1 \leqslant i \leqslant m\,(\Phi_i \sigma_1 ... \sigma_n = \Psi_i \sigma_1 ... \sigma_n)$, and F otherwise.

Given a sequence of n narrowings, the interpreter $\mathsf{WI}_{\mathscr{L}}$:

- does at most n steps (i.e. always terminates);
- for all equation lists $\langle \Phi_i = \Psi_i \rangle_{i=1}^{m}$ returns either T or F, hence $\mathsf{WI}_{\mathscr{L}}$ never falls in deadlock.

Specialization of $\mathscr{L}$-interpreters

Given the call $\mathsf{WI}_{\mathscr{L}}(\mathrm{P}, \langle \Phi_i = \Psi_i \rangle_{i=1}^{n})$, we replace the $\mathscr{L}$-program P with a parameter $\mathcal{P}$ ranging over $\mathscr{L}$-programs. Thus, the specialization task is as follows.

$$\mathsf{WI}_{\mathscr{L}}(\mathcal{P}, \langle \Phi_i = \Psi_i \rangle_{i=1}^{n})$$

The unfolding of this initial configuration results in a possibly infinite tree: a description of the runs of **all** possible $\mathscr{L}$-programs on $\langle \Phi_i = \Psi_i \rangle_{i=1}^{n}$.

- The program lengths are unknown $\Rightarrow$ runs are described by means of graphs, which may contain loops.
- Most of the programs return F.

The verification task

Consider the following verification task over the $\mathscr{L}$ programs.

Given a word equation system $\mathcal{E}qs$, we say that the verification task succeeds iff $\mathcal{E}qs$ has solutions if and only if the residual program generated by specialization of $\mathsf{WI}_{\mathscr{L}}(\mathcal{P}, \mathcal{E}qs)$ contains a function returning T.

We do not require the specialization to terminate for every system $\mathcal{E}qs$.

Syntax of the residual programs

Syntax definition

Program ::= Rule; Program | ε

Rule ::= Name(Pattern) = Expression

Pattern ::= (Narr) | (Narr) ++ Pattern | p | ε

Expression ::= T | F | Name(p)

There p is a variable ranging over Narrs, Name is a function name.

Every function definition contains a single argument, and the only variable occurring at most once in its left- and right-hand sides is p.

Examples

Given the equation $Ax = xA$, a specializer produces the following residual program, where the entry point is $\mathtt{F}(\mathrm{p})$, and p is a variable ranging over Narrs.

$$\begin{aligned}
&\mathtt{F}((\text{'}\mathrm{x} \to \varepsilon\text{'})) = \mathsf{T}\\
&\mathtt{F}((\text{'}\mathrm{x} \to \mathsf{A}\mathrm{x}\text{'}) \mathbin{{+}{+}} \mathrm{p}) = \mathtt{F}(\mathrm{p})\\
&\mathtt{F}(\mathrm{p}) = \mathsf{F}
\end{aligned}$$

Given the equation $Ax = xB$, the residual program is as follows, where the entry point is $\mathtt{G}(\mathrm{p})$.

$$\begin{aligned}
&\mathtt{G}((\text{'}\mathrm{x} \to \varepsilon\text{'})) = \mathsf{F}\\
&\mathtt{G}((\text{'}\mathrm{x} \to \mathsf{A}\mathrm{x}\text{'}) \mathbin{{+}{+}} \mathrm{p}) = \mathtt{G}(\mathrm{p})\\
&\mathtt{G}(\mathrm{p}) = \mathsf{F}
\end{aligned}$$

The general interpreters' structure

The main loop

`Main` function	1. Take the first program rule
`Subst` function	2. Apply (substitute)
`Smpl` function	3. Simplify the result

The function `Smpl` varies in the different interpreters.

- `Smpl` takes a constant equation list and returns a constant equation list with the same set of solutions.
- `Smpl` terminates on every constant equation list.

Basic interpreter WIBase$_{\mathscr{L}}$

Structure of `Smpl` function

$\boxed{\text{Reduce prefixes}} \rightarrow \boxed{\text{Reduce suffixes}}$

Further we refer to this simplification operation as Reduce.

Input format

$\mathtt{p}$ — ranges over sequences of the rules;
$\mathcal{E}qs$ — ranges over equations.

$$\mathtt{Go}(\mathtt{p}, \mathcal{E}qs) = \mathtt{Main}(\mathtt{p}, \mathtt{Smpl}(\mathcal{E}qs));$$

- Specialization of the scheme WIBase$_{\mathscr{L}}(\mathcal{P}, \Phi = \Psi)$ successfully solves all the quadratic equations $\Phi = \Psi$ (e.g. $\mathtt{x}\mathrm{AB}\mathtt{y} = \mathtt{y}\mathrm{BA}\mathtt{x}$).

Splitting interpreter $\mathrm{WISplit}_{\mathscr{L}}$

Structure of `Smpl` function

$$\boxed{\text{Reduce}} \rightarrow \boxed{\text{Left-split}} \rightarrow \boxed{\text{Reduce}}$$

Input format

$$\mathtt{Go}(\mathrm{p}, \mathcal{E}qs) = \mathtt{Main}(\mathrm{p}, \mathtt{Smpl}(0, \mathcal{E}qs));$$

- The first argument of `Smpl` (initially valued 0) is added to prevent an unwanted folding.
- Specialization of the scheme $\mathrm{WISplit}_{\mathscr{L}}(\mathcal{P}, \langle \Phi = \Psi \rangle)$ successfully solves every regular-ordered equation with var-repetitions $\Phi = \Psi$ (e.g. $\mathtt{xx}\mathrm{AB} = \mathrm{BA}\mathtt{xx}$).

Counting interpreter $\mathrm{WICount}_{\mathscr{L}}$

Finds contradictions, comparing variables and constants multisets in the left- and right-hand equation sides.

Structure of `Smpl` function

$$\boxed{\text{Reduce}} \rightarrow \boxed{\text{Left-split}} \rightarrow \boxed{\text{Right-split}} \rightarrow \boxed{\text{Reduce}} \rightarrow \boxed{\text{Count}}$$

Input format

$$\mathtt{Go}(\mathtt{p}, \mathcal{E}qs) = \mathtt{Main}(\mathtt{p}, \mathtt{Smpl}(0, \mathcal{E}qs));$$

- Specialization of $\mathrm{WICount}_{\mathscr{L}}(\mathcal{P}, \langle \Phi = \Psi \rangle)$ successfully solves every one-variable word equation $\Phi = \Psi$.

Optimality lemma

Lemma

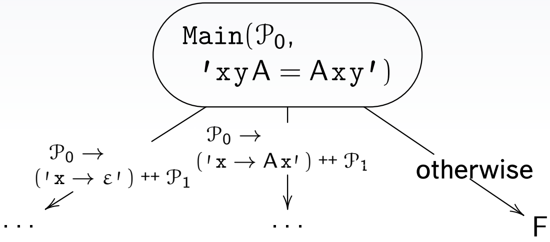
All the folding operations in the process graph of $\mathsf{WI}_{\mathscr{L}}(\mathcal{P}, \langle \Phi_i = \Psi_i \rangle_{i=1}^{n})$ occur only on the pairs of the configurations:

$$\mathtt{Main}(\mathcal{P}_j, \langle \Phi_i^j = \Psi_i^j \rangle_{i=1}^{n_j})$$

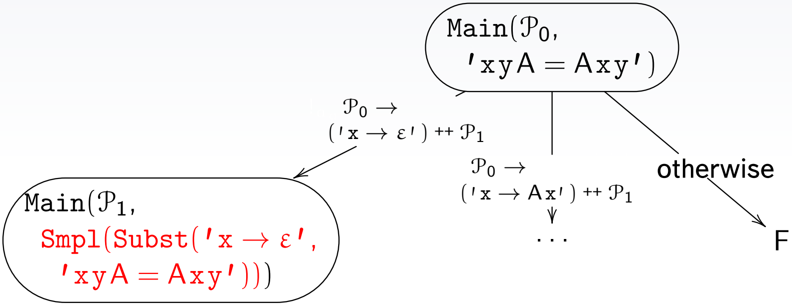
where $\mathcal{P}_j$ is a parameter, and the equation system does not contain parameters.

The lemma implies a mapping between the process graph of $\mathsf{WI}_{\mathscr{L}}(\mathcal{P}, \langle \Phi_i = \Psi_i \rangle_{i=1}^{n})$ and the solution graph of the equation list $\langle \Phi_i = \Psi_i \rangle_{i=1}^{n}$.

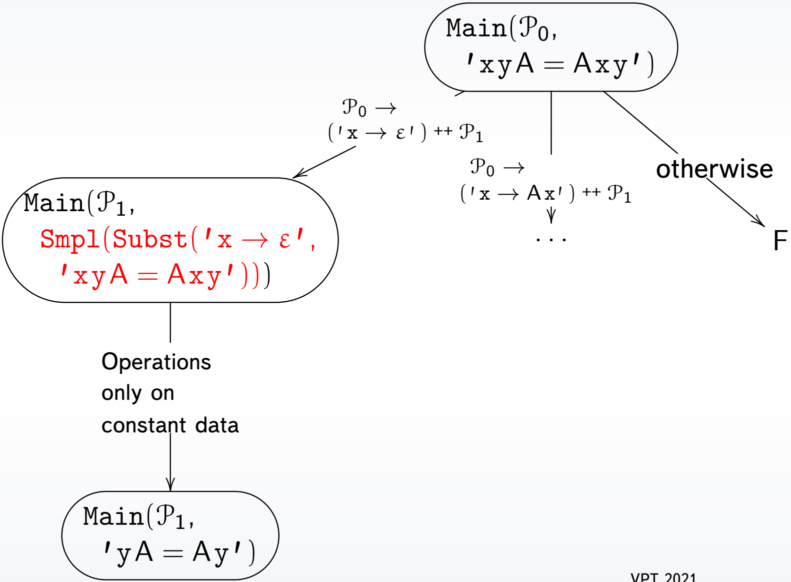
Generating the narrowings



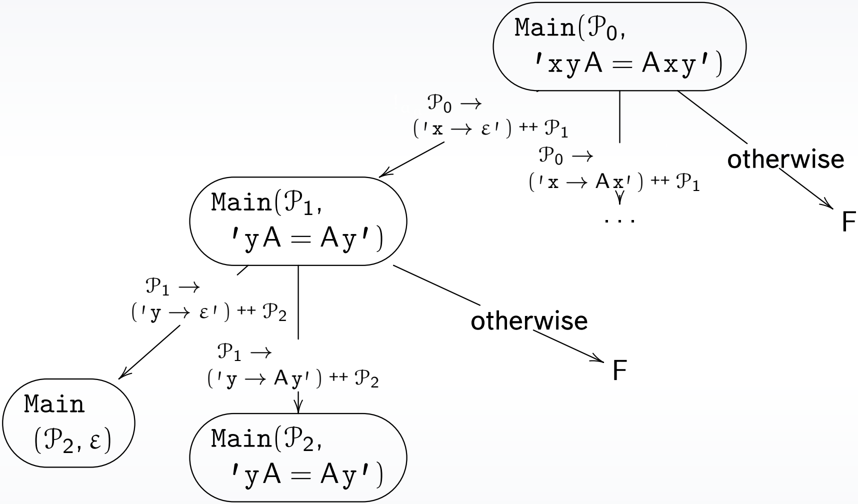
Generating the new configuration



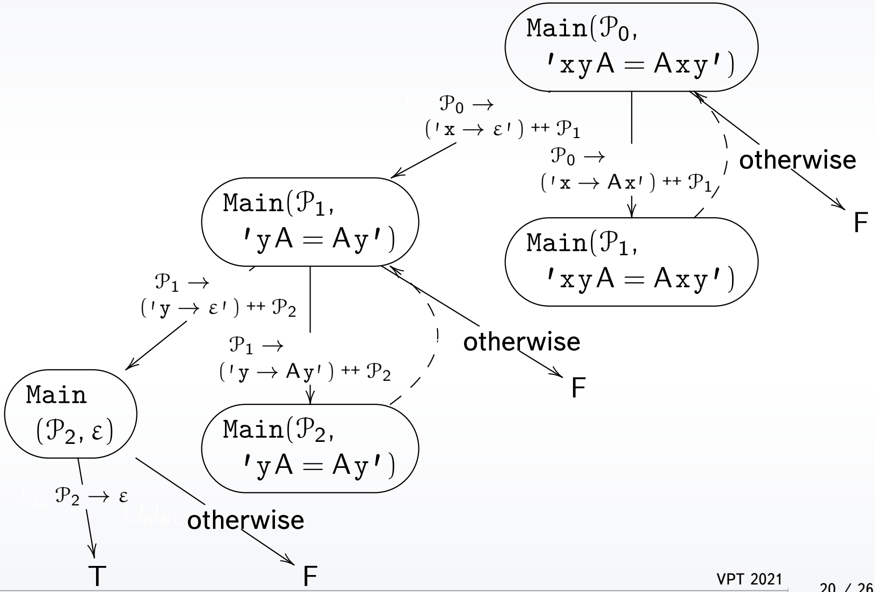
Transient operations



The next unfolding step

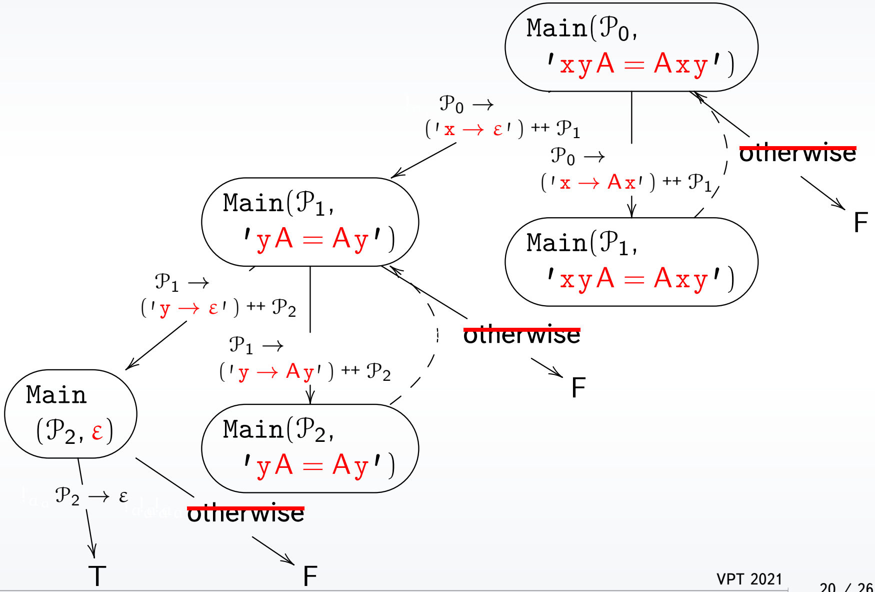


The folding

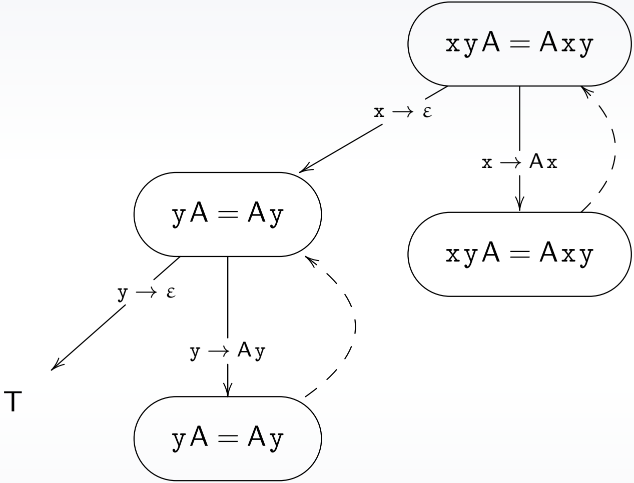


Main($\mathcal{P}_0$, 'xyA = Axy')
$\mathcal{P}_0 \to$ ('x $\to \varepsilon$') ++ $\mathcal{P}_1$
$\mathcal{P}_0 \to$ ('x $\to$ Ax') ++ $\mathcal{P}_1$
otherwise
F
Main($\mathcal{P}_1$, 'yA = Ay')
Main($\mathcal{P}_1$, 'xyA = Axy')
$\mathcal{P}_1 \to$ ('y $\to \varepsilon$') ++ $\mathcal{P}_2$
$\mathcal{P}_1 \to$ ('y $\to$ Ay') ++ $\mathcal{P}_2$
otherwise
F
Main ($\mathcal{P}_2$, ε)
Main($\mathcal{P}_2$, 'yA = Ay')
$\mathcal{P}_2 \to \varepsilon$
otherwise
T
F

Deleting interpreter data



The solution graph



Summary of the verification results

The classes of equations not solvable by CVC4 and Z3Str3 in general but solvable by our verification scheme:

- the quadratic equations with no solution (*e.g.* $x_1 x_2 x_3 ABABAB = AAABBB x_2 x_3 x_1$);
- the regular-ordered equations with var-repetitions and no solution (*e.g.* $ABxxyy = xxyyBA$).

The one-variable word equations not solvable by CVC4 and Z3Str3 also belong to the regular-ordered with repetitions and no solution.

Benchmark results

Benchmark	Tests	Not terminating		
		`CVC4`	`Z3str3`	$\mathrm{WICount}_{\mathscr{L}}$
Track 1 (Woorpje)	200	8	13	21
Track 5 (Woorpje)	200	4	14	19
Our benchmark	50	21	28	10

Average time for $\mathrm{WICount}_{\mathscr{L}}$: 3,5 min for **one equation**.

Time for `CVC4` and `Z3str3` is less than 2 min for all the solved equations.

Challenges

- The function composition depths of all the interpreters are implicitly bounded, i.e. they comprise formal rather than semantic loops.
- The lengths of the programs being interpreted are not explicitly bounded, because the lengths are finite while unknown.

Our solutions to the challenges

- The renaming relation is used on the set of interpreters configurations along a given unfolding path.
- The word equations solved by the proposed method can be encoded with lisp lists.
- The interpreters are also can be written in the functional languages based on the list data rather than the string data.
- The suggested approach can be used by means of various specializers manipulating the lisp lists.

Open challenges

- Refine the method in order to solve wider (or other) classes of the word equations.
- Solving the word equations over the nested words.
- Solving equations over the regular expressions.
- Development of an online SMT string-solver based on the suggested method.

Conclusion

- Supercompilation can be used to solve word equations, based on the specialization of various interpreters of a simple logic language.
- The method has shown itself to be useful to prove unsatisfiability of the word equations sharing variables in left- and right-hand sides.

Thank you for your attention!