Verifying String Replacing Procedures by Supercompilation

Antonina Nepeivoda

Program Systems Institute of RAS

Verification and Program Transformation, April 2, 2019

Verification of String-Replace Algorithms

- Strings are the main data type for the majority of scripting languages (Python, JavaScript, PHP). String-replace algorithms are used either in sanitization and implicit browser transduction ([Bjorner 2009], [Yu 2014], [Trinh 2016], [Abdulla 2017]).
- Replace-all algorithms for strings are Turing complete [Markov 1960].

```
var b = new B();
var doc = document;
var w = "wr"+ "ite";
doc[w](b.f);
```

```
function json_decode(str) {
  str = str.replace(/ip/g, "ip address");
  str = str.replace(/dom/g, "domain");
  return str;}
function json_show(str) {
  var arr = JSON.parse(str);
  var c = arr[0].content.split("&");
  var s = c[0]++c[1];
  document.getElementById("info").innerHTML = s;}

res = json_decode(input);
json_show(res);
```

Main Contribution

The existing methods of supercompilation are developed and implemented to verify safety properties of programs over strings of an arbitrary length.

- The program state is a pair

$$\{[\texttt{Stack}], [\texttt{Constraints}]\},$$

  where the set of constraints contains word equations and inequalities of a restricted form.
- The methods to unfold, fold, and generalize the program states with constraints are developed.
- The constraints are generated either from the program syntax or from the program behaviour on computation paths.

Verification of Safety Properties by Transformation

Consider a program $\mathcal{P}(u)$, where $u = \langle u_1, \ldots, u_n \rangle$, $u \in \mathfrak{D}$, and a parameterized input point $\mathcal{P}(u')$, $u' \in \mathfrak{D}'$, $\mathfrak{D}' \subset \mathfrak{D}$. Given a safety property q,

- specialize $\mathcal{P}(u')$ to residual program $\mathcal{P}'(u')$;
- check a simple explicit syntactic property of $\mathcal{P}'(u')$ showing that q holds.

Then the safety property of $\mathcal{P}$ is verified on the data set $\mathfrak{D}'$.

Example

The input points are $q(\mathtt{G}(u))$ and $\mathtt{H}(u)$ respectively.

Input Program			Residual Program		
$q(\alpha_1)$	$=$	$\mathbf{T}$;	$\mathtt{H}(\delta_0)$	$=$	$\mathbf{T}$;
$q(\alpha_2)$	$=$	$\mathbf{F}$;	$\mathtt{H}(\delta_1)$	$=$	$\mathtt{H}(\xi_1)$;
$\mathtt{G}(\beta_1)$	$=$	$\mathtt{G}(\gamma_1)$;	$\mathtt{H}(\delta_2)$	$=$	$\mathtt{H}(\xi_2)$;
	$\ldots$			$\ldots$	
$\mathtt{G}(\beta_k)$	$=$	$\mathtt{G}(\gamma_k)$;	$\mathtt{H}(\delta_m)$	$=$	$\mathtt{H}(\xi_m)$;

The residual program does not contain rules returning $\mathbf{F}$. Hence, the input program is safe.

Markov Normal Algorithms

Let Σ be an alphabet, $\Phi, \Psi \in \Sigma^+$, $\mathtt{x}$ and $\mathtt{y}$ be variables taking values from Σ^*.
A Markov normal algorithm is a list of rewriting rules [Markov 1960].

$\mathtt{x}{+}{+}\Phi{+}{+}\mathtt{y} \to \mathtt{x}{+}{+}\Psi{+}{+}\mathtt{y}$	$\mathtt{x}{+}{+}\Phi{+}{+}\mathtt{y} \to_{\cdot} \mathtt{x}{+}{+}\Psi{+}{+}\mathtt{y}$
normal rule	*terminating rule*

The order of the rewriting process is controlled by:

- *Screening.* A string is matched against the left-hand sides of the rules from the beginning to the end of the rule list.
- *Markov's rule.* If a string has several occurrences of Φ, we choose the first one to apply the rule.

Example

$$\mathtt{x}{+}{+}\mathbf{A}{+}{+}\mathbf{A}{+}{+}\mathtt{y} \to \mathtt{x}{+}{+}\mathbf{A}{+}{+}\mathbf{B}{+}{+}\mathtt{y}$$
$$\mathtt{x}{+}{+}\mathbf{B}{+}{+}\mathbf{A}{+}{+}\mathtt{y} \to \mathtt{x}{+}{+}\mathbf{B}{+}{+}\mathbf{B}{+}{+}\mathtt{y}$$
$$\mathtt{x}{+}{+}\mathbf{B}{+}{+}\mathtt{y} \to_{\cdot} \mathtt{x}{+}{+}\mathbf{B}{+}{+}\mathtt{y}$$

A run (the associative concatenation $+{+}$ is omitted):

$$\mathbf{AAAAB} \to \mathbf{ABAAB} \to \mathbf{ABABB} \to \mathbf{ABBBB} \to_{\cdot} \mathbf{ABBBB}$$

Presentation Language: Sample Example

$$
\begin{array}{lcl}
\texttt{Go}(\texttt{x}) & = & \texttt{Test}(\texttt{F}(\varepsilon, \texttt{x}));\\
\texttt{F}(\texttt{x}, \mathbf{AB}\,\texttt{y}) & = & \texttt{F}(\texttt{x}, \texttt{y});\\
\texttt{F}(\texttt{x}, \texttt{y}\,\texttt{c}\,\mathbf{AB}\,\texttt{z}) & = & \texttt{F}(\texttt{x}\,\texttt{y}, \texttt{c}\,\texttt{z});\\
\texttt{F}(\texttt{x}, \texttt{y}) & = & \texttt{x}\,\texttt{y};
\end{array}
\qquad
\begin{array}{lcl}
\texttt{Test}(\texttt{x}\,\mathbf{AB}\,\texttt{y}) & = & \mathbf{F};\\
\texttt{Test}(\texttt{x}) & = & \mathbf{T};
\end{array}
$$

ε — the empty string, $\texttt{x}$, $\texttt{y}$, $\texttt{z}$ — string type variables, $\texttt{c}$ — character type variable.

The entry point: $\texttt{Go}(v)$.

A run example:

$$\texttt{Go}(\mathbf{BAABB}) \to \texttt{Test}(\texttt{F}(\varepsilon, \mathbf{BAABB})) \to \texttt{Test}(\texttt{F}(\mathbf{B}, \mathbf{AB})) \to \texttt{Test}(\texttt{F}(\mathbf{B}, \varepsilon)) \to \texttt{Test}(\mathbf{B}) \to \mathbf{T}$$

The rule application order is the same as for Markov normal algorithms.

Presentation Language: Sample Example

$$
\begin{array}{lcl}
\texttt{Go}(\texttt{x}) & = & \texttt{Test}(\texttt{F}(\varepsilon, \texttt{x}));\\
\texttt{F}(\texttt{x}, \mathbf{AB}\,\texttt{y}) & = & \texttt{F}(\texttt{x}, \texttt{y});\\
\texttt{F}(\texttt{x}, \texttt{y}\,\texttt{c}\,\mathbf{AB}\,\texttt{z}) & = & \texttt{F}(\texttt{x}\,\texttt{y}, \texttt{c}\,\texttt{z});\\
\texttt{F}(\texttt{x}, \texttt{y}) & = & \texttt{x}\,\texttt{y};
\end{array}
\qquad
\begin{array}{lcl}
\texttt{Test}(\texttt{x}\,\mathbf{AB}\,\texttt{y}) & = & \mathbf{F};\\
\texttt{Test}(\texttt{x}) & = & \mathbf{T};
\end{array}
$$

Does F a correct exhaustive deletion? Does not!
The program is not safe.

$$\texttt{Go}(\mathbf{AAABBB}) \to \texttt{Test}(\texttt{F}(\varepsilon, \mathbf{AAABBB})) \to \texttt{Test}(\texttt{F}(\mathbf{A}, \mathbf{ABB}))$$
$$\to \texttt{Test}(\texttt{F}(\mathbf{A}, \mathbf{B})) \to \texttt{Test}(\mathbf{AB}) \to \mathbf{F}$$

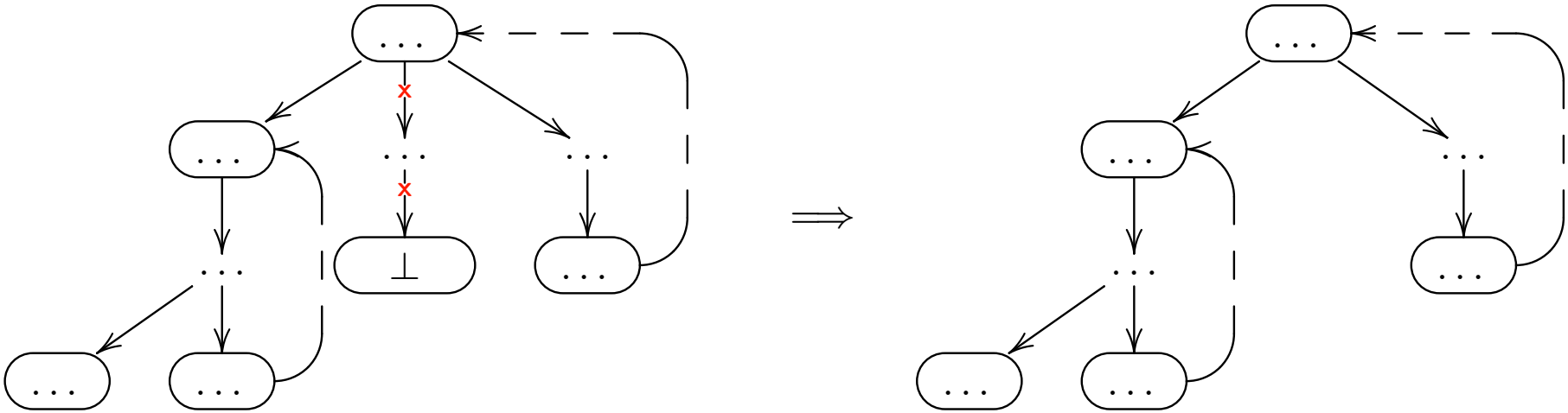
More Examples of String-Replace Algorithms

Which of the following programs does a correct exhaustive deletion?

$$
\begin{array}{lcl@{\qquad}lcl}
\mathtt{Go(x)} & = & \mathtt{Test(G(\varepsilon, x))}; & & & \\
\mathtt{G(\varepsilon, AB\,x)} & = & \mathtt{G(\varepsilon, x)}; & \mathtt{Test(x\,AB\,y)} & = & \mathbf{F}; \\
\mathtt{G(x\,c, AB\,y)} & = & \mathtt{G(x, c\,y)}; & \mathtt{Test(x)} & = & \mathbf{T}; \\
\mathtt{G(x, y\,c\,AB\,z)} & = & \mathtt{G(x\,y, c\,z)}; & & & \\
\mathtt{G(x, y)} & = & \mathtt{x\,y}; & & &
\end{array}
$$

$$
\begin{array}{lcl@{\qquad}lcl}
\mathtt{Go(x)} & = & \mathtt{Test(H(\varepsilon, x))}; & & & \\
\mathtt{H(x, y\,AAB\,z)} & = & \mathtt{H(x, y\,A\,z)}; & \mathtt{Test(x\,AB\,y)} & = & \mathbf{F}; \\
\mathtt{H(x, y\,AB\,z)} & = & \mathtt{H(x\,y, z)}; & \mathtt{Test(x)} & = & \mathbf{T}; \\
\mathtt{H(x, y)} & = & \mathtt{x\,y}; & & &
\end{array}
$$

Brief Introduction to Supercompilation

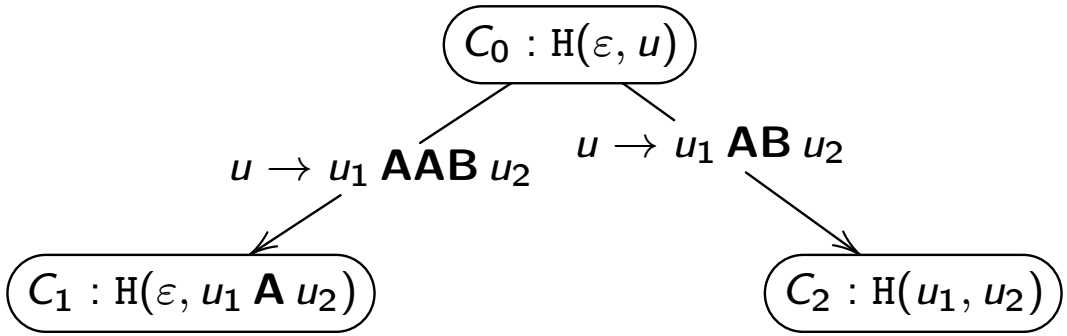


Supercompilation is a program specialization method based on unfold/fold operations. Introduced by V. F. Turchin using the string operating language Refal as the input language [Turchin 1986].

One of the ways of verification by supercompilation is to prune unreachable branches in the unfolding tree.

Unfolding for String Data Type

The arrow $\rightarrow$ stands for the narrowing relation.



Problem: The narrowing for C_1 is an instance of the narrowing for C_2.

We use the negative constraints to preserve information in the program states.

Configurations

Σ — an alphabet, $\mathcal{V}_c$ — a set of the character type variables.

Definition

A linear pattern P is an expression $\Phi_0{+}{+}z_1{+}{+}\Phi_1{+}{+}\ldots{+}{+}z_n{+}{+}\Phi_n$, where $\Phi_i \in (\Sigma \cup \mathcal{V}_c)^*$, and all the string type variables z_i *are distinct*.

Let $\texttt{Match}(\Delta, P)$ return **T** if Δ matches against P and **F** otherwise.
A linear word inequality is a predicate $\mathcal{I} = \neg\,\texttt{Match}(u, P)$, where u is a parameter. Briefly: $u \neq P$.

Definition

A configuration is a pair $\{\gamma, \mathcal{Q}\}$, where $\mathcal{Q}$ is a constraint of the form $\wedge_{i=1}^{n} \vee_{j=1}^{k_i} \mathcal{I}_j^i$, and γ is a parameterized expression.

Example

$c{+}{+}z_0{+}{+}c$, $c \in \mathcal{V}_c$ is a linear pattern; while $z_0{+}{+}\mathbf{A}{+}{+}z_0$ is not.

Simplifying Linear Patterns

Given a pattern P we say that a variable $c \in \mathcal{V}_c$ is *floating* in P, if

- c occurs in P only once;
- c is either preceded or succeeded by a string type variable.

We say that a linear pattern P is *in a normal form*, if

- it does not contain two or more string variables in a row;
- all the floating variables from $\mathcal{V}_c$ are both preceded and succeeded by string type variables in P.

Example

In the pattern $z_0\ c_1\ \mathbf{A}\ c_2\ \mathbf{B}\ c_3\ z_1\ z_2\ c_3$, only c_1 is a floating variable.
The normal form is $z_0\ c_1\ z_1\ \mathbf{A}\ c_2\ \mathbf{B}\ c_3\ z_2\ c_3$.

Simplifying Linear Patterns

The pattern language $\mathcal{L}(P)$ is a set of all strings Δ s.t. $\texttt{Match}(\Delta, P) = \mathbf{T}$. We consider the case of an infinite Σ.

Inclusion of pattern languages for linear patterns with no character type variables occurs iff a substitution of pattern variables exists [Angluin, 1980].

Does not hold for all linear patterns: $\mathcal{L}(\mathtt{c}\,\mathtt{x}) = \mathcal{L}(\mathtt{x}\,\mathtt{c})$.

Theorem (Inclusion for normalized patterns)

Given two linear patterns P_1 and P_2 in the normal form, $\mathcal{L}(P_1) \subseteq \mathcal{L}(P_2)$ iff a substitution σ of variables exists s.t. $\sigma(P_2) = P_1$.

Folding

$L(C_i)$ is a set of computation paths starting from C_i. u, w, v are the string type parameters.

Given two configurations $C_i = \{\gamma_i(w_1, \ldots, w_{k_i}), \mathcal{Q}_i\}$, $i = 1, 2$,
C_2 *is folded* with C_1 if there is a substitution σ of parameters of C_1 s.t.

- $\gamma_1(\sigma(w_1), \ldots, \sigma(w_{k_1})) \equiv \gamma_2(w_1, \ldots, w_{k_2})$;
- $\mathcal{Q}_2$ implies $\sigma(\mathcal{Q}_1)$.

Hence, $L(C_2) \subseteq L(C_1)$.

Example

The configuration $\{\mathtt{F}(\mathbf{B}, v_1\mathbf{A}\mathbf{B}v_2), v_1 \neq \mathtt{z}_1\mathbf{A}\mathtt{z}_2\}$ is folded with
$\{\mathtt{F}(u_0, u_1\mathbf{A}u_2), u_0 \neq \mathtt{z}_0\mathbf{A}\mathtt{z}_1\}$.
The substitution is $\sigma(u_0) = \mathbf{B}$, $\sigma(u_1) = v_1$, $\sigma(u_2) = \mathbf{B}v_2$.

Obviously,

$$v_1 \neq \mathtt{z}_1\mathbf{A}\mathtt{z}_2 \Rightarrow (\mathbf{B} \neq \mathtt{z}_0\mathbf{A}\mathtt{z}_1)$$

Generalization

A generalization of configurations $\{\gamma_1, \mathcal{Q}_1\}$, $\{\gamma_2, \mathcal{Q}_2\}$ is a configuration $\{\gamma', \mathcal{Q}'\}$ and substitutions σ_1, σ_2 s.t.

- $\gamma'(\sigma_1(u_1), \ldots, \sigma_1(u_n)) \equiv \gamma_1(w_1, \ldots, w_{k_1})$,
  $\gamma'(\sigma_2(u_1), \ldots, \sigma_2(u_n)) \equiv \gamma_2(w_1, \ldots, w_{k_2})$;
- $\mathcal{Q}_1 \Rightarrow \sigma_1(\mathcal{Q}')$, $\mathcal{Q}_2 \Rightarrow \sigma_2(\mathcal{Q}')$.

Hence $L(C_1) \subseteq L(C')$ & $L(C_2) \subseteq L(C')$.

Example

$\{\mathrm{F}(\varepsilon, v), \mathbf{T}\}$ and $\{\mathrm{F}(s, u), s \neq \mathbf{A}\}$, where s is a character type parameter, can be generalized to $\{\mathrm{F}(w_1, w_2), w_1 \neq \mathrm{z}_0\mathbf{A}\mathrm{z}_1\}$.

$$\sigma_1(w_1) = \varepsilon; \qquad \sigma_2(w_1) = s;$$
$$\sigma_1(w_2) = v; \qquad \sigma_2(w_2) = u;$$
$$\mathbf{T} \Rightarrow \varepsilon \neq \mathrm{z}_0\mathbf{A}\mathrm{z}_1; \qquad s \neq \mathbf{A} \Rightarrow s \neq \mathrm{z}_0\mathbf{A}\mathrm{z}_1.$$

Generalization Problems for Strings

Many possible generalizations, not comparable with each other.

		$\Phi_1 = \mathbf{B}$	$\Phi_2 = \mathbf{B}\,\mathbf{B}\,\mathbf{B}$
	$\Psi = \mathbf{B}u$	$\sigma_1(u) = \varepsilon$	$\sigma_2(u) = \mathbf{B}\,\mathbf{B}$
OR	$\Psi = u\mathbf{B}$	$\sigma_1(u) = \varepsilon$	$\sigma_2(u) = \mathbf{B}\,\mathbf{B}$
OR	$\Psi = u\mathbf{B}u$	$\sigma_1(u) = \varepsilon$	$\sigma_2(u) = \mathbf{B}$
OR	$\Psi = \mathbf{B}uu$	$\sigma_1(u) = \varepsilon$	$\sigma_2(u) = \mathbf{B}$
OR	$\Psi = uu\mathbf{B}$	$\sigma_1(u) = \varepsilon$	$\sigma_2(u) = \mathbf{B}$

...and an infinite list of the other generalizations.

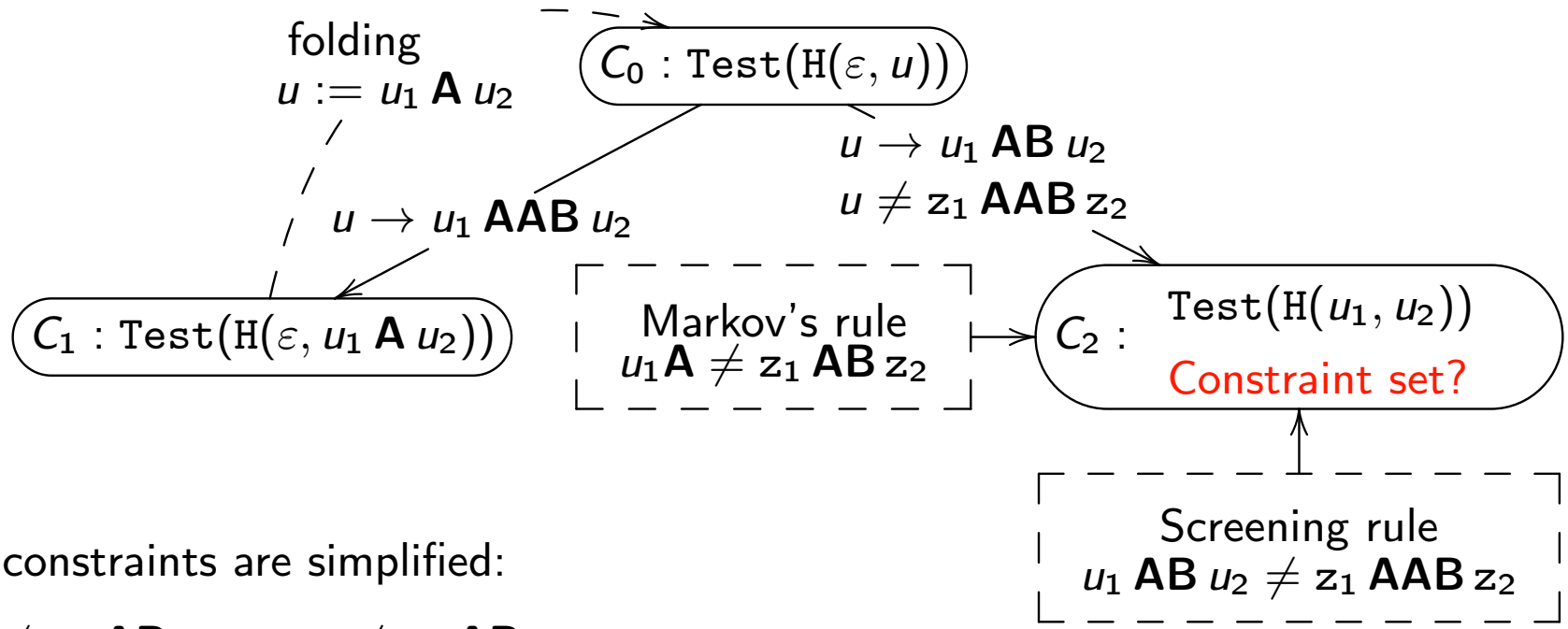
The notion of *a minimal* generalization instead of *the* most specific generalization:

- Two or more string parameters in a row are forbidden.
- For all u, $\sigma_2(u) \trianglelefteq \sigma_1(u)$.[1]
- For no u, $\sigma_2(u) = \varepsilon$.

[1] $\trianglelefteq$ is the homeomorphic embedding relation.

Supercompilation Example

$$
\begin{array}{lcl}
\mathtt{Go(x)} & = & \mathtt{Test(H(\varepsilon, x))};\\
\mathtt{H(x, y\,AAB\,z)} & = & \mathtt{H(x, y\,A\,z)};\\
\mathtt{H(x, y\,AB\,z)} & = & \mathtt{H(x\,y, z)};\\
\mathtt{H(x, y)} & = & \mathtt{x\,y};
\end{array}
\qquad
\begin{array}{lcl}
\mathtt{Test(x\,AB\,y)} & = & \mathtt{F};\\
\mathtt{Test(x)} & = & \mathtt{T};
\end{array}
$$



The constraints are simplified:

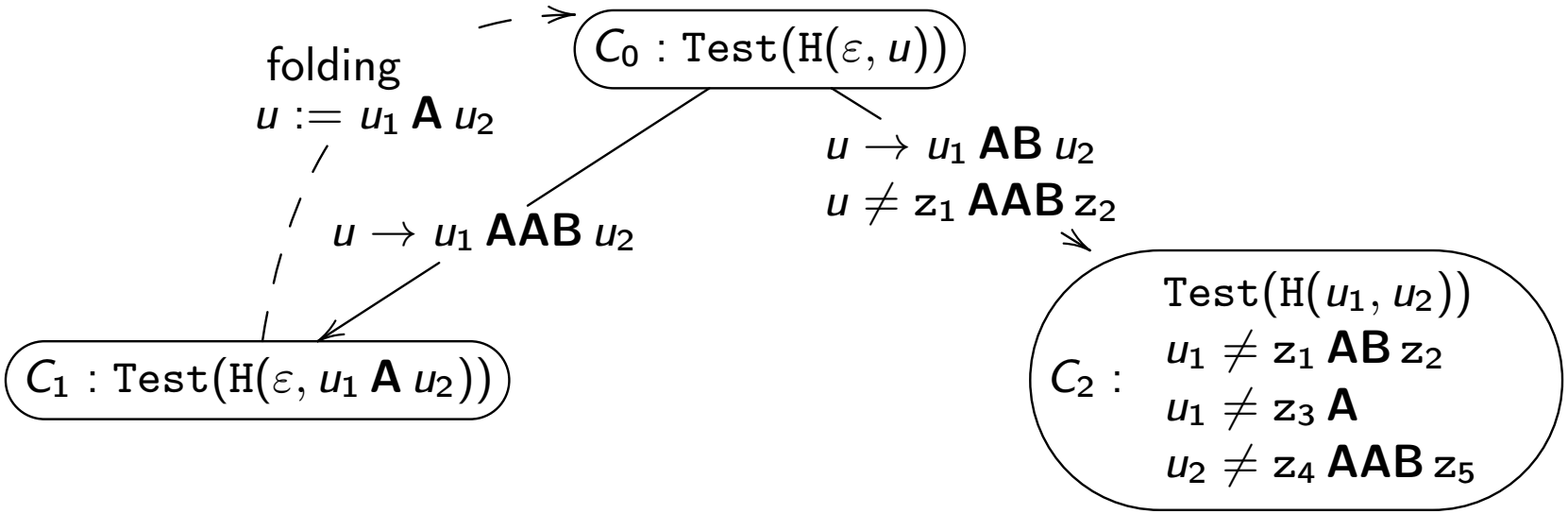
$u_1\mathbf{A} \neq z_1\,\mathbf{AB}\,z_2 \Rightarrow u_1 \neq z_1\,\mathbf{AB}\,z_2$

$u_1\mathbf{AB}u_2 \neq z_1\,\mathbf{AAB}\,z_2 \Rightarrow (u_1 \neq z_3\,\mathbf{A})\ \&\ (u_2 \neq z_4\,\mathbf{AAB}\,z_5)\ \&\ (u_1 \neq z_6\,\mathbf{AAB}\,z_7)$.

Since $u_1 \neq z_1\,\mathbf{AB}\,z_2 \Rightarrow u_1 \neq z_6\,\mathbf{AAB}\,z_7$, the last constraint is removed.

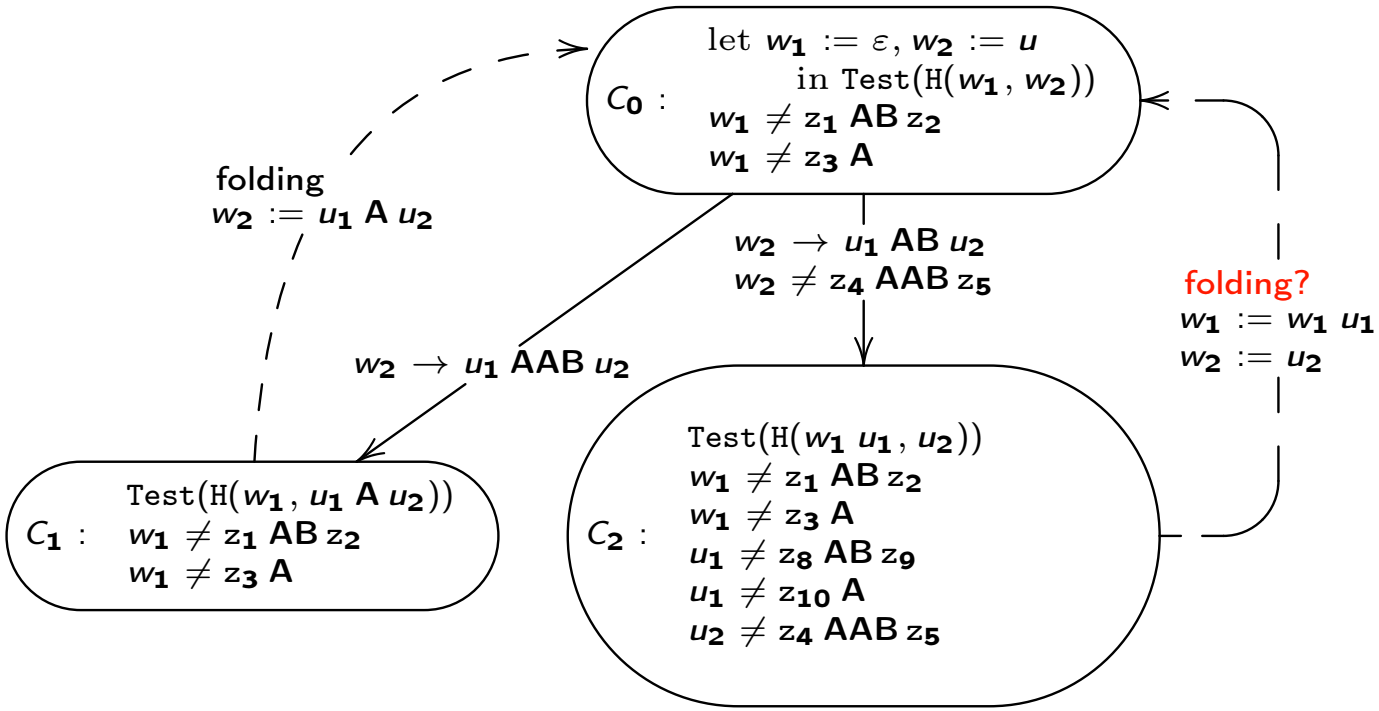
Supercompilation Example

$$
\begin{array}{lcl}
\texttt{Go(x)} & = & \texttt{Test(H}(\varepsilon, \texttt{x}\texttt{)}\texttt{)}; \\
\texttt{H(x, y }\mathbf{AAB}\texttt{ z)} & = & \texttt{H(x, y }\mathbf{A}\texttt{ z)}; \\
\texttt{H(x, y }\mathbf{AB}\texttt{ z)} & = & \texttt{H(x y, z)}; \\
\texttt{H(x, y)} & = & \texttt{x y};
\end{array}
\qquad
\begin{array}{lcl}
\texttt{Test(x }\mathbf{AB}\texttt{ y)} & = & \mathbf{F}; \\
\texttt{Test(x)} & = & \mathbf{T};
\end{array}
$$



Generalization of C_0 and C_2 is constructed: $\{\texttt{Test(H}(w_1, w_2)\texttt{)}, ???\}$.
$\sigma_1(w_1) = \varepsilon$, $\sigma_2(w_1) = u_1$; $\sigma_1(w_2) = u$, $\sigma_2(w_2) = u_2$.
No constraints on u, hence no constraints on w_2 in the generalization.
$\varepsilon \neq z_1\,\mathbf{AB}\,z_2$ and $\varepsilon \neq z_3\,\mathbf{A}$, hence both constraints on u_1 in C_2 hold for w_1.

Supercompilation Example

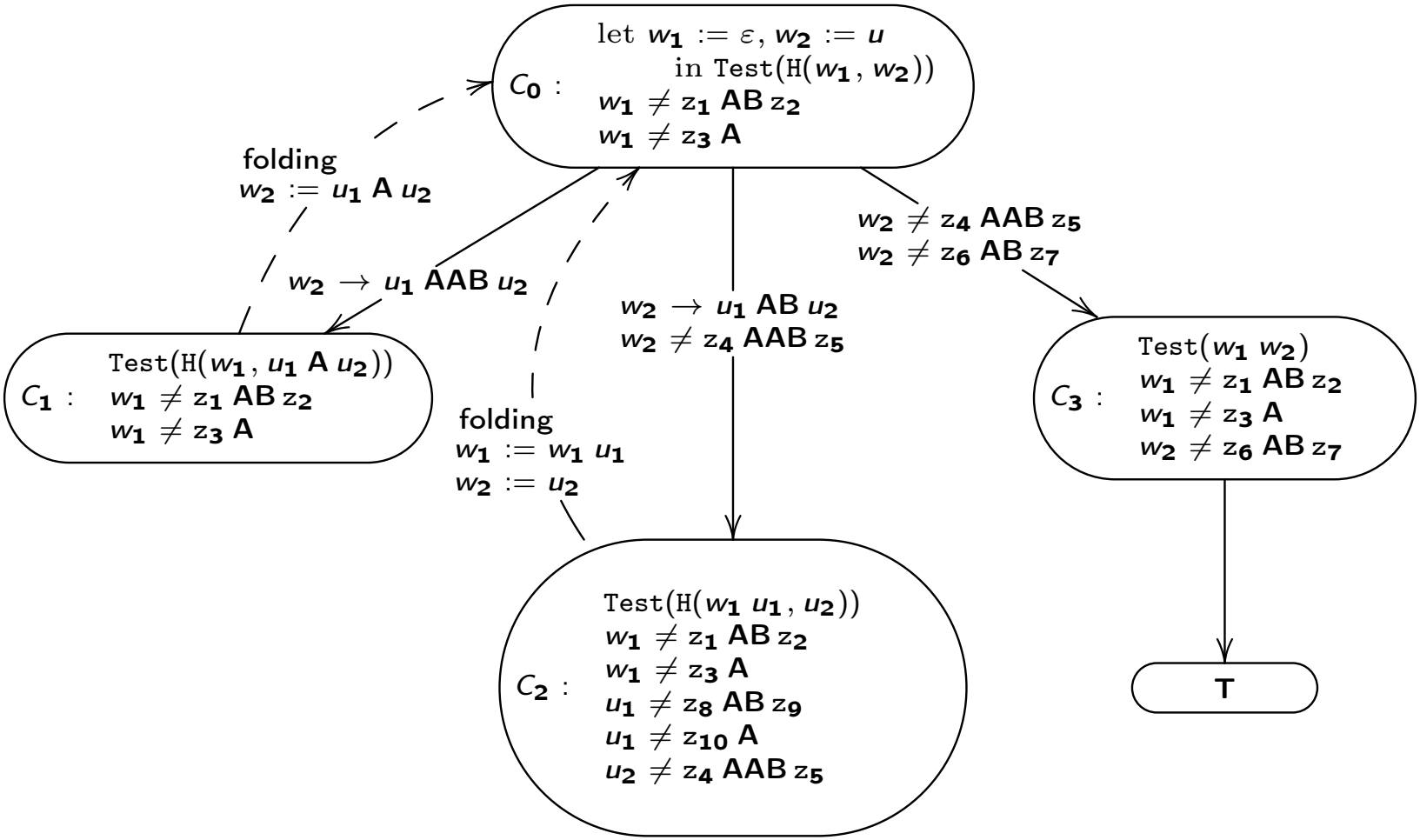


The implication is checked.

$$(w_1 \neq z_1\,\mathbf{AB}\,z_2)\ \&\ (w_1 \neq z_3\,\mathbf{A})\ \&\ (u_1 \neq z_8\,\mathbf{AB}\,z_9)\ \&\ (u_1 \neq z_{10}\,\mathbf{A}) \Rightarrow (w_1\,u_1 \neq z_1\,\mathbf{AB}\,z_2\ \&\ w_1\,u_1 \neq z_3\,\mathbf{A}).$$

It holds, hence the folding occurs.

Entire Residual Graph of Supercompilation



The residual graph does not contain configurations returning **F**.

The exhaustive deletion algorithm given in the function H is safe.

Two Residual Programs

$$
\begin{array}{lcl}
\mathtt{Go(x)} & = & \mathtt{Test(H(\varepsilon, x))};\\
\mathtt{H(x, y\,\mathbf{AAB}\,z)} & = & \mathtt{H(x, y\,\mathbf{A}\,z)};\\
\mathtt{H(x, y\,\mathbf{AB}\,z)} & = & \mathtt{H(x\,y, z)};\\
\mathtt{H(x, y)} & = & \mathtt{x\,y};\\
\mathtt{Test(x\,\mathbf{AB}\,y)} & = & \mathbf{F};\\
\mathtt{Test(x)} & = & \mathbf{T};
\end{array}
$$

$$
\begin{array}{lcl}
\mathtt{Go(x)} & = & \mathtt{Test_0(x)};\\
\mathtt{Test_0(x\,\mathbf{AAB}\,y)} & = & \mathtt{Test_0(x\,\mathbf{A}\,y)};\\
\mathtt{Test_0(x\,\mathbf{AB}\,y)} & = & \mathtt{Test_1(x, y)};\\
\mathtt{Test_0(x)} & = & \mathbf{T};\\
\mathtt{Test_1(z, x\,\mathbf{AB}\,y)} & = & \mathtt{Test_1(z\,x, y)};\\
\mathtt{Test_1(z, x)} & = & \mathbf{T};
\end{array}
$$

$$
\begin{array}{lcl}
\mathtt{Go(x)} & = & \mathtt{Test(G(\varepsilon, x))};\\
\mathtt{G(\varepsilon, \mathbf{AB}\,x)} & = & \mathtt{G(\varepsilon, x)};\\
\mathtt{G(x\,c, \mathbf{AB}\,y)} & = & \mathtt{G(x, c\,y)};\\
\mathtt{G(x, y\,c\mathbf{AB}\,z)} & = & \mathtt{G(x\,y, c\,z)};\\
\mathtt{G(x, y)} & = & \mathtt{x\,y};\\
\mathtt{Test(x\,\mathbf{AB}\,y)} & = & \mathbf{F};\\
\mathtt{Test(x)} & = & \mathbf{T};
\end{array}
$$

$$
\begin{array}{lcl}
\mathtt{Go(x)} & = & \mathtt{Test_0(x)};\\
\mathtt{Test_0(\mathbf{AB}\,x)} & = & \mathtt{Test_0(x)};\\
\mathtt{Test_0(x\,c\mathbf{AB}\,y)} & = & \mathtt{Test_1(x\,c, y)};\\
\mathtt{Test_0(x)} & = & \mathbf{T};\\
\mathtt{Test_1(\mathbf{A}, \mathbf{B}\,x)} & = & \mathtt{Test_0(x)};\\
\mathtt{Test_1(x\,c\mathbf{A}, \mathbf{B}\,y)} & = & \mathtt{Test_1(xc, y)};\\
\mathtt{Test_1(x, \mathbf{AB}\,y)} & = & \mathtt{Test_1(x, y)};\\
\mathtt{Test_1(x, y\,c\mathbf{AB}\,z)} & = & \mathtt{Test_2(x, yc, z)};\\
\mathtt{Test_1(x, y)} & = & \mathbf{T};\\
\mathtt{Test_2(x, \mathbf{A}, \mathbf{B}\,z)} & = & \mathtt{Test_1(x, z)};\\
\mathtt{Test_2(x, y\,c\mathbf{A}, \mathbf{B}\,z)} & = & \mathtt{Test_2(x, yc, z)};\\
\mathtt{Test_2(x, y, \mathbf{AB}\,z)} & = & \mathtt{Test_2(x, y, z)};\\
\mathtt{Test_2(x, y, z_1\,c\mathbf{AB}\,z_2)} & = & \mathtt{Test_2(x, y\,z_1c, z_2)};\\
\mathtt{Test_2(x, y, z)} & = & \mathbf{T};
\end{array}
$$

Termination Issue

Theorem

For every term γ, the set of generalizations of γ is finite (due to the choice of the minimal generalization).

Theorem

The length of every linear word inequality $u \neq P$ generated by the algorithm is uniformly bounded by the size of the input program.

Conclusion

The supercompilation methods are developed and implemented for programs modelling Markov normal algorithms.

- multiple occurrences of the character type variables;
- repeated string type variables in the right-hand sides;
- the constraints may be generated both from the syntax or from a program behaviour.

Future work

An entire supercompilation method for a language allowing multiple occurrences of string type variables in the patterns.

The model supercompiler

The page of the model supercompiler `MSCP-A`:

`http://refal.botik.ru/mscp/mscp-a_eng.html`