Verifying String Replacing Procedures by Supercompilation

Antonina Nepeivoda [*]

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

Due to the complexity of find-replace algorithms for strings [2, 7], the works on verification of string replacing programs mostly aim at verifying properties of strings with uniformly bounded lengths or they restrict the verification domain by programs implementing finite automata [2, 5, 16].

Turchin's supercompilation is one of program specialization methods based on unfold/fold operations [14, 12, 4]. A supercompiler is a specializer based on this method. Turchin's original works use the string operating language Refal [15] as the input language of a supercompiler. This language is based on the Markov normal algorithms computation model, which is known to be Turing complete [7].

A Markov normal algorithm is a list of string rewriting rules of the two following forms.

$$\mathtt{x}{+\!+}\Phi{+\!+}\mathtt{y} \to \mathtt{x}{+\!+}\Psi{+\!+}\mathtt{y} \qquad | \qquad \mathtt{x}{+\!+}\Phi{+\!+}\mathtt{y} \to_{.} \mathtt{x}{+\!+}\Psi{+\!+}\mathtt{y}$$

There Φ, Ψ are constant strings; $\mathtt{x}$ and $\mathtt{y}$ are distinct string variables; $\to$ stands for a normal transition, $\to_{.}$ stands for a transition stopping computations; $+\!+$ is the associative concatenation. The left-hand sides of the rules should be seen as patterns. If an input string Δ matches against the pattern $\mathtt{x}{+\!+}\Phi{+\!+}\mathtt{y}$ then we say the corresponding rule is applicable to Δ. Given an input string Δ the Markov model uses the following two assertions in order to make the rewriting process deterministic.

- *Screening.* Δ is matched against the left-hand sides of the rules from the beginning to the end of the rule list. Thus, if the n-th rule from the list is applied to Δ, then all the rules preceding it are not applicable to Δ.
- *Markov's rule.* If Δ has several occurrences of Φ, we choose the first one to apply the rewriting rule. In particular, the variable $\mathtt{x}$ never takes a value containing the substring Φ.

Markov's rule is a natural choice which is assumed in almost all deterministic find-replace algorithms over strings. In programming languages with the syntax not supporting the associative concatenation (*e.g.*, Lisp, Haskell), the constraints hidden in Markov's rule are to be expressed with the use of recursion. The things change if the program rules can use patterns like $\mathtt{x}{+\!+}\Phi{+\!+}\mathtt{y}$ in the left-hand sides.

Let Σ be an alphabet, ε be the empty string, $\mathscr{V}$ be a set of all variables, $\mathscr{V}_c$ be a set of the variables of the character type. A passive expression $[\mathtt{Passive}]$ over the set of variables $\mathscr{V}$ is defined as follows.

$$[\mathtt{Passive}] \quad ::= \quad \varepsilon \mid c \in \Sigma \mid v \in \mathscr{V} \mid [\mathtt{Passive}]{+\!+}[\mathtt{Passive}]$$

We use the presentation language based on the following version of the Markov normal algorithms. Replacing functions are defined in the following two ways.

$$\mathtt{F}(\mathtt{x}_1,\ldots,\mathtt{x}_{i_1}{+\!+}\Phi{+\!+}\mathtt{x}_{i_2},\ldots,\mathtt{x}_n) = \mathtt{F}(\xi_1,\ldots,\xi_n) \quad | \quad \mathtt{F}(\mathtt{x}_1,\ldots,\mathtt{x}_{i_1}{+\!+}\Phi{+\!+}\mathtt{x}_{i_2},\ldots,\mathtt{x}_n) = \xi$$

[*]The reported study was partially supported by RFBR, research project No. 17-07-00285_a, and by the Ministry of Science and Higher Education of the Russian Federation, research project No. AAAA-A19-119020690043-9.

There $\Phi \in \{\Sigma \cup \mathscr{V}_c\}^*$, ξ_i, ξ are passive expressions over the set of variables occurring in the left-hand side, and all the string variables $\mathtt{x}_j$ are distinct. The rules returning ξ correspond to the final transitions; the others perform the normal transitions. In order to define the verification task, we also introduce a special function `Test` computing a predicate, with the following syntax.

$$\mathtt{Test}(\mathtt{x}_1,\ldots,\mathtt{x}_{i_1}{+}{+}\Phi{+}{+}\mathtt{x}_{i_2},\ldots,\mathtt{x}_n) = \mathbf{T} \mid \mathbf{F} \mid \mathtt{Test}(\xi_1,\ldots,\xi_n)$$

Every program in the presentation language computes a predicate. Thus, the input point of any program is of the form $\mathtt{Test}(\gamma)$, where γ is a composition of the replacing functions.

Given such kind of a language we use the rules of the Markov normal algorithms above to generate constraints, which allow us to prove some safety properties of string manipulating programs. Here the verification is understood in the following sense [6]. Given an initial program P_1 which returns a boolean value, we say P_1 is successfully verified if the residual program P_2 generated by the supercompilation of P_1 does not contain rules returning $\mathbf{F}$. Thus, the implicit semantics feature of P_1 becomes the explicit *syntactic* property of P_2. We do not aim to prove termination of the analysed program, we are only interested in the unreachability of the output value $\mathbf{F}$.

We use the notion of a parameter for an object which already has a value but it is unknown to us; while a variable value is undefined and is to be assigned. Henceforth the string parameters are denoted with the letters u, w, v. We consider the following verification task.

Problem 1 *Given an input string Δ, let $\mathtt{Test}(\Delta)$ return $\mathbf{F}$ if Δ contains a forbidden substring Φ, and $\mathbf{T}$ otherwise. Given a program with the parameterized input point $\mathtt{Test}(\mathtt{F}(u))$, where $\mathtt{F} = \mathtt{F}_1 \circ \mathtt{F}_2 \circ \ldots \mathtt{F}_n$, can it return $\mathbf{F}$?*

Let u be a string presenting some html-code, and Φ be a part of a malicious script possibly included in this code. Then the composition of the functions $\mathtt{F}_i$ can be understood as a sanitization function [3, 13], which tries to prevent a possible attack encoded by an intruder in the string u. The issue is: how to verify that the sanitization procedure is correct in the sense described above or to construct a counterexample refuting its correctness? Using negative constraints as a part of parameterized program states, the supercompilation is sometimes able to solve the problem considered.

We say that γ is *a parameterized expression*, if γ is a constant string, a parameter, a concatenation of parameterized expressions, or a call of a replacing function over parameterized expressions. To reason about the functions defined above, it is enough to use the negative constraints of the form $\wedge_{i=1}^{n} \vee_{j=1}^{k_i} I_j^i$, where I_j^i are linear word inequalities defined as follows.

Definition 1 *Let Σ be an alphabet, $\mathscr{V}_c$ be a set of the character type variables*[1]*. A linear pattern P is an expression $\Phi_0{+}{+}\mathtt{z}_1{+}{+}\Phi_1{+}{+}\ldots{+}{+}\mathtt{z}_n{+}{+}\Phi_n$, where $\Phi_i \in (\Sigma \cup \mathscr{V}_c)^+$, and all the string type variables $\mathtt{z}_i$ are distinct. Let the predicate $\mathtt{Match}(\Delta, P)$ return $\mathbf{T}$ iff Δ matches against P and $\mathbf{F}$ otherwise.*

A linear word inequality *is an inequality I of the form $\neg\mathtt{Match}(u,P)$, where u is a string parameter. For the sake of brevity, we also write I as $u \neq P$.* A parameterized program state *is a pair $\{\gamma, \mathscr{P}red\}$, where $\mathscr{P}red$ is $\wedge_{i=1}^{n} \vee_{j=1}^{k_i} I_j^i$, and γ is a parameterized expression.*

[1] Actually, this extended abstract does not contain any examples with the use of the character type objects. However, they are used in the input language of the supercompiler `MSCP-A` [9] and allow us to operate with alphabets of the unbounded size. Note that the multiple occurrences of the same *character type* variable are not forbidden both in the input language and in the constraint language, thus, the following rules can be used, where $\mathtt{c} \in \mathscr{V}_c$.

$$\mathtt{Pal}(\mathtt{c}{+}{+}\mathtt{x}{+}{+}\mathtt{c}) = \mathtt{Pal}(\mathtt{x})$$

We denote the literal coincidence with $\equiv$, the logical implication with $\Rightarrow$. Given an inequality $I = \neg\texttt{Match}(u,P)$ and a substitution σ from the set of parameters to the set of parameterized expressions, we define $\sigma(I)$ as a predicate which is true iff $\neg\texttt{Match}(\sigma(u),P)$ is true. Given two parameterized program states $C_1 = \{\gamma_1(w_1,\ldots,w_{k_1}),\mathscr{P}red_1\}$ and $C_2 = \{\gamma_2(w_1,\ldots,w_{k_2}),\mathscr{P}red_2\}$, we say C_2 *is folded* with C_1 if there is a substitution σ s.t. $\mathscr{P}red_2 \Rightarrow \sigma(\mathscr{P}red_1)$ and $\gamma_1(\sigma(w_1),\ldots,\sigma(w_{k_1})) \equiv \gamma_2(w_1,\ldots,w_{k_2})$. *A generalization* of parameterized states $\{\gamma_1,\mathscr{P}red_1\}$, $\{\gamma_2,\mathscr{P}red_2\}$ is a state $\{\gamma',\mathscr{P}red'\}$ and substitutions σ_1, σ_2 s.t.

- $\gamma'(\sigma_1(u_1),\ldots,\sigma_1(u_n)) \equiv \gamma_1(w_1,\ldots,w_{k_1})$, $\gamma'(\sigma_2(u_1),\ldots,\sigma_2(u_n)) \equiv \gamma_2(w_1,\ldots,w_{k_2})$;
- $\mathscr{P}red_1 \Rightarrow \sigma_1(\mathscr{P}red')$, $\mathscr{P}red_2 \Rightarrow \sigma_2(\mathscr{P}red')$.

We emphasize that we do not require the parameter values lengths to be uniformly bounded. Constraints are generated by the supercompiler as follows. First, they are taken from the screening rule and Markov's rule. Second, they may be generated by the generalization algorithm based on the program behaviour, without a reference to the syntax of the input program (*e.g.*, see Example #15 in the list of `MSCP-A` demo examples [9]). This approach is the main novelity of the presented work. In both cases, the constraints are considered as hypotheses that are to be proved or refuted along the computation paths starting from the state where they are generated. Our contributions are the following:

1. We have developed the algorithms to generate, simplify, and check the inference of the negative constraints.
2. We have developed the algorithms to unfold, fold, and generalize the program states of the form given above.
3. We proved that the version of supercompilation using these algorithms terminates.

The algorithms we have developed continue the work on unfolding string programs [8], providing the entire scheme of supercompilation for the string programs of the restricted type. In addition to the algorithm given in [8], we have constructed an algorithm extracting and simplifying the negative constraints introduced in the paper [8] in an informal way. As far as we know, our unfolding algorithm is the first algorithm for the perfect driving of the string manipulating programs given in a language whose syntax supports the associative concatenation. Unlike the algorithm for the perfect driving of programs over lists given in [11], our algorithm for the string manipulating programs uses negative constraints which cannot be checked in a uniformly bounded number of steps. Moreover, not only the unfolding scheme, but the whole scheme of the supercompilation of the string manipulating programs from the class discussed in [8] has been developed and implemented.

All the mentioned algorithms are implemented in the model supercompiler `MSCP-A` transforming programs written in the language Refal (the web-page of the supercompiler is [9]). The web-page also contains a number of examples showing how the supercompiler works, including the examples verifying models of the programs given in the papers by the other authors [1, 3, 13].

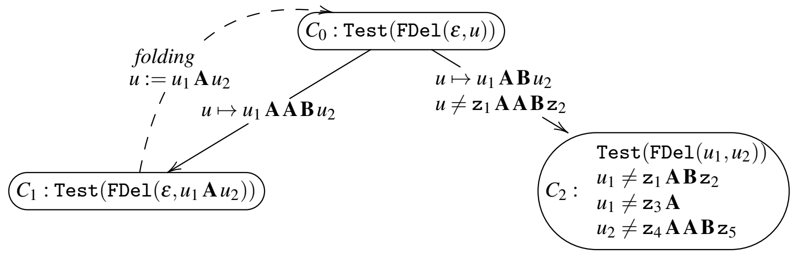
Example 1 *Below* $\texttt{x}_i$, $\texttt{y}_i$, *and* $\texttt{z}_i$ *are the string type variables;* u_i, w_i *are the string type parameters;* **A**, **B** *belong to* Σ*;* ε *is the empty string; the concatenation symbol* $++$ *is omitted.*

Let us show how our version of the supercompilation does work on the input point $\texttt{Test}(\texttt{FDel}(\varepsilon,u))$ *and the following definition.*

$$\begin{array}{lcllcl}
\texttt{FDel}(\texttt{y},\texttt{x}_1\,\mathbf{A}\,\mathbf{A}\,\mathbf{B}\,\texttt{x}_2) & = & \texttt{FDel}(\texttt{y},\texttt{x}_1\,\mathbf{A}\,\texttt{x}_2); & \texttt{Test}(\texttt{x}_1\,\mathbf{A}\,\mathbf{B}\,\texttt{x}_2) & = & \mathbf{F};\\
\texttt{FDel}(\texttt{y},\texttt{x}_1\,\mathbf{A}\,\mathbf{B}\,\texttt{x}_2) & = & \texttt{FDel}(\texttt{y}\,\texttt{x}_1,\texttt{x}_2); & \texttt{Test}(\texttt{x}) & = & \mathbf{T};\\
\texttt{FDel}(\texttt{y},\texttt{x}) & = & \texttt{y}\,\texttt{x}; & & &
\end{array}$$

The screening rule and Markov's rule are both used in the program semantics. Thus, given a string matched against the pattern $\mathtt{x}_1\,\mathbf{A\,A\,B}\,\mathtt{x}_2$*, its prefix string matched against* $\mathtt{x}_1\,\mathbf{A\,A}$ *cannot contain the substring* $\mathbf{A\,A\,B}$ *(due to Markov's rule); given a string matched against the second pattern* $\mathtt{x}_1\,\mathbf{A\,B}\,\mathtt{x}_2$*, it cannot contain the substring* $\mathbf{A\,A\,B}$ *(due to the screening rule), etc.*

In the diagrams below, the branches outgoing from the root node are ordered from left to right. If the set of the negative constraints is empty, it is omitted for the sake of brevity. The arrow $\mapsto$ *means a narrowing relation imposed on a parameter. The dashed arc stands for the folding.*

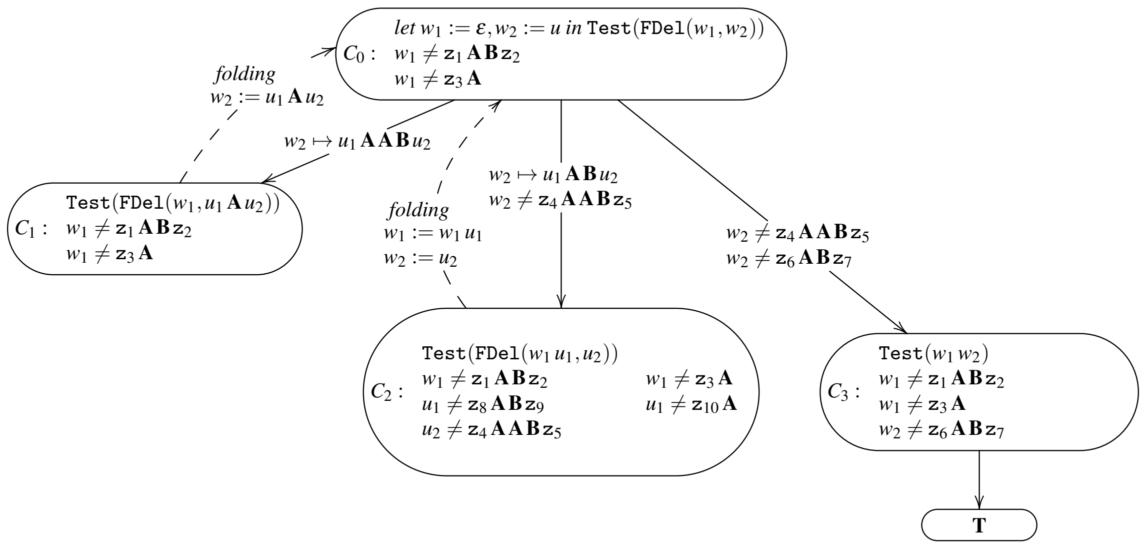


$C_0 : \mathtt{Test}(\mathtt{FDel}(\varepsilon, u))$
folding
$u := u_1\,\mathbf{A}\,u_2$
$u \mapsto u_1\,\mathbf{A\,A\,B}\,u_2$
$u \mapsto u_1\,\mathbf{A\,B}\,u_2$
$u \neq \mathtt{z}_1\,\mathbf{A\,A\,B}\,\mathtt{z}_2$
$C_1 : \mathtt{Test}(\mathtt{FDel}(\varepsilon, u_1\,\mathbf{A}\,u_2))$
$C_2 :$ $\mathtt{Test}(\mathtt{FDel}(u_1, u_2))$
$u_1 \neq \mathtt{z}_1\,\mathbf{A\,B}\,\mathtt{z}_2$
$u_1 \neq \mathtt{z}_3\,\mathbf{A}$
$u_2 \neq \mathtt{z}_4\,\mathbf{A\,A\,B}\,\mathtt{z}_5$

In the parameterized program state C_2*, the constraint* $u_1 \neq \mathtt{z}_1\,\mathbf{A\,B}\,\mathtt{z}_2$ *is generated by Markov's rule. The other two constraints are corollaries of the screening rule.*

The parameterized program state C_1 *is folded with* C_0*. When the program state* C_2 *has been generated,* C_0 *and* C_2 *are generalized. The generalization algorithm generates expression* $\mathtt{Test}(\mathtt{FDel}(w_1, w_2))$ *and substitutions* σ_1*,* σ_2*:* $\sigma_1(w_1) = \varepsilon$*,* $\sigma_1(w_2) = u_1$*,* $\sigma_2(w_1) = u_1$*,* $\sigma_2(w_2) = u_2$*. Both of the constraints imposed on* u_1 *are preserved in the generalized state, while the constraint on* u_2 *disappears.*

The entire residual graph generated by our version of the supercompilation is as follows.



$C_0 :$ $let\ w_1 := \varepsilon, w_2 := u\ in\ \mathtt{Test}(\mathtt{FDel}(w_1, w_2))$
$w_1 \neq \mathtt{z}_1\,\mathbf{A\,B}\,\mathtt{z}_2$
$w_1 \neq \mathtt{z}_3\,\mathbf{A}$
folding
$w_2 := u_1\,\mathbf{A}\,u_2$
$w_2 \mapsto u_1\,\mathbf{A\,A\,B}\,u_2$
$w_2 \mapsto u_1\,\mathbf{A\,B}\,u_2$
$w_2 \neq \mathtt{z}_4\,\mathbf{A\,A\,B}\,\mathtt{z}_5$
folding
$w_1 := w_1\,u_1$
$w_2 := u_2$
$w_2 \neq \mathtt{z}_4\,\mathbf{A\,A\,B}\,\mathtt{z}_5$
$w_2 \neq \mathtt{z}_6\,\mathbf{A\,B}\,\mathtt{z}_7$
$C_1 :$ $\mathtt{Test}(\mathtt{FDel}(w_1, u_1\,\mathbf{A}\,u_2))$
$w_1 \neq \mathtt{z}_1\,\mathbf{A\,B}\,\mathtt{z}_2$
$w_1 \neq \mathtt{z}_3\,\mathbf{A}$
$C_2 :$ $\mathtt{Test}(\mathtt{FDel}(w_1\,u_1, u_2))$
$w_1 \neq \mathtt{z}_1\,\mathbf{A\,B}\,\mathtt{z}_2$
$w_1 \neq \mathtt{z}_3\,\mathbf{A}$
$u_1 \neq \mathtt{z}_8\,\mathbf{A\,B}\,\mathtt{z}_9$
$u_1 \neq \mathtt{z}_{10}\,\mathbf{A}$
$u_2 \neq \mathtt{z}_4\,\mathbf{A\,A\,B}\,\mathtt{z}_5$
$C_3 :$ $\mathtt{Test}(w_1\,w_2)$
$w_1 \neq \mathtt{z}_1\,\mathbf{A\,B}\,\mathtt{z}_2$
$w_1 \neq \mathtt{z}_3\,\mathbf{A}$
$w_2 \neq \mathtt{z}_6\,\mathbf{A\,B}\,\mathtt{z}_7$
$\mathbf{T}$

The inequality set in the program state C_3 *hinders the successful matching of* $w_1\,w_2$ *against the pattern* $\mathtt{z}_1\,\mathbf{A\,B}\,\mathtt{z}_2$*. Thus the program cannot return* $\mathbf{F}$ *for any input string, and* `FDel` *does a correct sanitization of the substring* $\mathbf{A\,B}$ *in the sense described above. Note that without the constraint* $w_1 \neq \mathtt{z}_3\,\mathbf{A}$ *generated by the supercompiler, we cannot prove the correctness of the sanitization. Hence, if the first rule of the function* `FDel` *is removed, the exhaustive deletion algorithm becomes incorrect. That can be shown by the input point* $\mathtt{Test}(\mathtt{FDel}(\varepsilon, \mathbf{A\,A\,B\,B}))$*.*

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. Available at `http://doi.acm.org/10.1145/3140587.3062384`.

[2] 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.

[3] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet & Ryan Berg (2011): *Saving the world wide web from vulnerable JavaScript*. In: *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, pp. 177–187, doi:10.1145/2001420.2001442. Available at `https://doi.org/10.1145/2001420.2001442`.

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

[5] 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. Available at `http://dx.doi.org/10.1007/s10703-016-0247-6`.

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

[7] Andrei Andreevich Markov (1960): *The Theory of Algorithms*. *American Mathematical Society Translations* 15, pp. 1–14.

[8] 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, The workshop is an event of the Vienna Summer of Logic 2014 and it is co-located with the 26th International Conference on Computer Aided Verification CAV 2014*, pp. 66–83. Available at `http://www.easychair.org/publications/paper/184415`.

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

[10] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant & D. Song (2010): *A symbolic execution framework for javascript*. In: *SP*, pp. 513–528.

[11] 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. Available at `https://doi.org/10.1007/3-540-46562-6_10`.

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

[13] 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.

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

[15] 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/`.

[16] 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.